A Modular Static Analysis Approach to Affine Loop Invariants Detection[1] (Extended Version)

Corinne Ancourt Fabien Coelho
François Irigoin
CRI, Maths & Systems, MINES ParisTech

Abstract

Modular static analyzers use procedure abstractions, a.k.a. summarizations, to ensure that their execution time increases linearly with the size of analyzed programs. A similar abstraction mechanism is also used within a procedure to perform a bottom-up analysis. For instance, a sequence of instructions is abstracted by combining the abstractions of its components, or a loop is abstracted using the abstraction of its loop body: fixed point iterations for a loop can be replaced by a direct computation of the transitive closure of the loop body abstraction.

More specifically, our abstraction mechanism uses affine constraints, i.e. polyhedra, to specify pre- and post-conditions as well as state transformers. We present an algorithm to compute the transitive closure of such a state transformer, and we illustrate its performance on various examples. Our algorithm is simple, based on discrete differentiation and integration: it is very different from the usual abstract interpretation fixed point computation based on widening. Experiments are carried out using previously published examples. We obtain the same results directly, without using any heuristic.

1 Introduction

Program analyzes such as interprocedural program parallelization [21, 20], array access bound checking [26], array initialization checking, aliasing checking [25] require some mechanism to approximate loop behaviors. In order to obtain a modular analyzer and to limit analysis times, we depart from the usual approach [8] and compute state transformers instead of state predicates, *i.e.* pre- and post-conditions. Transformers are used to summarize functions: each function is analyzed once and its transformer is reused at each call site. Preconditions are then propagated using the transformers. Since transformers require

[1]This work is funded by the ACI SI as part of the APRON project. URL:http://www.cri.ensmp.fr/apron

twice as many variables as preconditions, we use polyhedra as finite abstractions of possibly infinite sets of states to maintain a sufficient accuracy.

In Section 2, we present a simple yet effective algorithm to compute transitive closures of transformers, which are then used to derive affine loop invariants. Then we show how to improve its effectiveness by using equivalent but different formulae for postconditions. They are equivalent when the analysis is exact, but they differ when approximations, such as affine approximations, are made. Several kinds of extensions are considered in Section 3. They are related to the transitive closure algorithm. Related work is introduced in Section 4 and we show on previously published examples that our algorithm provides the expected loop invariants without using any widening heuristic.

2 Simple Transitive Closure of Affine Transformers

Pugh and al. studied the transitive closure of transfer functions defined by Presburger formulae [22]. Here, transfer functions are approximated by affine relations. The graph of the relation between the initial state and the final state is defined by a polyhedron, i.e. a set of affine equalities and inequalities. Below, once transformers and preconditions are defined, we present our algorithm to compute transformer closures for while loops together with its proof. We illustrate its working on a motivating example, a safety controller for a toy robot car [17].

2.1 Affine Transformers and Preconditions

Each program command, elementary or compound statement or procedure call is approximated by an affine transformer. The underlying mechanism is similar to [8] but extended from the states to state transitions. The idea of transformers is quite general and is also used, for instance by Boigelot & al. [4].

The set of possible program states, before a command is executed, is defined by a precondition. The set of program states after the command execution is defined by a postcondition. The postcondition is the image of the precondition by the command transformer. A legal affine abstract postcondition contains the effective postcondition, i.e. it is an over-approximation.

For simplicity of exposure, the relationship between identifiers and memory locations is assumed to be a one-to-one mapping for scalar variables. In this paper, we deal only with integer scalar variables and states taking values in $\mathbb{Z}^n$, where the dimension n is the number of analyzed variables.

For semantic analysis purposes, control flow graphs are structured as while loops, e.g. using Bourdoncle's heuristic [5]. Other structured loops are decomposed into while loops. As a result, the only control structure with an iterative behavior studied here is the simple while loop.

2.2 The Affine Derivative Closure Algorithm

```
transformer T*(x,x') affine_derivative_closure(transformer T(x,x'))
{
  // add the difference vector dx
  transformer Q(x,x',dx)  = T(x,x') ^ (dx = x'-x);
  // eliminate the initial and final states x and x'
  transformer T'(dx) = project((project(Q(x,x',dx), x), x'));
  // compute dx for any iteration number k
  T'(dx) =  multiply_constant_terms(T'(dx), k) ^ (k >= 0);
  // eliminate the iteration number k and substitute back dx by x'-x
  return project(project(T'(dx),k) ^ (dx = x'-x), dx);
}
```

Figure 1: Affine Derivative Closure Algorithm

Our algorithm is outlined in Fig. 1 and works as follows:

Let us assume that T is a valid affine transformer for a while loop body and its continuation condition. T includes the loop entry condition, or at least an affine approximation of this condition. Let k be the iteration number, x^{k-1} be the integer memory state when the loop body is started the k-th time, and x^k be the final state when the loop condition is evaluated to true again. The predicate $T(x^{k-1}, x^k)$ holds for all possible $k > 1$.

Let δx be $x^k - x^{k-1}$ and $T'(\delta x)$ be the projection of $T \wedge \delta x = x^k - x^{k-1}$ along x^k and x^{k-1}. Note that T' does depend neither on k which is not a component of the memory state x nor on the names x^k and x^{k-1}, which have been eliminated by the projection.

Let x^0 be the state on loop entry. The state x^k that may be reached after k iterations of the loop, if such an iteration is executed, is:

$$x^k = x^0 + \sum_{i=1}^{k} \delta x^i \quad with\ \delta x^i = x^i - x^{i-1} \tag{1}$$

For all positive integers i, $T'(\delta x^i)$ holds. Since T' is a polyhedron, it can be defined by affine equalities and inequalities:

$$T'(\delta x) = \{\delta x \,|A\delta x = b \wedge A'\delta x \leq b'\} \tag{2}$$

where A and A' are integer matrices and b and b' the corresponding constant terms.

Multiplying Eq.(1) by the matrices A and A', we have:

$$Ax^k = Ax^0 + \sum_{i=1}^{k} A\delta x^i \qquad and \qquad A'x^k = A'x^0 + \sum_{i=1}^{k} A'\delta x^i \tag{3}$$

Since $A\delta x^i$ and $A'\delta x^i$ are equal to b or bounded by b', we have:

$$Ax^k = Ax^0 + kb \qquad and \qquad A'x^k \leq A'x^0 + kb' \tag{4}$$

The loop transformer $T^*(x^0, x)$ may be approximated by:

$$T^*(x^0, x) \subseteq \left\{ (x^0, x) \,\middle|\, \exists k \in [0, \infty[\ Ax = Ax^0 + kb \ \wedge\ A'x \leq A'x^0 + kb' \right\} \quad (5)$$

The set $P^*(x^0)$ of states reachable from x^0 by executing the loop may thus be approximated by:

$$P^*(x^0) \subseteq \left\{ x \,\middle|\, T^*(x^0, x) \right\} \quad (6)$$

Then the whole loop transformer $T^*(x^0, x)$ is over-approximated by projecting k from the constraints in Eq.(5). And an affine over-approximation of the loop postcondition is obtained by projecting x^0 too and by adding a safe approximation of the loop last iteration and exit condition.

