Computing Invariants with Transformers: Experimental Scalability and Accuracy

Vivien Maisonneuve[1], Olivier Hermant[2] and François Irigoin[3]

MINES ParisTech, France

Abstract

Using abstract interpretation, invariants are usually obtained by solving iteratively a system of equations linking preconditions according to program statements. However, it is also possible to abstract first the statements as transformers, and then propagate the preconditions using the transformers. The second approach is modular because procedures and loops can be abstracted once and for all, avoiding an iterative resolution over the call graph and all the control flow graphs.
However, the transformer approach based on polyhedral abstract domains encurs two penalties: some invariant accuracy may be lost when computing transformers, and the execution time may increase exponentially because the dimension of a transformer is twice the dimension of a precondition.
The purposes of this article are 1) to measure the benefits of the modular approach and its drawbacks in terms of execution time and accuracy using significant examples and a newly developed benchmark for loop invariant analysis, ALICe, 2) to present a new technique designed to reduce the accuracy loss when computing transformers, 3) to evaluate experimentally the accuracy gains this new technique and other previously discussed ones provide with ALICe test cases and 4) to compare the executions times and accuracies of different tools, ASPIC, ISL, PAGAI and PIPS.
Our results suggest that the transformer-based approach used in PIPS, once improved with transformer lists, is as accurate as the other tools when dealing with the ALICe benchmark. Its modularity nevertheless leads to shorter execution times when dealing with nested loops and procedure calls found in real applications.

Keywords: model checking, abstract interpretation, static program analysis, linear relation analysis, automatic invariant detection, loop invariant, transformer, benchmark

1 Introduction

Using abstract interpretation, invariants are usually obtained by solving iteratively a system of equations linking preconditions according to program statements. However, it is also possible to abstract first the statements as state transformers, and then propagate the preconditions using these transformers. The second approach is modular because procedures and loops can be abstracted once and for all, avoiding an iterative resolution over the call graph and all control flow graphs.

[1] Email: vivien.maisonneuve@cri.mines-paristech.fr
[2] Email: olivier.hermant@cri.mines-paristech.fr
[3] Email: francois.irigoin@cri.mines-paristech.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs

However, the transformer approach, based on polyhedral abstract domains [14,2], encurs two possible penalties: some invariant accuracy may be lost when computing transformers, and the execution time may increase exponentially because the dimension of a transformer is twice the dimension of a precondition. Polyhedral operators have a worst-case exponential complexity and transformers must deal with two values per variable, the value in the precondition, *i.e.* the past value, and the value in the postcondition, *i.e.* the new value, whereas preconditions require only the current value.

The purposes of this article are 1) to measure the benefits of the modular approach and its drawbacks in terms of execution time and accuracy using parametric examples and a newly developed benchmark suite for loop invariant analysis, ALICe [18], 2) to present a new technique developed to reduce the accuracy loss when computing loop invariants, namely control path transformers, 3) to evaluate the accuracy gains this new technique and older ones, previously discussed in [2] but not implemented, provide with ALICe test cases and 4) to compare the execution times and accuracies of different tools using either precondition propagation or transformer computation. Namely, we compare ASPIC [8], which is a standard abtract interpretation (AI) tool based on widening and acceleration, PAGAI [13], a SMT-based AI tool, the Integer Set Library, ISL [24], which is a library including a transitive closure for Presburger relations and PIPS [15,14], which is a compilation framework using polyhedral sets to abstract transformers and preconditions. The comparisons are difficult because the tools have different input and output languages, but this is dealt with by the ALICe framework.

The techniques considered in this paper to improve the accuracy of the transformer approach are the computation of transformers along different control paths to postpone convex hull operations until the precondition propagation phase, the iterative computation of new transformers based on previously computed preconditions [2], the exploitation of idempotent transformers [2] and the control simplification that can be obtained by splitting and specializing control nodes [17]. Although we strived to make this paper self-contained, it might be useful to read [2] first or when encountering difficulties.

The outline of the paper is the following. Since the transformer-based approach is unusual, we introduce it briefly in Section 2. We then present a first set of experimental results to explore the accuracy and execution time issues existing with the techniques presented in [2] (Section 3). Accuracy issues encountered can be traced back to early convex hull operations, and several techniques designed to postpone them as much as possible are detailed in Sections 4 and 5. Finally, we measure the impact of these improvements with ALICe (Section 6) and conclude.

2 Generation of Invariants with Transformers

Invariants are usually computed by propagating preconditions along the control paths of a program until stable preconditions are obtained for each control node. To avoid infinite propagations, special approximation operators, *e.g.* widening operators, are used to guarantee the convergence within a finite number of steps.

