Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 267 (2010) 3–13

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Accelerated Invariant Generation for C Programs with Aspic and C2fsm

Paul Feautrier[1]

LIP
University of Lyon
Lyon, France

Laure Gonnord[2]

LIFL
University of Lille
Villeneuve d'Ascq, France

Abstract

In this paper, we present ASPIC, an automatic polyhedral invariant generation tool for flowcharts programs. ASPIC implements an improved Linear Relation Analysis on numeric counter automata. The "accelerated" method improves precision by computing locally a precise overapproximation of a loop without using the widening operator. `c2fsm` is a C preprocessor that generates automata in the format required by ASPIC. The experimental results show the performance and precision of the tools.

Keywords: Abstract interpretation, polyhedral abstract domain, acceleration, fixpoint iteration, flowchat programs, compilation, tools.

1 Introduction

In this paper, we describe the design and implementation of ASPIC, a tool which constructs invariants for an affine interpreted automaton, and presents the heuristics we chose to implement in order to improve the precision of the generated invariants (via the "abstract acceleration" technique). We also describe our experience with applying ASPIC on various examples. The experimental results show the relevance of the method and show that ASPIC is a practical tool for software verification. We also present the `c2fsm` preprocessor, which eases the use of ASPIC by automating the construction of the input automaton from a standard C program.

[1] Email: `paul.feautrier@ens-lyon.fr`

[2] Email: `laure.gonnord@lifl.fr`

doi:10.1016/j.entcs.2010.09.014

2 Aspic

Aspic takes as input a textual representation of a numerical interpreted automaton, and eventually a proof goal. The output of Aspic is a mapping from all control points to affine numerical invariants, and, if required, a diagnostic w.r.t. the proof goal. Aspic provides a collection of options to improve the precision of the analysis, and also a graphical editor.

2.1 Quick Tour

2.1.1. ***Input Language*** Aspic takes as input a variant of the textual automata input format of the tool Fast ([3]), which is composed of (Figure 1):

- A "model", which contains a textual description of a *unique* counter automaton: numerical variables, control points and transition functions consisting of a source, a destination, a boolean affine guard (possibly non convex) and an affine action over the numerical variables.
- A "strategy", which defines "regions", and computation objectives. In contrast with the tool Fast itself, our tool only needs an initial region; an "error region" is optional, and no additional information is required.

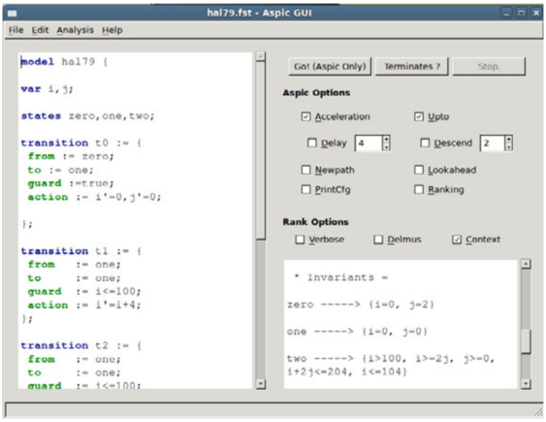
The Aspic input language grammar can be found in the Research Report [11]. In particular, the Aspic language enables the use of a non deterministic operation $x' :=?$, whose semantics is the loss of any information concerning the variable x. The expressivity of the Fast language is thus improved (all affine relations can be encoded).