Loop invariants are obtained for each state dimension i such that b_i is zero since then Eq.(4) shows that $(Ax^k)_i = (Ax^0)_i$. If there exists a dimension j such that b_j is not zero, then the iteration number k can be derived from a combination of variables and substituted everywhere else to obtain more invariants.

If not, k is still bounded by $k \geq 0$ and some inequalities can be saved according to the Fourier-Motzkin elimination rule. If some term $a'\delta x$, where a' is a row of A', is upper-bounded by a negative constant, or lower-bounded by a positive constant, monotony constraints are obtained. Strict monotonicity leads to loop termination proofs when the derivatives of the affine components of the while condition that imply non-termination are incompatible with T'.

Note that $T^+ = T^* \circ T$ can be computed by setting $k \geq 1$ in Eq.(5). Transformer T^+ may contain strict monotonicity conditions, which are useful for dependence testing in automatic loop parallelization [30] and array bound checking [29].

2.3 An Example: Robot Car Safety

Let us take the toy example described in [17] and recently reused by [23]. A robot car must follow autonomously a track painted on the floor. In case it loses the track, it should not crash against a wall; however it is not stopped right away since the track might be found again. The car should not accelerate too much when it is looking for the lost track. The safety controller must ensure that a limited amount of time is allowed to search the painted track at bounded speed. Since time and speed are bounded in the track search mode, the car is safe if the track is far enough from the walls.

```
1:   int s = 0, t = 0, d = 0;
2:   while(s <= 2 && t <= 3)
3:      if(alea())
4:         t++, s = 0; // increment time, reset speed estimation
5:      else
6:         d++, s++;   // meter increment, speed estimation increment
```

Figure 2: Car safety example

Let t be the time in seconds, d the distance from the starting point in meters and s the current estimation of the speed in meters per second. A model of the controller ensuring the physical safety of the car is encoded in C as shown in Fig. 2. Function *alea* is used to model a random event: either the clock counter is going to tick for the next second and the time is incremented while the speed estimation is reset to 0, or the distance and the speed estimation are increased because another meter has been reached. The safety is enforced by the loop guard. If nothing else happens within three seconds or if the speed is greater than two meters per second, the car is stopped. If walls are 10 meters away from the starting position, the car cannot reach a wall.

We explain the steps performed here by our Affine Derivative Closure Algorithm using a primed notation that distinguishes the values of each variable between the old and primed new state. If x is the memory state of (d, s, t), the transformer for the first branch of the test (line `4`) is:

$T_4(x, x') = \{s' = 0, t' = t + 1, d' = d, s \leq 2, t \leq 3\}$

For the second test branch (line `6`) transformer is:

$T_6(x, x') = \{d' = d + 1, s' = s + 1, s \leq 2, t' = t, t \leq 3\}$

Their convex hull used to approximate their combined effect is:

$T_3(x, x') = \{d'+t' = d+t+1, s+3t+1 \leq s'+3t' \leq 3t+3, t \leq t' \leq t+1, t \leq 3\}$

Projecting the old and new state, this transformer is rewritten as:

$T'(\delta x) = \{\delta d + \delta t = 1, 1 \leq \delta s + 3\delta t, 0 \leq \delta t \leq 1\}$

which leads to $\delta d + \delta t \leq \delta s + 3\delta t$, or $\delta d \leq \delta s + 2\delta t$. This is the speed equation we looked for to prove the car safety. If the speed and the time are bounded, the distance travelled is bounded. Here, the numerical speed bound of 2 produces a linear speed equation. Note that the time bound can be a parameter n and the distance is also found bounded by $2 * n + 3$ (see Example 10). This extends Halbwachs' car safety case.

2.4 Discussion

The algorithm is very simple yet powerful enough to derive non trivial conditions. Its weaknesses come from 1) computing T' as a relation on δx instead of a more accurate relation on $(\delta x, x)$ to ease the summation and stay in the affine setting, and 2) in computing T in the first place as an affine transformer using the convex hull to model tests. The complexity of the algorithm is dominated by the complexity of the projection steps: its worst case is exponential with the number of variables projected, but in practice it is polynomial when the constraints are sparse.

3 From Transformers to Loop Invariants

Several simple extensions are useful to cope with non-affine behaviors such as iteration independent assignments or periodic and polynomial behaviors. They occur when an iteration independent assignment is equivalent to a differential

assignment, when two (or more) buffers are used in a flip-flop mode or when triangular matrices are accessed. Note that polynomial behaviors [15] are frequent when accessing symmetric matrices, but that monotonicity or strict monotonicity information is often sufficient to make a decision about data dependence or array bound overflow issues.

3.1 Using T^+ instead of T^*

If the loop w is certainly entered when reached with precondition P^w, that is when the affine approximation of the negation of its condition combined with P^w generates a contradiction, it is better to compute T^+ instead of T^*. The constraints on the image of T can be added to the image of T^+, but not of T^*.

In a loop such as "`while(alea()) m = 10;`" no information on `m` is gathered in the postcondition because its value may be unchanged, when the loop is not entered, as well as set to 10.

Note that $T^+ = T \circ T^* = T^* \circ T$ when T is exact, but that the first formula is more precise with an approximate T. In the second case, the information added by T may be lost by T^*. Also it is better to use:

$$P^* = P^0 \bigsqcup T^+(P^0) \tag{7}$$

rather than the equivalent formula $P^* = T^*(P^0)$ when the range of T and P^0 have common constraints. The convex hull operator $\bigsqcup$ is used instead of the union operator, which is not internal for polyhedra. Since it is not accurate, it should always be applied as late as possible when equivalent formulae are available.

3.2 Periodic Behaviors

Periodic behaviors are observed when a variable or a set of variables is used to flip-flop the accesses to two or more buffers; for instance this is often used in signal processing applications to switch between receiving or sending and computing buffers, or in scientific programs to switch between new and old values [4, 3], as depicted in Fig. 3.

```
double x[2][10];
int old = 0, new = 1, i, t;
for(t = 0; t<1000; t++) {
  for(i = 0; i<10;i++)
    x[new][i] = g(x[old][i]);
  old = new, new = 1 - old;
}
```

Figure 3: Flip-flop example

The t loop is parallel if the value of *new* is proven to be always different from the value of *old* because $old + new = 1$, which is found thanks to Eq.(7).