Instead, PIPS relies on a modular alternative approach using *affine transform-*

```
void foo(float x) {
    int n = 0;
    while (1)
        if (x)
            if (n<60)
                n++;
            else
                n = 0;
}
```

```
// T() {0==-1}
void foo(float x) {
// T(n) {n==0}
    int n = 0;
// T(n) {n#init==0}
    while (1)
// T(n) {n<=n#init+1}
        if (x)
// T(n) {n<=60, n<=n#init+1}
            if (n<60)
// T(n) {n==n#init+1, n<=60}
                n++;
            else
// T(n) {n==0, 60<=n#init}
                n = 0;
}
```

```
void foo(float x) {
// P() {}
    int n = 0;
// P(n) {n==0}
    while (1)
// P(n) {0<=n, n<=60}
        if (x)
// P(n) {0<=n, n<=60}
            if (n<60)
// P(n) {0<=n, n<=59}
                n++;
            else
// P(n) {n==60}
                n = 0;
}
```

Fig. 1. Statements, transformers and preconditions for `counter` (Halbwachs & *al.*)

ers [15], *i.e.*, polyhedra representing transfer functions. Transformers and preconditions are shown in Figure 1 for the `counter` example used by Halbwachs & *al.* [12], as comments just above the related statement. Transformers for elementary statements contain many equations because usually few variables are modified. These equations are kept implicit and instead the modified variables are listed as arguments. Finally, a `#init` suffix is used to distinguish the old value from the new value.

2.1 Generation of Transformers by PIPS

The algorithms used in PIPS assume no cycles in the call graph and proceeds as follows. First, each program command S, elementary or compound statement or procedure call, is over-approximated by an affine transformer $\mathcal{T}(S, P)$, possibly using information about a precondition P of S. This is a bottom-up procedure, detailed in the next paragraphs, because a default value can be used for the precondition P when no information is available. Each function is analyzed once and its transformer is reused at each call site. Then, preconditions are propagated from the program starting point using the transformers.

This approach can also be used with unstructured programs: the control flow graphs are either turned into equivalent structured graphs [1], or approximated and simplified by adding transitions, or analyzed using a decomposition into cycles [5].

The two-stage approach used by PIPS is the following. We suppose that elementary instructions have been turned into transformers, and show how control structures are handled.

Sequence Let x_i, x_i' and x_i'' denote values of variable x_i at different states. A sequence of affine transformers "T_1 followed by T_2" is overapproximated by the union of constraints in T_1 (on values $x_1, \ldots, x_n, x_1'', \ldots, x_n''$) with constraints in T_2 (on values $x_1'', \ldots, x_n'', x_1', \ldots, x_n'$), then projected on $x_1, \ldots, x_n, x_1', \ldots, x_n'$ to eliminate the "intermediate" values $x_1'', \ldots, x_n''$. 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 polyhedra is not a convex polyhedron). The best convex approximation is the convex union $T_1 \sqcup T_2$. This is a lossy operation in general.

Loop Given an affine transformer T for a loop body, the affine transformer T^* represents the effect of any number of iterations of T. It is computed by the Affine Derivative Closure algorithm [2].

The two main sources of imprecision that appear are the abstractions of loops ($*$) and parallel paths ($\sqcup$). Their impact is cumulated when multiple control paths appear within loops and nested loops.

Note that the pure bottom-up approach may be used or not. Since transformers are not computed concurrently but by traversing the AST, information gathered previously can be used right away. The range of a transformer or the condition of a test can be used as a precondition for the next statement to improve the accuracy of $\mathcal{T}$. This explains, for instance, why condition $n \leq 60$ appears in the transformer of statement `n++;`.

2.2 Generation of Invariants

Invariants, also known as preconditions, are forward propagated from the initial state of the program using the transformers computed during the previous phase. However, accuracy is improved when some preconditions are recomputed directly for compound statements. For instance, the postcondition of a conditional `if (`c`)` T_{T} `else` T_{F} can be obtained either as the convex hull of the postconditions of the two branches, $\mathrm{Post} = (T_{\mathrm{T}} \circ T_c)(\mathrm{Pre}) \sqcup (T_{\mathrm{F}} \circ T_{\neg c})(\mathrm{Pre})$, or as the precondition transformed by the conditional transformer, $\mathrm{Post} = (T_{\mathrm{T}} \circ T_c \sqcup T_{\mathrm{F}} \circ T_{\neg c})(\mathrm{Pre})$. The first equation provides more accurate results at little cost, because the branch postconditions are computed anyway.

3 Modularity and Accuracy: Experimental Results

PIPS has been developed as a compilation framework, able to process large applications interprocedurally [21]. It is important to check that the modularity provided by transformers results in the expected speed improvement and to measure the extent of its negative impact on accuracy. We show firstly that PIPS obtains accurate results in a small amount of time with respect to three other tools, ASPIC, ISL and PAGAI, when dealing with loop nests and procedure calls. We then recall previous experimental results showing a lack of accuracy when dealing with small test cases previously published to illustrate invariant generation algorithms [18].

3.1 Tools Used

When dealing with a state transition relation, a transitive closure is useful to compute invariants [6,2], to check loop termination or derive loop complexities [11], to build dependency constraints or to move convex array regions from a control point to another [16]. When dealing with a dependence relation, a transitive closure is useful to optimize or synthesize code [26,25,4].