2.1.2. *Aspic Interface and options*

```
model hal79 {
var i,j;
states zero,one,two;

transition t0 := {
 from := zero;
 to := one;
 guard :=true;
 action := i'=0,j'=0;
};

transition t1 := {
 from   := one;
 to     := one;
 guard  := i<=100;
 action := i'=i+4;
};
...
```



```
model hal79 {

var i,j;

states zero,one,two;

transition t0 := {
 from := zero;
 to := one;
 guard :=true;
 action := i'=0,j'=0;

};

transition t1 := {
 from   := one;
 to     := one;
 guard  := i<=100;
 action := i'=i+4;
};

transition t2 := {
 from   := one;
 to     := one;
 guard  := i<=100;
```

Aspic Options

☑ Acceleration ☑ Upto

☐ Delay 4 ☐ Descend 2

☐ Newpath ☐ Lookahead

☐ PrintCfg ☐ Ranking

Rank Options

☐ Verbose ☐ Delmus ☑ Context

```
 * Invariants =

zero -----> {i=0, j=2}

one -----> {i=0, j=0}

two -----> {i>100, i>=2j, j>=0,
i+2j<=204, i<=104}
```

Fig. 1. Aspic Format — Aspic GUI

The Aspic distribution[3] provides a GUI written in QT (Figure 1), which provides syntax coloring and a frontend (and help) to the main Aspic options. The user writes the automaton to analyse in the left window, choses the options and

[3] http://laure.gonnord.org/pro/aspic

then launch the analysis. The results (invariants associated to each control point) are then printed in the right window.

Aspic provides many analysis options:

- standard analysis (no option): linear relation analysis, with standard widening [8], and accelerations (see Section 2.2) when possible.
- `-noaccel`: no acceleration is performed. The widening is now constrained with a set of "upto" contraints [14], except if the additional option `-nouptos` is added.
- `-reps` performs the "lookahead widening" algorithm described in [12].
- `-delay k` delays the first application of the widening to step k.
- `-descend d` performs a descending sequence of maximum length d after convergence of the forward analysis.

The results can be printed in various formats, including a Dot output for the visualisation of both automaton and invariants.

2.1.3. *Connection to other tools*

The Aspic tool has options to translate Fast files into:

- the input format of the StInG tool, which implements the invariant generation algorithm described in [22].
- the input format of the Rank tool, which implements the algorithm described in [2,1] for proving termination of programs. The invariant generation is performed before any call to the Rank tool.

The Aspic tool is also connected to C and Lustre ([15]) programs :

- `c2fsm` translates a (quite large) subset of C to Fast programs. The main implementation issues of `c2fsm` are described in Section 3.
- oc2fst translates an Oc file (coming from the compilation of Lustre programs), to a Fast file. For the moment oc2fst is not included in the Aspic distribution.

The main difficulties for these tools is to provide an abstraction for non numeric operations which must be as precise as possible, while staying safe (overapproximations). These overapproximations will be discussed in Section 3.

2.2 Implementation Issues

Aspic performs a Linear Relation Analysis, which is improved thanks to the concept of *abstract acceleration*, which was introduced in [10,11], and which basically consists in computing more precise "accelerated" postfixpoints (without widening) when possible.

The Aspic tool makes a forward accessibility analysis. If an error region is defined (a formula over numerical variables and control points), the goal is transformed into a non accessibility problem by creating new bad states and new transitions; if, after convergence, all the bad states are associated to an empty polyhedron, the goal is proved, otherwise the result is inconclusive.

Strongly connected subcomponents are processed individually, according to the strategy of [5]. The decomposition is precomputed at the beginning of the analysis by a variant of the Tarjan algorithm [24]. Some precomputations are made at the beginning in order to apply the acceleration results. Some changes are also made on the topology of the automaton: for instance, some nodes are split. The iteration is a classical fixpoint iteration, except when some loops are accelerable: in this case, a (post) fixpoint is computed locally thanks to the acceleration results of [11].

2.2.1. *Detecting and preprocessing accelerable loops.*

During the first phase of the analysis the transition functions are preprocessed, an internal structure encodes the type of the action (identity, translation, translation reset, idempotent transition, . . .), of the guard (always true, simple, complex, . . .), whether the transition is accelerable, and other useful informations that can be precomputed (postconditions, rays to add, . . .).