An interesting aspect in flip-flop analysis is its robustness with respect to the flip-flop encoding scheme. Ideally, different encodings leading to the same execution traces should produce the same analysis result. Different encodings use different mathematical functions, as illustrated by Figure 4, but they have the same value over the useful subset of their domains. Thus the analysis result depends on the accuracy of the loop precondition used to characterize the effective function domains.

1	`new = old; old = 1-old;`
2	`new = 1 - new; old = 1 - old;`
3	`t = new; new = old; old = t;`
4	`if(new==1) { new = 0, old = 1; } else { new = 1, old = 0; }`

5	`if(new==1) { new = 0; old = 1; }`
	`else { if(new==0) { new = 1; old = 0; } else`
	`exit(1); }`

6	`new = (new+1) %2; old = (old+1) %2 ;`

Figure 4: Six different encodings of **flip-flop** operations

The invariant $new + old = 1$ is found by our tool PIPS [24] for Cases 1, 3, 4 and 5 thanks to Eq.(7). But it fails for Case 6 because the modulo operator is not analyzed as well as a multiplication or a division: the sources of failure are not limited to convex hulls and transitive closures.

Case 2 requires a loop unrolling of two to obtain the invariant. Larger periods can be obtained using integer rotation matrices or ad hoc constructs. More generally, k-periodic behaviors can be captured by computing T^* and T^+ as:

$$T_k{}^* = \bigsqcup_{i=0,k-1} T^i \circ \left(T^k\right)^* \qquad T_k{}^+ = \bigsqcup_{i=1,k} T^i \circ \left(T^k\right)^* \tag{8}$$

This is equivalent to a loop unrolling of degree k and similar to a delayed widening. These definitions can be used to refine Eq.(7) and to obtain better loop preconditions. With $k = 2$, the precondition becomes:

$$P^* = P^0 \bigsqcup T(P^0) \bigsqcup T^{2+}(P^0) \bigsqcup T\left(T^{2+}(P^0)\right) \tag{9}$$

With Eq.(9), PIPS is able to deal with the second encoding of flip-flop because `new` and `old` are invariant by T^2. The effective period does not have to be known as each T_k^* is a proper over-approximation of T^* and their intersection can be used:

$$T^* = \bigcap_{i\in[1,k]} T_i{}^*$$

The same holds for T^+ and for the loop precondition. Equation (9) can be generalized at order k:

$$P_k^* = \left(\bigsqcup_{i=0}^{k-1} T^k(P^0) \right) \bigsqcup \left(\bigsqcup_{i=0}^{k-1} T^i \left(T^{k^+}(P^0) \right) \right)$$

and all preconditions P_k^* obtained with T_k^* and this equation can be intersected since each of them is a valid loop precondition. The user must provide a maximal value k_{max} for k to obtain

$$P^* = \bigcap_{k \in [1, k_{max}]} P_k^*$$

and this process can capture invariants about cyclic variables with periods less than k_{max}. Note that different sets of variables can have different periods in the same loop: there may be no ideal k and the intersection is necessary to capture all the information available.

It should also be noted that all invariants of T^* are also invariants of $(T^k)^*$. No information is lost when a power of T is used instead of T.

Invariant Completeness Theorem *Any invariant found by our Affine Derivative Closure Algorithm for transformer T^* is also found by the same algorithm for transformer $(T^k)^*$.*

Lemma *If the elimination of variable z in a linear constraint system S by Fourier-Motzkin elimination produces the new system S', the elimination of z in system S modified by multiplying all z coefficients by a positive integer produces the same S'.*
In other words, the Fourier-Motzkin elimination of a variable z is not perturbed if all coefficients of z are multiplied by the same positive constant.

Lemma's Proof Let S be $Ax \leq zb$ (since S is linear, there are no constant terms). S is decomposed into: $\{A^+x \leq zb^+, A^-x \leq -zb^-\}$ with $b = b^+ - b^-$.

The new constraints are built as $\left\{a_i^+x \leq zb_i^+, a_j^-x \leq -zb_j^-\right\}$, which leads to $b_j^- a^+x + b_i^+ a_j^- x \leq 0$.

If all coefficients of z are multiplied by the same positive constant c, the decomposition in A^+ and A^- is not modified because c is positive and the new relations are equal to the old ones $\left\{a_i^+x \leq czb_i^+, a_j^-x \leq -czb_j^-\right\}$, and the inequation $cb_j^- a^+x + cb_i^+ a_j^- x \leq 0$ which can be divided by c. QED

Theorem Proof $T(x, y)$ is a polyhedral transformer leading to $T'(y - x) = \{(y - x)|A(y - x) \leq b\}$ [no need to distinguish between equations and inequalities]. Then $(T^k)'$ verifies $(T^k)'(y - x) = \{(y - x)|A(y - x) \leq kb\}$. This is true for $k = 1$ and 2. If it is true for k, then $(T^{k+1})'(z-x) = (T^k)'(y-x) \wedge T'(z-y)$. Since $A(y - x) \leq kb$ and $A(z - y) \leq b$, $A(z - x) \leq (k + 1)b$. So the constraint

systems defining T' and $(T^k)'$ differ only because the coefficients of z are multiplied by k. Using Lemma 1, the elimination of z (last step of the algorithm) leads to the same T^*. Hence all invariants of T^* are included in invariants of $(T^k)^*$. QED

3.3 Higher-Order Differences

The scheme could be first generalized to second order differences by setting:

$$T(x,x') \wedge \delta x = x' - x \wedge T(x',x'') \wedge \delta x' = x'' - x' \wedge \Delta x = \delta x' - \delta x$$

Let us consider the non linear example on the left hand side of Fig. 5. One possible application of such differences is to prove that variable `i` is bounded and reaches its maximum on the loop boundary or when its discrete difference is zero. Indeed since the second difference of `i` is the difference of `j`, -1, and is negative, sooner or later, `i` is going to decrease.

```
int i = 0, j = 2, k = 1;
while(k<=10)
  j--, i += j, k++;
```

```
int i = 0, j = 0, n;
if(n<0) exit(1);
  while(i<=n)  i++, j+=i;
```

Figure 5: Non linear (left) and parabolic (right) examples

Exact closed form polynomials are computed in [15] and [28], but the polynomial closed form would be uselessly complicated to use for this purpose, although admittedly mandatory for code generation after automatic parallelization.

3.4 Monotonicity and Iterative Analysis

Postcondition $\{j = -8, k = 11\}$ is directly derived from the code in the left of Fig. 5. It does not bound i. However, if the transformers are recomputed with this precondition, the loop postcondition is refined iteratively as shown on the left hand side of Fig 6.