It is difficult to compare the algorithms and heuristics used by different tools designed for one of these goals because they require different inputs and produce incompatible outputs. Also, the encoding of the input often impacts the analysis.

Depth	1	2	3	4	5	6	7	8	9
ASPIC	0.037	0.043	0.040	0.053	0.047	0.063	0.067	0.087	0.100
ISL	0.000	0.010	0.037	0.083	0.370	0.853	1.197	7.927	5.713
PAGAI	0.067	0.187	0.420	0.797	1.373	2.260	3.620	5.780	9.643
PIPS	0.004	0.009	0.015	0.021	0.030	0.039	0.053	0.071	0.090

Table 1
Times in seconds for invariant analyses of empty nested `for` loops with varying depths

We chose to compare PIPS with two standard abstract interpretation tools based on preconditions, ASPIC [9,8] and PAGAI [13], and with a Presburger-equivalent library, ISL [24,25], because it contains a powerful transitive closure heuristics designed initially to compute data dependence relations better than [23]. ASPIC and ISL's transitive closure functions use as input a state machine format, fsm, or a proprietary relational format. Unlike PAGAI and PIPS, they deal neither with the intricacies of C nor with their automatic abstraction.

Since the tools have very different structures and execution times, we report either directly the sum of the User and IO times reported by the `time` command for ASPIC, ISL and PAGAI, or the times obtained using `LOG_TIMINGS` for the transformer and precondition passes of PIPS. In this way, the C parsing part of PIPS is eliminated, as it is for ASPIC and ISL who use internal formats and for PAGAI who uses Clang for parsing, and we compare the execution time of the passes of PIPS that might be replaced by new passes based upon the other tools. Furthermore, the evolution of the execution times for each tool is fully relevant and interesting.

Benchmarks were run on a computer powered by a four-core Intel i7-2600 processor clocked at 3.40 GHz with 16 GB of memory, using ASPIC version 3.3, ISL version 0.12.2, PAGAI version 14-04-07 and PIPS revision 22 134.

3.2 Impact of Cycle Nesting on Convergence

Nested loops are common in scientific codes and easy to analyze with transformers. However, a 2D loop nest is used by Halbwachs in [12] to show improvements in widening techniques. Let us consider the code in the left part of Figure 2, a matrix multiplication. To check that all array accesses are safe, invariants on `i`, `j` and `k` are needed, which requires the analysis of a 3D loop nest.

To understand time behaviors of the tools, we measured the execution times of the transformer and precondition passes for nest depths of one to nine. Each loop has a zero lower bound and a symbolic upper bound, `for (i_k=0; i_k<b_k; i_k++)`, and the loop body is empty, `;`. No information about the upper loop bounds `b_k` is available: each loop can be either entered or skipped. The results are shown in Table 1. The execution time of tools using transformers or acceleration is not an affine function of the depth because the number of variables increases twice as fast as the depth and because the polyhedral operators are known for their exponential worst case complexities. However, it does not increase very fast, especially for the nesting depths usual in functions when inlining is not used.

```
void mm(int l, int n, int m,
        float A[l][m], float B[l][n],
        float C[n][m]) {
    int i, j, k;
    for (i=0; i<l; i++) {
        for (j=0; j<m; j++) {
            A[i][j] = 0.;
            for (k=0; k<n; k++)
               A[i][j] += B[i][k]*C[k][j];
        }
    }
}
```

```
void mp(int n, int p,
        float A[n][n], float B[n][n]) {
    int i, j, k;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            A[i][j] = B[i][j];
    for (k=1; k<p; k++) {
        float T[n][n];
        for (i=0; i<n; i++)
            for (j=0; j<n; j++)
               T[i][j] = A[i][j];
        mm(n, n, n, A, T, B);
    }
}
```

Fig. 2. Matrix multiplication and exponentiation: $A = B \times C$ and $A = B^p$

	main-1		main-2		main-3		main-4		main-5	
	2 calls	inlined	3 calls	inlined	4 calls	inlined	5 calls	inlined	6 calls	inlined
ASPIC	–	0.043	–	0.061	–	0.087	–	0.108	–	0.149
ISL	–	261.810	–	274.580	–	370.960	–	413.300	–	456.360
PAGAI	0.980	1.417	1.383	5.680	2.030	14.677	2.990	30.007	4.467	53.247
PIPS	0.048	0.043	0.049	0.063	0.048	0.084	0.050	0.108	0.051	0.127

Table 2
Times in seconds for interprocedural analyses and intraprocedural analyses of inlined versions

3.3 Interprocedural Analysis or Inlining

The code in the right part of Figure 2, a matrix exponentiation, `mp`, contains a call to a matrix multiplication, `mm`, within a loop. Function `mp` is called from a `main` function that reads a matrix and an exponent, and prints the resulting matrix. This code can be analyzed interprocedurally or intraprocedurally after inlining. Table 2 contains execution time measurements for main programs calling `mp` from one to five times. Each measurement was performed 10 times and the median time is displayed. The code is either analyzed interprocedurally or the callees are inlined, which reduces the number of call sites to zero. Modularity becomes more and more useful when multiple call sites of the same function are present in the analyzed program. Also, inlining increases the loop nest depths, which is bad for the execution time as seen in the section above.