The control structure of the automaton is modified in order to deal with accelerable loops:

- **The unique single loop case** (a unique circuit around the head of the strongly connected subcomponent) is dealt with as follows: if the loop is accelerable, then the control point is split into two points, linked by a *meta-transition*, as shown by Fig. 2. This splitting for single loops aims at suppressing the widening at control point q. At q_{split}, we compute the (abstract) effect of the acceleration on the polyhedron associated to q.

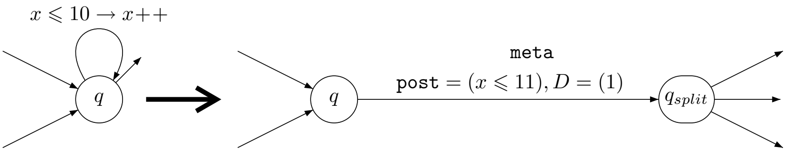

$x \leqslant 10 \to x{+}{+}$
q
q
meta
post $= (x \leqslant 11), D = (1)$
q_{split}

Fig. 2. The unique single loop case

- **The multiple single loop case**. For multiple single loops, we also decide to split the control point, as shown in Fig. 3. Multiple loops can be dealt with in two ways:

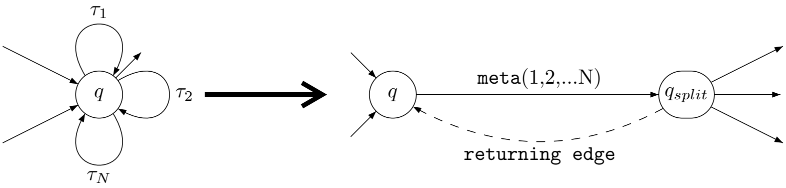


Fig. 3. The multiple single loops case

  - · If we have only partial acceleration results, we introduce a return identity edge, which creates a new loop, so a widening node must be chosen among q and q_{split}.
  - · If complete acceleration results are available, which means that the multiple loops can be accelerated all together, this return arc is not necessary. This case is similar to the single loop one.

- **The complex loop case** (loops are circuits) We deal with this case by (possibly) precomputing the meta transitions associated to the circuits that are detected by the Tarjan algorithm (deep first search). We compute the associated transformation backward, by composing actions and computing preconditions of guards. For instance, let us consider the following circuit: $(q, \tau_1, q_1)(q_1, \tau_2, q)$ with $q_1 : x \leqslant 7 \rightarrow x := x + 1$ and $q_2 : x \leqslant 4 \rightarrow x := x + 3$. In this case, we compute the transition $q_2 \circ q_1 : x \leqslant 3 \rightarrow x := x + 5$, and it is accelerable, hence we add a meta-transition over the control point q. Both initial transitions are kept in order to preserve the semantic of the CFG. The main drawback of this approach is that not every circuit is detected, in particular in the case of two circuits sharing the same entry point. We chose to avoid the detection of all circuits, and to focus on the loops detected by the DFS.

2.2.2. *The choice of widening nodes:* Since the first phase modifies the graph structure, the computation of widening control points is done afterwards. Bourdoncle's strategy [5] has been modified as follows: if the head q of a strongly connected subcomponent has been split (with the creation of q_{split}), then q_{split} is chosen as a widening point, instead of q. The reason is that it is better to widen at a control point where the most precise information has been collected. Experiments show that widening after acceleration is a good heuristic.

2.2.3. *Fixpoint iterations* The fixpoint iteration is quite classical : the inner loops are processed before the outer ones. The applications of the transition functions are basically the same as in classical LRA, except for meta transitions, where we apply the algorithms described in [11] to compute the image of a given polyhedron by a meta transition.

2.2.4. *Back to the initial automaton* At the end of the fixpoint iteration on the modified automaton, the final results are computed w.r.t. the initial control points, by taking the union (convex hull) of the invariants associated to the two control points obtained after spliting.

2.3 Implementation

Aspic is implemented over a fixpoint generic analyzer called Analyseur[4]. This tool performs a fixpoint analysis, given an encoding of the control flow graph and an implementation of the abstract lattice of properties. We chose the polyhedral library NewPolka[5], which has an Ocaml interface. These two librairies are now embedded in the Apron Interface ([17]. The cumulated number of lines of Ocaml code is 20000 (without NewPolka).