A similar code, without loop numerical bound, on the right hand side in Fig. 5, can also be analyzed iteratively, but without ever reaching a fixed point, as shown on the right hand side of Fig 6.

1	$\{j = -8, k = 11, 0 \leq i + 71, i + 14 \leq 0\}$	1	$\{i = n + 1, 1 \leq i\}$
2	$\{j = -8, k = 11, 0 \leq i + 71, i + 25 \leq 0\}$	2	$\{..., 2i \leq j + 1, 3i \leq j + 3, 4i \leq j + 6\}$
3	$\{j = -8, k = 11, 0 \leq i + 71, i + 32 \leq 0\}$	3	$\{..., 5i \leq j + 10, 6i \leq j + 15\}$
4	$\{j = -8, k = 11, 0 \leq i + 71, i + 35 \leq 0\}$	4	$\{..., 7i \leq j + 21, 8i \leq j + 28\}$
5	$\{j = -8, k = 11, 0 \leq i + 71, i + 35 \leq 0\}$	5	$\{..., 9i \leq j + 36, 10i \leq j + 45\}$

Figure 6: Results for the non linear and parabolic examples of Fig. 5

The iterative relationship between transformers and preconditions is formalized by the next two equations where B stands for the loop body statement and

the continuation condition, and $\mathcal{T}$ for the function that converts a statement into a convex transformer:

$$T^*_{n+1} = \mathcal{T}(B, P^*_n) \wedge P^*_n \qquad P^*_n = P^0 \sqcup T_n \left(T^*_n(P^0)\right) \tag{10}$$

Note that the previous precondition P_n impacts the transformer T_{n+1} in two different ways. The affine abstraction $\mathcal{T}$ is sharpened and the resulting transformer also is restricted by the previous precondition.

The iterative refinement process does not always converge and it may even lead to a precision loss, due to magnitude overflows. These are not handled with a good heuristic in the present PIPS implementation, but it is not critical as the refinement process is not automatic: it must be specified by the user.

3.5 Postponing Convex Hulls

If a loop contains a test, the test is abstracted by a convex hull and the transitive closure is applied later. In other words, the convex hull loses information at the very beginning of the invariant computation.
Hence it is useful to convert: `while(c) if(t) a; else b;`
into the equivalent: `while (c) { while (c&&t) a; while (c&&!t) b; }`

This transformation, which is somehow similar to the *abstract acceleration* defined by Laure Gonnord [10] after the acceleration used in model-checking [4, 6, 7], eliminates the early convex hull and lets PIPS find the proper invariants for cases 1 in [11] and 2 in [14, 13]. This transformation can be applied to the car safety example used in Section 2.3 and its parametric extension (See Figure 10). The safety property is proved again by PIPS in both cases.

4 Related Work

PIPS [24] development has been driven for almost twenty years by the needs of automatic analysis and parallelization for large size real-life Fortran and C programs of up to hundreds of functions and 100 KLOCS. Our derivative algorithm is used for all loops and on most of the control-flow graphs, after restructuring.

Its input is a deterministic C or Fortran program and not a non-deterministic finite automata as used in model-checking benchmarks or by Gonnord [10]. It is difficult to be sure to convert an automaton into a program without performing some intelligent structuring that may turn out to be the key to its successful analysis. Moreover, the PIPS semantic analyzer uses Bourdoncle's algorithm [5] to deal with unstructured control flow graph. This increases the number of convex hulls to decrease the number of widenings, and we now know that it often prevents acceleration opportunities. Most of our comparisons are based on published pieces of code, not on transcoded automata.

Fig. 7 presents examples found in the literature [8, 10, 11, 12, 14, 16, 17, 23, 13] about the widening operator and its improvements. The relations found by PIPS are given in the third column: they are obtained in less than a second on a typical PC, and are equivalent to those of the other tools.

Original example	After restructuration	PIPS

(1) *Gopan'07 [11], Gulwani'09 [13]*
`x=y=0;`
`while(*) {`
`  if (x ≤ 50)  y++;`
`  else y--;`
`  if (y < 0) break;`
`  x++;`
`}`
	`x=y=0;`
	`while (y ≥ 0) {`
	`  while (y ≥ 0 && x ≤ 50)`
	`  y++;`
	`  x++;`
	`  while (y ≥ 0 && x > 50)`
	`  y--;`
	`  x++;`
	`}`
	`x--;`
		$x = 102$

(2) *Gulwani 2007 [14, 13]*
`x=1; y=50;`
`while(x < 100) {`
`  if (x < 50) x++;`
`  else { x++; y++; }`
`}`
	`x=1; y=50;`
	`while (x < 100) {`
	`  while (x < 50) x++;`
	`  while (x < 100 && x ≥ 50)`
	`  { x++; y++; }`
	`}`
		$y = 100$

(3) *Gulavani 2006 [12], Gulwani 2009 [13]* `x=1; lock=y=0; while (x ≠ y) { lock=1; x=y; if (*) lock =0; y++; }`		`lock` $= 1$

(4) *Gaz Burner - Chaochen [31], Gonnord [10]*
`t=l=x=0; while(*) {`
`x=0; while (x ≤ 9 && alea()) x++, t++, l++;`
`x=0; while (x ≤ 49 || alea()) x++, t++;}`
		$6l \leq t + 5x$

(5) *Halbwachs [16]* `x=y=0; while(x ≤ 100) { if (alea()) x = x+2; else x++, y++; }`		$2 \leq x + y$ $y \leq x$ $x + y \leq 202$

(6) *Halbwachs [16]*
`x=y=0;`
`while (x ≤ 100) { if (alea()) x = x+4; else x=x+2, y++; }`
		$4 \leq x + 2y$
		$2y \leq x$
		$x + 2y \leq 204$

(7) *Robot car safety example* *Halbwachs [17], Merchat [23], Gonnord [10], Section 2.3-Fig. 2*		$d \leq s + 2t$
(8)*Subway example* *Halbwachs [19], Gonnord [10]*		$20 \leq b$ $b - s \leq 20$

Figure 7: PIPS Experimental results

Examples 1, 2 and 3 illustrate the need to compute disjunctive invariants. Examples 5 and 6 express linear invariants. Some periodic cases are presented in Section 3.2. Examples 4 and 7 and the Subway example [19] characterize automata with more complex invariants, and our results are equivalent to [10] (p. 115). However our algorithm does not find accurate results with simple C encodings of automata such as the bakery mutual exclusion algorithm [6].

Using polyhedra instead of Presburger arithmetic, we do not claim to obtain

more accurate results than others. Our philosophy is to use real-life cases, avoiding artificial or contrived examples. We only claim our simple and direct algorithm gets the same results as iterative approaches like widening.