3.4 Analysis of Accuracy Results Obtained with ALICe

In this section, we recall experimental results obtained with ALICe and published in [18]. The ALICe benchmark has been developed to assess the robustness and accuracy of various invariant generating tools. Version 1.0 supports three tools, ASPIC, ISL and PIPS, and contains 102 test cases gathered from papers dealing with loop invariant or termination. It uses the FAST format as neutral reference format and provide different encodings of each test case.

Regardless of the encoding, PIPS is the less accurate of the three tools, with 43 good results, versus 75 for ASPIC and 63 for ISL. But no tool is *strictly better* than another one: for each tool, at least one model is successfully analyzed only by this tool (see Figure 3). Moreover Table 3 shows no clear trend for the accuracy of generated invariants, measured by inclusion of invariant sets.

A closer analysis shows that ISL performs comparatively well on test cases en-

coded with *concurrent loops* (several loops on a single control point), unlike PIPS whose results are particularly bad on such cases. 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 formulæ, which it is not able to accelerate.

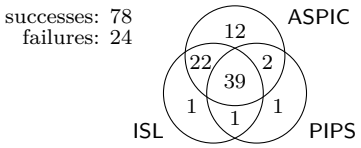

Fig. 3. Venn diagram for ALICe 102 test cases

$\supseteq_{\nearrow}$	ASPIC	ISL	PIPS
ASPIC	–	21	23
ISL	49	–	54
PIPS	33	23	–

Table 3. Invariant inclusions

3.5 Experimental Execution Times and Accuracies

The modular approach, as implemented in PIPS, is effective in terms of accuracy and execution time when dealing with large programs using function calls and nested loops [21]. However, it lacks accuracy when computing invariants for small transition systems often targeted in the literature about automatic invariant generation. The accuracy loss is mostly due to convex hulls performed in the transformer space before the transitive closure is approximated, but several integer overflows are also observed.

4 New Improvements in Transformer Computation

We present two new improvements for transformer-based analyses. The first one deals with concurrent loops, and the second one with integer overflows.

4.1 Concurrent Loops

We are dealing with structured code. A control path set is built for each loop body. When a test or a loop is found, each pre-existing control path is duplicated to take into account the true and false branches, or the loop entrance and skip. This is not performed recursively down the branches or the inner loop bodies. Thus the total number of control path is at most 2^k, where k is the number of (possibly compound) statements in the loop body.

4.2 Control-Path Transformers in Loops

Let us assume that a loop contains several control paths, each defined by its transformer T_i. Such loops are called *concurrent* loops above. The loop precondition, P^*, can be decomposed into a set of preconditions, each obtained after a variable number of iterations and merged together by convex hull operators, $\sqcup$:

$$P^* = P_0 \sqcup P_2 \sqcup P_+ \sqcup P_{3+} \tag{1}$$

P_0 is the precondition holding the first time the loop is entered. P_2 is obtained after two iterations with different control paths. P_+ corresponds to the case when

only one control path is used for all iterations; it includes P_1, the precondition of the second iteration. Finally, P_{3+} is the loop precondition corresponding to all the longer control paths with at least three iterations and two different control paths.

$$P_2 = \bigsqcup_i \bigsqcup_{j \neq i} T_i(T_j(P_0)) \qquad P_+ = \bigsqcup_i T_i^+(P_0) \qquad P_{3+} = \bigsqcup_i \bigsqcup_{j \neq i} T_i^+(T_j(C^*(P_1)))$$

C^* is an over-approximation of the transitive closure of the convex hull C of the transitions, $C = \bigsqcup_i T_i$ and P_1 is the precondition after one iteration exactly, $\bigsqcup_i T_i(P_0)$.

When the default formula in PIPS, $P^* = P_0 \sqcup C^+(P_0)$, is used, the convex hull operations are mostly performed before the transitive closure, in the transformer space. With Equation 1, they are executed later in the precondition space and each elementary transition T_i is applied as last transition. Hence, the information brought by idempotent transformations is preserved.

The formula used to compute the loop invariant has been unrolled to take into account explicitly up to three iterations. It is possible to generalize this to k steps, with an exponential increase in the number of terms. But we are lacking experimental cases justifying this development.

Halbwachs & *al.* present in [12] the example in Figure 1. The reset to 0 is abstracted by a transformer that cannot be merged accurately with a transformer abstracting a conditional increment and no information is obtained with the equations presented in [2]. However, Equation 1 provides an accurate loop invariant, $0 \leq n \leq 60$, by combining them, their transitive closures and the initial loop precondition.