2.4 Future extensions

We are currently extending the Aspic tool in two directions :

[4] http://pop-art.inrialpes.fr/people/bjeannet/analyzer/index.html

[5] http://pop-art.inrialpes.fr/people/bjeannet/newpolka/index.html

- The connection to synchronous programs though OC2FST is being improved and will be included in a next release.
- The acceleration of complex loops will be enhanced by a better choice of the circuits which will be accelerated. This choice will be done *at each iteration step* by choosing a *relevant* circuit by means of SMT requests.

Future work also includes interprocedural analysis via the strategy described in [14]: all procedures are considered as (affine) relations between outputs and input. The invariants of each inner procedure are computed first, and these invariants are used to compute the effect of a given procedure call.

3 C2FSM

3.1 Overview

Constructing an interpreted automaton from a C program is no different, in principle, from a control flow graph construction by a compiler. The elementary statements of the program (mainly assignments), become states of the automaton. Transitions encode the flow of control, including sequential execution, conditionals loops and even GOTOs. The predicate of tests and loops become guards on transitions. When an assignment meets the constraints of the interpreted automaton paradigm, it translates into an action which is affixed to the outgoing transition of the state. The main difficulty here is how to approximate non-affine expression and guards.

3.2 Input Language

`c2fsm` accepts programs in a slightly out-of-date dialect of C. It does not parse recent extensions like booleans and inner functions. Other constructs, like bit fields, initializations in declarations, bitwise operators, enums, `sizeof`, pointers, structures and unions, are parsed but are handled as untractable constructs by the generator. All the C control constructs with the exception of `switch` (`if, while, for, do, goto`) are implemented.

We have plans to extend this input language, probably by using the `gcc` front end as a parser.

3.3 Interfaces and Options

`c2fsm` is run from the command line:

```
c2fsm <file>.c <options>
```

Options control the nature of the output (`-fst` for the FAST format, and `-dot` for the DOT format, suitable for drawing the resulting automaton). Other options (`-s` and `-cut`) control the degree to which the resulting automaton is simplified (see Sect. 3.4.4).

3.4 Implementation Issues

3.4.1. ***The Parser*** The parser for `c2fsm` has been written in Ocaml using the `ocamlyacc` implementation of Yacc and the C grammar in [23]. The result is an XML representation of the abstract syntax tree. Error detection is minimal, and diagnostics are rudimentary. The user is advised to use a standard C compiler as a filter before attempting to use `c2fsm`.

3.4.2. ***Construction of the Raw Automaton***

`c2fsm` creates a state per statement in the source. The name of the state is the label of the statement. The parsing tool create conventional labels for unlabeled statements; however, user assigned labels are an unvaluable help for understanding the results of `c2fsm` and subsequent tools.

One difficulty comes from the fact that the C assignment symbol is an operator which return a value, thus allowing such conundrums as `if((c = f(x = y+z)) > x)`. In this case, the tool applies a process of unwinding, which may generate more than one state per statement. The semantics of C does not specify in which order multiple assignments may be executed. `c2fsm` applies an innermost leftmost policy.

`c2fsm` then proceeds to the expansion of the control statements (`while, for, if, do, goto`) using the familiar definitions, and adding new states as needed. In the interest of simplicity, no attempt is made to minimize the automaton at this stage. For instance, if a test has a `then` but no `else`, the tool generates a blank transition for the `else` branch.

3.4.3. ***Actions and Guards*** `c2fsm` collect all integer scalars to become the variables of the resulting automaton. Assignments become actions. However, if the right hand side of a scalar assigment is not affine (e.g., if it accesses an array or call a function), the resulting action assigns the unknown value `'?'` to the scalar. Assignments to non integer or non scalar variables are ignored, and generate other blank transitions in the automaton.