The concept of abstract acceleration introduced by Gonnord in [10] is very similar in its goal to our algorithm, but it is implemented by pattern matching for different specific cases (see chapters 5 to 7 in [10]), whereas we can deal with function calls and any control construct as the loop body transformer is computed in a modular way. Also, the exploitation of the accelerated cycles is part of a heuristic and not a program transformation as in Section 3.5. And the final result is obtained iteratively. All examples found in [10] are successfully processed by our algorithm, including the swimming pool [9].

Kelly & al. [22] present an algorithm to compute the transitive closure of a relation encoded by a Presburger formulae. This heuristic includes the notion of *d-form relation* which leads to an explicit transitive closure. It is stated that any relation can be put in a *d-form* at the expense of accuracy. We show here that it is not necessary to put the relation into a *d-form* to obtain an explicit transitive closure. We explain how to transform any relation into constraints about the state evolution and finally we explain how to get precise results by postponing convex hull operations.

Bielecki & al. [2] describe a procedure to obtain exact non-linear transitive closures, but only for normalized relations written as systems of recurrence equations and solved by Mathematica. The related work is limited to [22] and a few examples are given. Regardless of the still unknown generality and practicality of this procedure, its results would require some processing to be used within an affine-base analyzer or algorithm as found in abstract interpretation and automatic parallelization.

```
// T() empty
void multiply01() {
  // T(m) {m==1}
  int m = 1;
  // T(m) {m#init==1, m<=10}
  // P(m) {m==1}
  while (m<=10)
    // T(m) {m==2m#init, m#init<=10}
    // P(m) {m<=10}
     m = 2*m;
  // T(m) {11<=m, m#init<=m,
  //
11<=m#init, m#init<=20}
  // P(m) {11<=m, m<=20}
  while (m>=1)
    // T(m) {m==2m#init, 1<=m#init}
    // P(m) {11<=m}
    m = 2*m;
  // T() empty
  // P() empty
  return;
}
```

```
// T() {}
void divide01() {
  // T(m) {}
  int m;
  // T(m) {2<=m, m<=m#init, 2<=m#init}
  while (m>1) {
    // T(m) {m#init<=2m+1, 2m<=m#init, 2<=m#init}
    // P(m) {2<=m}
    m = m/2;
    // T(m) {1<=m}, P(m) {1<=m}
    printf("m=%d\n",m);
  }
  // T(m) {m+2<=0, m#init<=m, m#init+2<=0}
  // P(m) {m<=1}
  while (m<-1) {
    // T(m) {m#init<=2m+1, 2m<=m#init, m#init+2<=0},
    // P(m) {m+2<=0}
    m = m/2;
    // T(m) {m+1<=0}, P(m) {m+1<=0}
    printf("m=%d\n",m);
  }
  // T() {0<=m+1, m<=1}, P(m) {0<=m+1, m<=1}
  return;
}
```

Figure 8: Beyond counters: multiply (left) & divide (right)

Let us consider the code in Fig. 8 (left). Note that m is found monotonic in T^* in the second loop, thanks to the loop bound, and that the never ending loop is detected (the set of reaching states for `return` is empty), although closed forms are not computed for m. The same kind of results are obtained for a division in Fig. 8 (right). Variable m is found decreasing in the first loop and increasing in the second one, thanks to the loop bounds. Note also that m is not initialized, but that its final value is properly found in $[-1, 1]$.

In [27], Paige and Koenig propose finite differencing as a program optimization method that generalizes strength reduction. If the value of $f(x)$ is known and if $f(x+dx)$ is needed, is it possible to compute $f(x+dx)-f(x)$ faster than $f(x+dx)$? The simple case of strength reduction shows that our approaches are dual. We analyze the piece of code that computes $f(x+dx)-f(x)$ and we infer the function f. Paige and Koenig start with the code to evaluate $f(x)$ and infer the code to evaluate $f(x+dx)-f(x)$. Their technique was developed to optimize SETL code and to deal with functions over sets. The challenge for us would be to extend our technique in a dual way to obtain predicates over arrays such as those found in [18].

Monotonicity analysis [29] has also been used to extend induction variable detection, the inverse transformation of strength reduction. Basically, assignment statements nested in loops are monotonic if the value assigned increases from one iteration to the next. The exact value of the difference is abstracted by its sign. This information does not lead to loop invariants but is useful for dependence testing [30] and for array bound checking. We could derive the same kind of information from T^+ by introducing the difference variables and by eliminating the program variables. The transformer we find for the contrived loop in Figure 7 of [29] is:

```
T(i,j,k,l,m,n,x,y,z) { i==i_0+1, i==j-2, 2i==k-2, i+m#init==l-2, l==m-1,
                       l==n-6, x==2y_0, 4y_0==z, 2<=i, y<=2i+2, i<=3y+1 }
```

The first equation shows that i is increasing, hence j and k from the next two equations. Equations (4) and (5) lead to `m-m#init=i+3`. The loop body precondition includes $i \geq 1$, so m is increasing, thus l and n as well. We also have `i-1<=3y<=6i+6`. The lower and upper bounds are increasing but we cannot derive monotonicity information about y, nor about x and z. As the loop body is summarized, the monotonicity information is linked to the variables and not to the assignments, but this is exactly the same information.

5 Conclusion

A simple algorithm to compute affine invariants over integer scalar variables in while loops is presented. Its development and refinements have been mostly application driven, targeting the automatic program analysis and transformation domain. Its low complexity is key to addressing large scientific codes of up to 100 KLOC. Our experience shows that it performs well on standard program

test cases, but not on complex automata whose states and transitions cannot be rewritten with simple C encodings.

This algorithm is more effective in finding loop invariants when inaccurate operations such as convex hulls and transitive closures are postponed as much as possible. If the analysis were exact, formulae such as $P^* = T^*(P^0)$ and $P^* = P^0 \bigsqcup T^+(P^0)$ would be equivalent. However, the formulae dealing with approximate transformers and preconditions are not and the best one must be chosen or a trade-off be made between accuracy and computational complexity. Developed formulae such as Eq.(9) correspond to peeling and unrolling the while loop in the computation. It is not compatible with Bourdoncle's heuristic, which aims at minimizing the number of widenings: better results are obtained by increasing the number of cycles and of transitive closures in order to delay convex hulls, as the closures are quite accurate.

When the program behavior is not affine, its affine approximations can be refined iteratively using the previous preconditions to obtain more accurate loop body transformers. This does not yield an algorithm as the iterations do not converge when the domain is not bounded (Section 3.4).