4.3 Motivations for Equation 1

Equation 1, combined with the definitions of P_2, P_+ and P_{3+}, was developped:

(i) To transfer as much as possible convex hulls performed in the transformer space into convex hulls performed in the invariant (pre- and post-condition) space; for instance, the merge of an incrementation and a decrementation results in no information, regardless of the conditions used to control their executions.

(ii) C^*, the transitive closure of C, the convex hull of the transformers related to each control path, is often imprecise. Information, especially idempotent components, can be restored using each control path transformer T_i as last step for the iterations. This can be generalized to T_i^+.

(iii) In some cases, control path i can follow control path j, but j cannot follow i ($T_j \circ T_i = \varnothing$). By considering the two last different kinds of iterations, some impossible multi-iteration paths are eliminated.

4.4 Arbitrary-Precision Numbers

Intermediate computations may generate polyhedra with huge coefficients, leading to arithmetic overflows even with 64-bit integers, because constraint constants are transformed into coefficients by the convex hull operator. Eventually, when an overflow occurs, some constraints may be dropped, and the resulting invariant is less accurate than it should. To address this problem, we added GMP support to some

of the PIPS polyhedral operators. This seemingly purely pratical implementation decision allows for a drastical simplification of the polyhedral algorithms because overflow exceptions no longer have to be handled. It turns out that this positively impacts both the execution time (about 6 times faster now) and the comparison between transformer- and precondition-based analyses.

5 Other Improvements in Transformer Computation

We recall several improvements for transformer-based analyses. They can either be applied directly to the source codes, or to existing analyses. They have already been published in [2,17] but they all still required implementation and experimentation.

5.1 Control Restructuring by Node Splitting

The invariant information is attached to control nodes. If their number is increased, more precise invariants can be found when using polyhedral invariants because the global invariant is the disjunction of the node invariants. Also, the behavior of the program may become easier to analyze because fewer arcs may join the split nodes. However, the presence of new nodes usually increases the analysis time and a trade-off must be found between accuracy and node number.

Maisonneuve has developed a heuristics to split control nodes [17], which splits nodes only if some benefit can be expected. This heuristics, fsmnodesplit, which has already been proved and validated experimentally [17,18] but is not integrated into PIPS, is a prime candidate to improve PIPS results and to observe its robustness with respect to different equivalent encodings of test cases obtained by splitting, merging or merging and then splitting nodes. It is also relevant for other tools.

5.2 Iterative Analysis

It is sometimes possible to improve the preconditions by recomputing the transformers a second time, using the first set of preconditions as input to limit their domains and ranges.

As explained in [2], the iterative relationship between transformers and preconditions is formalized by the two equations below where B stands for the loop body statement and the continuation condition, $\mathcal{T}$ for the function that converts a statement into a convex transformer, P_0^* represents any state and n is positive:

$$T_{n+1}^* = \mathcal{T}(B, P_n^*) \cap P_n^*, \qquad P_n^* = P_0 \sqcup T_n\,(T_n^*(P_0)) \tag{2}$$

Note that the previous precondition P_n^*, which can be computed more precisely using Equation 1, impacts the transformer T_{n+1}^* in two different ways. The affine abstraction function $\mathcal{T}$ is sharpened and the domain of the resulting transformer also is restricted by the previous precondition, P_n^*.

The example in Figure 4 was published in [7]. Function `foo` has two different behaviors depending on the value of `flag` and it uses a non-affine condition in the loop, `i%2==0`. Different behaviors that depend on a formal parameter can be separated by cloning the function and values returned by the modulo operator can

```
void foo(int flag, float x) {
    int i, j = 1, a = 0, b = 0;
    if (flag) i = 0;
    else i = 1;
    while (x>0.) {
        a++; b += (j-i); i += 2;
        if (i%2==0) j += 2;
        else j++;
    }
    if (flag) assert(a==b);
}
```

Iteration 1:

```
while (x>0.) {
// P(a,b,i,j) {2a==i, j<=2a+1, a+1<=j}
```

Iteration 2:

```
while (x>0.) {
// P(a,b,i,j) {2a==i, 2a==j-1, 0<=a, b<=a}
```

Iteration 3:

```
while (x>0.) {
// P(a,b,i,j) {a==b, 2a==i, 2a==j-1, 0<=a}
```

Fig. 4. Example by Dilig & *al.*: source code and loop invariants obtained iteratively when `flag!=0`

```
int main() {
    int x=0, new=0, old=1, y=0, z=0;
    while (x<10) {
        if (new==0)
            y++;
        else
            z++;
        new = 1 - new;
        old = 1 - old;
        x++;
    }
    if (new==1 && old==0
            || new==0 && old==1)
        printf("property verified\n");
    else
        printf("property not found\n");
}
```

Iteration 1:

```
    while (x<10) {
// P(new,old,x,y,z) {new+old==1, y+z==x,
// 0<=new, new<=1, new<=x, x<=9, y<=x, 0<=y}
    }
// P(new,old,x,y,z) {new+old==1, x==10, y+z==10,
// 0<=new, new<=1, new<=y, y<=new+9}
```