Our current implementation in PIPS is not fully satisfying as some extensions described in this paper are not available yet. They are not required often enough to justify the potential average slowdown and implementation time. Using a domain product instead of a unique general abstract domain could be investigated. Another idea would be to combine a widening heuristic and a derivative transitive closure algorithm. The later could be used for abstract acceleration in a widening heuristic along the lines of [10], but a combined approach is still in want of motivating test cases.

Acknowledgments

N. Halbwachs suggested years ago that our transitive closure algorithm should be published, however simple it was. It benefited from observations by B. Creusillet and N. Nguyen who performed tedious experiments and screened the results for missing information. P. Jouvelot and reviewers suggested many improvements in the presentation and in the related work. L. Gonnord helped us understand how her ASPIC tool works. P. Jouvelot formally proved the correctness of the code transformation of Section 3.5, see [1].

References

[1] Corinne Ancourt, Fabien Coelho, and FranÃ§ois Irigoin. A modular static analysis approach to affine loop invariants detection (extended version). Technical Report A-419, CRI, MINES ParisTech, 2010. Parts to be published in NSAD 2010.

[2] W. Bielecki, T. Klimek, and K. Trifunovic. Calculating exact transitive closure for a normalized affine integer tuple relation. *Electronic Notes in*

Discrete Mathematics, 33:7 – 14, 2009. International Conference on Graph Theory and its Applications.

[3] Bernard Boigelot, Louis Bronne, and Stéphane Rassart. Symbolic verification with periodic sets. In *Proc. 9th International Conference on Computer-Aided Verification, volume 1254, Lecture Notes in Computer Science*, pages 167–177. Springer-Verlag, 1997.

[4] Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In *6th International Conference on Computer Aided Verification, number 808 in LNCS*, pages 55–67. Springer-Verlag, 1994.

[5] François Bourdoncle. *Sémantiques des langages d'ordre supérieur et interprétation abstraite*. PhD thesis, École polytechnique, novembre 1992.

[6] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Proc. 9th International Conference on Computer-Aided Verification, volume 1254, Lecture Notes in Computer Science*, pages 400–411. Springer-Verlag, June 1997.

[7] Hubert Comon and Yan Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 268–279, London, UK, 1998. Springer-Verlag.

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

[9] Laurent Fribourg and Hans Olsen. Proving safety properties of infinite state systems by compilation into presburger arithmetic. In *CONCUR'97, LNCS 1243*, pages 213–227. Springer, 1997.

[10] Laure Gonnord. *Acceleration abstraite pour l'amelioration de la precision en analyse des relations lineaires*. PhD thesis, Université Joseph Fourier, Grenoble, France, 2007.

[11] Denis Gopan and Thomas W. Reps. Guided static analysis. In *Static Analysis, 14th International Symposium, SAS 2007*, pages 349–365, 2007.

[12] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT '06*, pages 117–127, 2006.

[13] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI '09*, pages 375–385, 2009.

[14] Sumit Gulwani and Nebojsa Jojic. Program verification as probabilistic inference. *SIGPLAN Not.*, 42(1):277–289, 2007.

[15] Mohammad Haghighat. *Symbolic analysis for parallelizing compilers*. Boston Kluwer Academic, 1996.

[16] N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Université Scientifique et Médicale de Grenoble, March 1979.

[17] Nicolas Halbwachs. Delay analysis in synchronous programs. In *Fifth Conference on Computer Aided Verification*, pages 4–13. Springer Verlag, 1993.

[18] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI '08*, pages 339–348, 06 2008.

[19] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Form. Methods Syst. Des.*, 11(2):157–185, 1997.

[20] FranÃ§ois Irigoin. Interprocedural analyses for programmig environments. In *Environments and Tools for Parallel Scientific Computing*, pages 333–350. Elsevier, September 1993.

[21] FranÃ§ois Irigoin, Pierre Jouvelot, and RÃ©mi Triolet. Semantical interprocedural parallelization: an overview of the pips project. In *ICS*, pages 144–151, June 1991.

[22] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. *Int. J. Parallel Program.*, 24(6):579–598, 1996.

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

[24] MINES-ParisTech. PIPS. `http://pips4u.org`, 1989–2009. Open source, under GPLv3.

[25] Nga Nguyen. *Efficient and Effective Software Verification for Scientific Applications Using Static Analysis and Code Instrumentation*. PhD thesis, École des mines de Paris, 2002.

[26] Thi Viet Nga Nguyen and Francois Irigoin. Efficient and effective array bound checking. *TOPLAS*, 27(3):527–570, May 2005.

[27] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *TOPLAS*, 4(3):402–454, 1982.

[28] Sebastian Pop, Albert Cohen, and Georges-Andre Silber. Induction variable analysis with delayed abstractions. Technical report, Ecole des mines de Paris, CRI A/367, 2005.

[29] Madalene Spezialetti and Rajiv Gupta. Loop monotonic statement. *IEEE-TOSE*, 21(6):497–505, Jun 1995.

[30] Peng Wu, Albert Cohen, Jay Hoeflinger, and David Padua. Monotonic evolution: an alternative to induction variable substitution for dependence analysis. In *ICS'01*, pages 78–91. ACM Press, June 2001.

[31] Chaochen Zhou, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, December 1991.

6 Appendix : Proof for the while loop conversion in Section 3.5

This proof has been provided graciously by P. Jouvelot.

The semantics of while loop W defined as `while (c) do if (t) then a else b;` is the lower fixed point $\mathrm{lfp}(F)$ with Functional F defined as:

$$F = \lambda w.\lambda s.s \text{ if } E[\texttt{c}]s \text{ is false}, w(C[\texttt{if (t) then a else b}]s) \text{ otherwise}$$

Following Kleene's theorem,

$$\mathrm{lfp}(F) = \lim_{i \to \infty} F^i(\bot)$$

If, for a given s at the beginning of the loop, we consider n as the smallest i such that $E[\texttt{c}](F^i(\bot)s)$ is false (and ω if there is no such n in $\mathbb{N}$), then:

$$\mathrm{lfp}(F) = (C[\texttt{if (t) then a else b}])^n$$

By the definition of n and F (which leads to fixed points equivalent to the identity function after n iterations when `c` is false in the current state), we have:

$$\mathrm{lfp}(F) = (C[\texttt{if (c and t) then a; if (c and not t) then b}])^\omega$$

By grouping the semantics linked to successive instances of `a` and `b`, we get:

$$\mathrm{lfp}(F) = (C[W'])^\omega$$

where $W' =$ `while (c and t) do a; while (c and not(t)) do b;`

Following the definition of n, n is an upper bound of the number of iterations necessary for W' to make $(C[W'])^\omega$ converge, so:

$$\mathrm{lfp}(F) = (C[W'])^n$$

And this $\mathrm{lfp}(F)$ is also the smaller fixed point of F' defined as:

$$F' = \lambda w.\lambda s.s \text{ if } E[\texttt{c}]s \text{ is false}, w(C[W']s) \text{ otherwise}$$

Thus W is semantically equivalent to `while (c) do W'` *i.e.* W

```
while (c) do {
  while (c and t) do a;
  while (c and not(t)) do b;
}
```

7 Examples

This section contains the source codes of all examples used above. Preconditions and transformers are available on PIPS SVN server (see `http://pips4u.org`) and can be downloaded with the validation suite.

The source codes are:

- Car Safety (see Figure 9).
- Parametric Car Safety: the maximal duration is a parameter n (see Figure 10).
- Restructured Car Safety: car safety code after restructuration (see Figure 11).
- Original Flip Flop presented in Section 3.2 (see Figure 12)
- Flip Flop - version 1 in Figure4, Section 3.2 (see Figure 13)
- Flip Flop - version 2 in Figure4, Section 3.2 (see Figure 14)
- Flip Flop - version 3 in Figure4, Section 3.2 (see Figure 15)
- Flip Flop - version 4 in Figure4, Section 3.2 (see Figure 16)
- Flip Flop - version 5 in Figure4, Section 3.2 (see Figure 17)
- Flip Flop - version 6 in Figure4, Section 3.2 (see Figure 18)
- Gopan'07 (see Figure 19)
- Restructured Gopan'07: Gopan'07 code after restructuration (see Figure 20)
- Gulwani'07 (see Figure 21)
- Restructured Gulwani'07: Gulwani'07 code after restructuration (see Figure 22)
- Gulwani'06 (see Figure 23)
- Halbwachs'79 - code 1 (see Figure 24)
- Halbwachs'79 - code 2 (see Figure 25)
- Second Order (see Figure 26)
- Parabolic (see Figure 27)
- Gaz Burner (see Figure 28)
- Subway (see Figure 29)
- Divide (see Figure 30)
- Multiply (see Figure 31)

```
#include <stdlib.h>
#include <stdio.h>

int alea(void)
{
  float fr = ((float) rand())/((float)RAND_MAX);
  return ((fr>0.5)?1:0);
}
int main()
{
  int s = 0, t = 0, d = 0;

  while(s <= 2 && t <= 3) {
    if(alea())
    t++, s = 0;
    else
    d++, s++;
  }
  if(d <= 10)
  printf("healthy");
  else
  printf("crashed!");
}
```

Figure 9: Example : Car Safety

```
// car-safety + Symbolic bound for t
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

int alea(void)
{
  float fr = ((float) rand())/((float)RAND_MAX);
  return ((fr>0.5)?1:0);
}

void main(int n)
{
  int s = 0, t = 0, d = 0;
  assert(n>=0);
  while(s <= 2 && t <= n) {
    if(alea())
    t++, s = 0;
    else
    d++, s++;
  }
  if(d <= 2*n+3)
  printf("healthy");
  else
  printf("crashed!");
}
```

Figure 10: Example : Parametric Car Safety

```
// car-safety + Symbolic bound for t
// after restructuration
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

int alea(void)
{
  float fr = ((float) rand())/((float)RAND_MAX);
  return ((fr>0.5)?1:0);
}

void main(int n)
{
  int s = 0, t = 0, d = 0;
  assert(n>=0);
  while(s <= 2 && t <= n) {
    while(s <= 2 && t <= n && alea() > 0.)
      t++, s = 0;
    while(s <= 2 && t <= n && alea()<= 0.)
      d++, s++;
  }

  if(d <= 2*n+3)
    printf("healthy");
  else
    printf("crashed!");
}
```

Figure 11: Example : Parametric Car Safety after restructuration

```
//  Check analysis of periodic behaviors
//
double g(double x)
{
  return x;
}

int main()
{
  double x[2][10];
  int old = 0, new = 1, i, t;

  t=0;
  while(t<1000) {
    for(i=0;i<10;i++)
      x[new][i] = g(x[old][i]);
    old = new;
    new = 1 - old;
    t++;
  }
}
```

Figure 12: Example : Flip-flop

```
//  Check analysis of periodic behaviors
//

double g(double x)
{
  return x;
}

int main()
{
  double x[2][10];
  int old = 0, new = 1, i, t;

  t=0;
  while(t<1000) {
    for(i=0;i<10;i++)
      x[new][i] = g(x[old][i]);
    new = old;
    old = 1 - old;
    t++;
  }
}
```

Figure 13: Example : Flip-flop 1

```
//  Check analysis of periodic behaviors
//

double g(double x)
{
  return x;
}

int main()
{
  double x[2][10];
  int old = 0, new = 1, i, t;

  t=0;
  while(t<1000) {
    for(i=0;i<10;i++)
      x[new][i] = g(x[old][i]);
    new = 1 - new;
    old = 1 - old;
    t++;
  }
}
```

Figure 14: Example : Flip-flop 2

```
//  Check analysis of periodic behaviors
//

double g(double x)
{
  return x;
}

int main()
{
  double x[2][10];
  int old = 0, new = 1, i, t, temp;

  t=0;
  while(t<1000) {
    for(i=0;i<10;i++)
      x[new][i] = g(x[old][i]);
    temp = new;
    new = old;
    old = temp;
    t++;
  }
}
```

Figure 15: Example : Flip-flop 3

```
//  Check analysis of periodic behaviors
//

double g(double x)
{
  return x;
}

int main()
{
  double x[2][10];
  int old = 0, new = 1, i, t;

  t=0;
  while(t<1000) {
    for(i=0;i<10;i++)
      x[new][i] = g(x[old][i]);
    if (new==1)
      new=0,old=1;
    else
      new=1,old=0;
    t++;
  }
}
```

Figure 16: Example : Flip-flop 4

```
//  Check analysis of periodic behaviors
//

double g(double x)
{
  return x;
}

int main()
{
  double x[2][10];
  int old = 0, new = 1, i, t;

  t=0;
  while(t<1000) {
    for(i=0;i<10;i++)
      x[new][i] = g(x[old][i]);
    if (new==1)
      new=0,old=1;
    else
      if (new==0)
        new=1,old=0;
      else
        exit(1);
    t++;
  }
}
```

Figure 17: Example : Flip-flop 5

```
//  Check analysis of periodic behaviors
//

double g(double x)
{
  return x;
}

int main()
{
  double x[2][10];
  int old = 0, new = 1, i, t;

  t=0;
  while(t<1000) {
    for(i=0;i<10;i++)
      x[new][i] = g(x[old][i]);
    new = (new+1)%2;
    old = (old+1)%2;
    t++;
  }
}
```