Iteration 2:

```
    while (x<10) {
// P(new,old,x,y,z) {new+old==1, new+x==2y,
// new+z==y, 0<=new, new<=1, new<=y, y<=new+4}
    }
// P(new,old,x,y,z) {new==0, old==1, x==10, y==5,
// z==5}
```

Fig. 5. Periodic behavior: loop invariants and post-conditions

be analyzed precisely when information about the parity of `i` is known. When `flag` is not zero, it is possible to derive that `a` equals `b`. To obtain three equations in the function postcondition, transformers must be computed three times because each time a precondition makes the abstraction of a statement more precise.

5.3 Periodicity

Periodic behaviors occur in scientific computing, when, for instance, two sub-arrays are swapped, to avoid copying new values in the locations of old values at each time step `x`, `A[new][*]=f(A[old][*])`, at the cost of a mere swapping of the indices. To parallelize such programs (see column 1 of Figure 5), the compiler must find the invariant `new+old==1`, which is then used in data dependence testing together with the conflict equation, `new==old`, to show that `A[new][*]` and `A[old][*]` refer to different sets of locations.

As pointed out in [2], there are different ways to encode the swap, and the above invariant may be more or less easy to generate. However, regardless of the encoding, the behavior is always periodic. More information is preserved if the transitive closure of the loop transformer is computed as a function of one of its powers [2]. For instance, the square is useful for idempotent function and functions with a period of 2:

$$T^* = \left(T^2\right)^* \sqcup T \circ \left(T^2\right)^*, \qquad T^+ = T \sqcup \left(T^2\right)^+ \sqcup T \circ \left(T^2\right)^+.$$

This can be generalized to any power of T. As no application has yet required a larger periodicity, our current implementation in PIPS is limited to T^2.

```
void masse_vmcai_2014_14(int x) {
    while(x!=0) {
        if(x>0) x--;
        else x++;
    }
}
```

```
void masse_vmcai_2014_14_transformed(int x) {
    while(x!=0) {
        while(x!=0 && x>0) x--;
        while(x!=0 && x<=0) x++;
    }
}
```

Fig. 6. While-if to while-while conversion: Example by Massé and Cook & *al.*

Note finally that an iterative analysis (see Section 5.2) improves the invariant for the periodic function: the relationship between `y` and `z` is found (see column 2 of Figure 5). As for the Dilig example in Figure 4, and unlike what is claimed in [2], the iterations are not linked only to non-polyhedral invariants, but also to polyhedral invariants and they may converge.

5.4 While-If to While-While Conversion

This optimization, the conversion of tests into while loops inside a while loop, was also used in [2] and proved correct in the extended version [3], but it has not been implemented, neither explicitly with a program transformation nor implicitly when computing loop invariants. In fact, it may have a detrimental effect with respect to using the convex hull to obtain a unique loop transformer, unless control path transformers are used. The example in Figure 6 [20] shows how easier it is to prove the loop termination once the internal test has been changed into a pair of while loops.

6 Experimental Evaluation of the Improvements

Although PIPS has not been designed to analyze the small test cases used in academic papers dealing with loop invariants and termination, it is yet interesting to use them to measure the impact of the improvements presented above and to compare PIPS to other tools able to compute invariants. Also, test cases left unsolved are good starting points for designing new improvements in invariant generation.

6.1 Impact of Encoding and Improvements for ASPIC, ISL and PIPS

Table 4 provides the numbers of successes and the total execution times obtained with the four different encodings of ALICe benchmark [18] by ASPIC, ISL, the baseline of PIPS and PIPS with some of the improvements defined in Section 5. As expected, the execution times increase and the return is sometimes quite small for the test cases in ALICe version 1.0. However, using sets of control path transformer and a favorable encoding provides excellent results in numbers of test cases solved per second, and future benchmarks for invariant generation may have different profiles. Note that periodic analysis, not shown in Table 4, does not improve any of the test cases currently in ALICe. We believe that this benchmark should be regularly increased with new cases; any feedback will be appreciated.

Note also that the execution times of PIPS in Table 4 differ widely from those published in [18]. In [18], the execution time is:

$$t_{\text{WING}} = \sum_{i \in \text{ALICe}} \text{time}\Big(\text{PIPS}\big(\text{fsm2c}(case_i)\big)\Big)$$

	fsm	C		ASPIC	ISL	PIPS				
						Default	CP	IA	CP-IA	CP-IA-MP
Direct	5497	15482	Succ.	75	63	43	69	45	72	73
		4323	Time	10.9	35.5	6.1	7.8	18.5	19.6	151.4
Split	9348	757222	Succ.	79	72	50	72	56	75	77
		4199	Time	12.8	43.0	5.7	6.8	14.4	19.0	113.3
Merged	5579	15380	Succ.	59	70	40	66	44	67	68
		5187	Time	16.7	26.2	6.2	8.0	18.5	19.7	225.9
Merged	6206	38941	Succ.	70	83	63	79	65	80	82
& Split		4549	Time	11.3	40.8	6.6	9.6	16.8	23.0	222.2

Table 4
Successes and execution times for Direct, Split, Merged & Merged-Split versions of ALICe test cases and for ASPIC, ISL and PIPS with different options (Control Path transformers, Iterative Analysis, MultiPrecision). Sizes in number of lines are indicated on the left for FAST files and for C files generated by fsm2c. C file sizes are reduced by eliminating files greater than 1 KLOC and by a first PIPS pass (control simplification). Accuracy figures for ASPIC, ISL and the default version of PIPS are reproduced from [19] for comparison purposes. Execution times for PIPS are not measured as in [19].

that is the sum of the PIPS processing time for all test cases in ALICe converted into C by fsm2c. This is highly detrimental to PIPS because of its large startup time, because number of passes, such as parsing, are not performed by other tools and because fsm2c has also negative impacts.

The C encoding chosen for fsm2c implies the parsing of the `stdlib` header, an interprocedural analysis by PIPS, lots of unreachable code and some exponential blow-ups in the generated code sizes. In order to reduce the overheads incurred by PIPS, we eliminate test cases which cannot be regenerated in less than 1000 lines, we replace the functional encoding of aleas by uninterpreted expressions, we use PIPS to eliminate unreachable statements and we process all test cases in one call to PIPS. Formally, the execution time is now:

$$t = \text{time}\left(\text{PIPS}\left(\text{PIPS}\left(\text{substitute}\left(\bigcup_{\substack{i\in\text{ALICe}\\ \text{Csize}(i)<1000}} \text{fsm2c}(case_i)\right)\right)\right)\right)$$

which reduces the PIPS execution time by a factor of 7 to 10.

6.2 Analysis of PIPS Failures

Out of 102 test cases, the ALICe benchmark contains 9 test cases that involve non-Presburger invariants and PIPS finds invariant for 82. This leaves 11 cases to investigate further, namely: `halbwachs7`, `henry`, `metro`, `microsoftex2`, `microsoftex5`, `popeea`, `realheapsort`, `realheapsort_step2`, `subway`, `synergy_bad` and `ticket` (http://alice.cri.mines-paristech.fr).

ALICe is based on the FAST format and the heuristics fsm2c used to generate structured C from FAST control flow graphs may blow up exponentially, up to 300 KLOC (`metro`, `realheapsort`, `realheapsort_step2` and `subway` at the very least). When some test cases are written in C, using while loops as they were published, the invariants for `henry`, `halbwachs7` and `synergy_bad` [10] are found by PIPS.

Case `microsoftex2` is analyzed in [11] to detect loop termination. The required information is not an invariant but a transformer, and PIPS computes the required transformer. Case `microsoftex5` is also analyzed in [11], using non-convex transformers. It contains two nested while loops, and the internal loop may or not be the identity function. The information about the identity behavior is not preserved by

PIPS because control paths are not built recursively going down loops to keep their number small. This test case requires a different algorithm to construct control paths in PIPS.

Case `popeea` [22,11] has a non-convex invariant, which can only be found by a convex tool if new control points are added. The heuristics used by ALICe, `fsmnodesplit`, probably fails to discover a proper node splitting. Case `ticket` [6] is interesting because it is easy to make the invariant convex. However, the number of control paths is large and the convex hulls used by PIPS lead to overflows when the while loop is unrolled. The algorithm is described as a transition system and the way it may be coded in C is critical to its analysis by PIPS.

7 Conclusion

We have provided experimental results supporting the transformer approach. It is time-efficient with respect to the usual abstract interpretation approach based on precondition propagation, when large pieces of code with many functions and nested loops are analyzed.

However, accuracy is lacking for small test cases as we observed thanks to the ALICe benchmark. So we have presented one new technique, control path transformers, and its use to compute loop invariants, and several improvements in transformer closure and loop invariant generation, including the use of multiprecision numbers, iterative analysis, support for idempotent and periodic behaviors, and input re-encoding.

These improvements, most of them described in [2], have been implemented in PIPS and tested against the 102 test cases of the ALICe benchmark. The results presented in Section 6 show that three improvements have a positive effect and that PIPS is now about as accurate as more specialized tools such as ASPIC and ISL. So, even though ALICe is a very useful first step, the developement of a general benchmark for research about loop invariants, affine or not, is still an open issue. The choices made for ALICe and its heuristics, fsm2c, should be revisited to support larger test cases and scalability studies. For instance, it would be useful to store C version of the test cases to ease the addition of new cases and to simplify the use of C analyzers such as PAGAI and PIPS.