Figure 18: Example : Flip-flop 6

```
// From Gopan 2006
//
#include <stdio.h>

int main()
{
  int x,y;
  x=y=0;

  while(true) {
    if (x<=50)
      y++;
    else y--;
    if (y<0) break;
    x++;
  }
  if(x==102)
    printf("property␣␣verified\n");
}
```

Figure 19: Example: Gopan'07

```
// Test from Gopan - SAS07
// Cited by Gulwani in Control-flow refinement and
Progress invariants for Bound
// Analysis - PLDI'09
//

#include <stdio.h>

int main()
{
  int x,y,z,k;
  x=0;
  y=0;

  while (y>=0)
    {
     while(y>=0 && x<=50)
        {
          x++; y++;
        }
      while (y>=0 && x>50)
        {
          y--;
          x++;
        }
    }
  if(x==103)
    printf("property␣␣verified\n");
}
```

Figure 20: Example: Gopan'07 after restructuration

```
//   Test from Gulwani 2007
//
#include <stdio.h>

int main()
{
  int x,y,z;
  x=0;
  y=50;

  while(x<100) {
    if (x<50)
      x++;
    else {
      x++; y++;
    }
  }
  if (y==100) printf("property␣␣verified\n");
}
```

Figure 21: Example: Gulwani'07

```
//  From Gulwani 2007
// Cited in Control-flow refinement and Progress invariants for Bound
// Analysis - PLDI'09
#include <stdio.h>
int main()
{
  int x,y,z;
  x=0;
  y=50;

  while(x<100)
    {
      while ( x<50)
        x++;
      while (x<100 && x>=50){
        x++; y++;
      }
    }
  if (x ==100 && y==100) printf("property␣␣verified\n");
}
```

Figure 22: Example: Gulwani'07 After restructuration

```
// From Gulwani 2006
// Cited in Control-flow refinement and Progress invariants for Bound
// Analysis - PLDI'09

#include <stdlib.h>
#include <stdio.h>

int alea(void)
{
  float fr = ((float) rand())/((float)RAND_MAX);
  return ((fr>0.5)?1:0);
}
int main()
{
  float z;
  int x,y,lock;

  x=1;
  lock = 0;
  y=0;

  while(x!=y)
    {
      lock =1; x=y;
      if (alea()) {
        lock =0; y++;
      }
    }
  if (lock ==1) printf("property␣␣verified\n");
}
```

Figure 23: Example: Gulwani'06

```
#include <stdio.h>
#include <stdlib.h>

int alea(void)
{
  float fr = ((float) rand())/((float)RAND_MAX);
  return ((fr>0.5)?1:0);
}

int main()
{
  float z;
  int x,y;
  x=y=0;

  while(x<=100)
    {
      if (alea())
        x = x+2;
      else {
        x++;y++;
      }
      if (x+y<=202) printf("property␣verified\n");
    }
}
```

Figure 24: Example 1 : Halbwachs'79

```
#include <stdio.h>
#include <stdlib.h>

int alea(void)
{
  float fr = ((float) rand())/((float)RAND_MAX);
  return ((fr>0.5)?1:0);
}

int main()
{
  float z;
  int x,y;

  x=y=0;

  while(x<=100)
    {
      if (alea())
        x = x+4;
      else {
        x=x+2;
        y++;
      }
      if (x+2*y<=204) printf("property␣verified\n");
    }
}
```

Figure 25: Example 2 : Halbwachs'79

```
// Example in Section 3.4
#include <stdio.h>

int main()
{
  int i = 0, j = 2, k = 1;
  while(k<=10) {
    j--;
    i += j;
    k++;
  }
  printf("i=%d,␣j=%d,␣k=%d\n", i, j, k);
}
```

Figure 26: Example: Non Linear

```
// Example in Section 3.4

#include <stdio.h>

int main()
{
  int i = 0, j = 0, n;
  if(n<0) exit(1);
  while(i<=n) {
    i++;
    j+=i;
  }
  printf("i=%d,␣j=%d\n", i, j);
}
```

Figure 27: Example: Parabolic

```
#include <stdio.h>
#include <stdlib.h>

int alea(void)
{
  float fr = ((float) rand())/((float)RAND_MAX);
  return ((fr>0.5)?1:0);
}
int main()
{
  float z;
  int x,l,t;

  t=l=x=0;
  while(1)
    {
      x=0;
      while (x<=9 && alea())
        {
          x++; t++;l++;
        }
      x=0;
      while(x<=49 || alea())
        {
          x++;t++;
        }
      if (6*l<= t+5*x) printf("property␣verified\n");
    }
}
```

Figure 28: Example: Gaz Burner

```
// Subway example
// The train detects beacons that are placed along the trasck,
// and receives the "second" from a central clock.
// The train adjusts its speed as:
//  - B-S >= 10 it is early and puts brake as long as B>S
//            it must stop before encountering 10 beacons
//  - B-S <= 10 it is late and is
//            considered late as long B <S.
#include <stdio.h>
#include <stdlib.h>
float alea(void)
{
  float fr = ((float) rand())/((float)RAND_MAX);
  return (fr);
}

int main()
{
  int s=0; // the number of seconds
  int b=0; // the number of beacons
  int d=0; // the number of beacons after
           // the train begins to brake

    // The train is ON TIME
 ontime:
  if (alea()>0.) b++;
  if (alea()>0.) s++;
  if (s-b>=10) goto late;
  if(s-b<=-10) goto early;
  goto ontime;

  // the train is LATE
 late:
  if (alea()>0.) b++;
  if (s==b) goto ontime;
  goto late;

  // the train is early
 early:
  d=0;

 brake:
  // the train puts brake while(s!=b) and d<=10
  if (alea()>0.) {b++;d++;}
  if (alea()>0.) s++;
  if (s==b) goto ontime;
  if (d>=10) goto stopped;
  goto brake;

  // It stops and waits after 10 beacons
 stopped:
  if (alea()>0.) s++;
  if (s==b) goto ontime;
  goto stopped;
}
```

Figure 29: Example: Subway

```
/* Check that m = m/2; leads to dm<=-1 when m>= 1 as 2*dm ~ m and hence
   dm <= 1.
 */
void divide01()
{
  int m ;
  while(m>1) {
    m = m/2;
    m = m; // to get postcondition
  }
  while(m<-1) {
    m = m/2;
    m = m; // to get postcondition
  }
  return;
}
```

Figure 30: Example: Divide

```
void multiply01()
{
  int m = 1;
  while(m<=10)
    m = 2*m;
  while(m>=1)
    m = 2*m;
  return;
}
```

Figure 31: Example: Multiply