Finally, some test cases in ALICe are not handled by any of the three tools that we used, primarily because their invariants are not Presburger formulæ. Some are not handled by PIPS because the transformers or the invariants are not convex. Some more work is still needed to generate more invariants automatically with polyhedral-based transformer techniques while avoiding exponential blowups when generating C code from FAST files or when analyzing codes with control path transformers and/or iteratively. The `ticket` algorithm in particular, but also the control restructurations in general, deserve new advances.

Acknowledgments

We thank Paul Feautrier, Laure Gonnord and Sven Verdoolaege who helped us use their respective tools C2fsm, ASPIC and ISL. We also thank Pierre Jouvelot and our three reviewers for their very careful reading and excellent suggestions.

References

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

[2] Ancourt, C., F. Coelho and F. Irigoin, *A modular static analysis approach to affine loop invariants detection* (2010), pp. 3 – 16, NSAD 2010.

[3] Ancourt, C., F. Coelho and F. Irigoin, *A Modular Static Analysis Approach to Affine Loop Invariants Detection (extended version)*, Technical Report A-419, MINES ParisTech, CRI (2010).

[4] Bielecki, W., K. Kraska and T. Klimek, *Transitive closure of a union of dependence relations for parameterized perfectly-nested loops*, in: *Parallel Computing Technologies*, 2013 pp. 37–50.

[5] Bourdoncle, F., "Sémantiques des langages d'ordre supérieur et interprétation abstraite," Ph.D. thesis, École polytechnique (1992).

[6] Bultan, T., R. Gerber and W. Pugh, *Symbolic model checking of infinite state systems using Presburger arithmetic*, in: *Computer Aided Verification*, 1997 pp. 400–411.

[7] Dillig, I., T. Dillig, B. Li and K. McMillan, *Inductive invariant generation via abductive inference*, in: *OOPSLA '13*, 2013, pp. 443–456.

[8] Feautrier, P. and L. Gonnord, *Accelerated invariant generation for C programs with aspic and c2fsm*, Electron. Notes Theor. Comput. Sci. **267** (2010), pp. 3–13.

[9] Gonnord, L. and N. Halbwachs, *Combining widening and acceleration in linear relation analysis*, in: K. Yi, editor, *Static Analysis*, Lecture Notes in Computer Science **4134**, 2006 pp. 144–160.

[10] Gulavani, B. S., T. A. Henzinger, Y. Kannan, A. V. Nori and S. K. Rajamani, *Synergy: A new algorithm for property checking*, in: *SIGSOFT '06*, 2006, pp. 117–127.

[11] Gulwani, S. and F. Zuleger, *The reachability-bound problem*, in: B. G. Zorn and A. Aiken, editors, *PLDI* (2010), pp. 292–304.

[12] Halbwachs, N. and J. Henry, *When the decreasing sequence fails*, in: *SAS*, 2012, pp. 198–213.

[13] Henry, J., D. Monniaux and M. Moy, *Pagai: A path sensitive static analyser*, Electron. Notes Theor. Comput. Sci. **289** (2012), pp. 15–25.

[14] Irigoin, F., *Interprocedural analyses for programming environments*, in: *Environments and Tools for Parallel Scientific Computing* (1993), pp. 333–350.

[15] Irigoin, F., P. Jouvelot and R. Triolet, *Semantical interprocedural parallelization: an overview of the PIPS project*, in: *ICS '91*, 1991, pp. 244–251.

[16] Khaldi, D., "Automatic Resource-Constrained Static Task Parallelization," Ph.D. thesis, MINES ParisTech (2013).

[17] Maisonneuve, V., *Convex invariant refinement by control node splitting: a heuristic approach* (2012), pp. 49 – 59, NSAD 2011.

[18] Maisonneuve, V., O. Hermant and F. Irigoin, *ALICe: A framework to improve affine loop invariant computation* (2014), 5th Workshop on INvariant Generation.

[19] Maisonneuve, V., O. Hermant and F. Irigoin, *Alice: A framework to improve affine loop invariant computation*, Technical Report A-559, MINES ParisTech, CRI (2014).

[20] Massé, D., *Policy iteration-based conditional termination and ranking functions*, 2014 pp. 453–471, VMCAI.

[21] Nguyen, T. V. N. and F. Irigoin, *Efficient and effective array bound checking*, ACM Trans. Program. Lang. Syst. **27** (2005), pp. 527–570.

[22] Popeea, C. and W.-N. Chin, *Inferring disjunctive postconditions*, in: *ASIAN'06*, 2007, pp. 331–345.

[23] Pugh, W., *The Omega library*, http://www.cs.umd.edu/projects/omega.

[24] Verdoolaege, S., *ISL: An Integer Set Library for the Polyhedral Model*, http://freshmeat.net/projects/isl (2010).

[25] Verdoolaege, S., A. Cohen and A. Beletska, *Transitive closures of affine integer tuple relations and their overapproximations*, in: *SAS'11*, 2011, pp. 216–232.

[26] Wonnacott, D., *Extending scalar optimizations for arrays*, in: *Languages and Compilers for Parallel Computing*, 2001 pp. 97–111.