Similar rules are applied to the conditions of tests and loops, with the following improvement. Consider a test whose condition $b \wedge f()$ is the conjunction of a boolean affine formula and of something untractable, such as a call to a random number generator. To make a transition to the `then` branch, b must be true. Hence, this transition is guarded by b. On the other hand, a transition to the `else` branch is possible whatever the value of b; hence, this transition has a true guard. Dual rules are applied to disjunctions.

3.4.4. ***Simplifications*** The heart of all operations on automata is *path coalescing*. Consider two consecutive transitions, with actions and guards (g_1, a_1) and (g_2, a_2). They can be replaced by one transition with guard $g_1 \wedge g_2 \circ a_1$ and action $a_2 \circ a_1$. `c2fsm` implements a quite sophisticated algebraic and logic calculator, which is able to simplifies such expressions, and detect, for instance, cases where the resulting guard is unsatisfiable.

This facility can be used in two ways: with the option `-s`, it is applied only if the intermediate state has unit fan-in and fan-out. It can then be removed after coalescing. With the option `-cut`, the tool first identifies a set of cutpoints (the source of each backedge in the automaton graph, plus the start and stop nodes). The tool then proceeds to eliminate all other nodes by path coalescing. The resulting automaton has usually much less states than the original. However, eliminating a state with fan-in m and fan-out n generates at most $m \times n$ transitions, baring simplifications as above. There is clearly a tradeoff here: the position of the optimum probably depends on the subsequent use of the automaton.

3.4.5. *Assume and Assert* In many situations, an analysis is to be conducted under *preconditions* on the initial values of the variables, and is used to prove *post-conditions* to be verified at some point in the computation. Two special constructs are recognized by `c2fsm`:

```
assume(<condition>);
```

The boolean condition, which must be affine in the integer variables of the program, is added to the definition of Aspic initial region.

```
assert(<condition>, <string>);
```

This construct is transformed into a conditional `goto` to an error state whose name is the string argument. If the condition is not met, a transition to the error state is executed. There are two ways of verifying (or not) the assertion. Firstly, when simplifying the automaton, it may happen that all paths to the error state have a false guard. In that case, the error state disappears. In the other case, Aspic eventually finds that the error state is unreachable.

4 Experiments

In this section we present some experiments driven with the Aspic tool. These results show that the method we have proposed gives interesting results in terms of precision and effectiveness. All these examples (and other ones) can be found in the Aspic webpage.

4.1 Finding invariants

Table 1 shows a brief comparison between some other methods. The first column shows the results with classical LRA with uptos, the second column shows the results obtained with the StInG tool ([22]), the third colum says whether or not the tool FastER ([20,3]) is able to compute the exact invariant (the Fast version we used does not give the fixpoint), and the last column shows the invariant obtained with our acceleration technique.

No computation time is given because all these analyzes are instantaneous, with the exceptions of the gas burner and the car analysis with the Fast tool (we stopped these two analysis after 15 min because the Presburger automata were too big at this time (more than 8000 states in each case).

Name	Classical LRA (uptos)	STING	Fast	Accelerated LRA
$Hal79a$	$\left\{\begin{array}{c} 0 \leqslant j \\ 2j \leqslant i \leqslant 104 \end{array}\right\}$	$\left\{\begin{array}{c} 0 \leqslant j \\ 2j \leqslant i \end{array}\right\}$	OK	$\left\{\begin{array}{c} \boldsymbol{i + 2j \leqslant 204} \\ i \leqslant 104, 0 \leqslant j \end{array}\right\}$
$Hal79b$	$\{0 \leqslant y \leqslant x \leqslant 102\}$	$\left\{ 0 \leqslant y \leqslant x \right\}$	OK	$\left\{\begin{array}{c} 0 \leqslant y \leqslant x \leqslant 102 \\ \boldsymbol{x + y \leqslant 202} \end{array}\right\}$
$Train1$	$\left\{\begin{array}{c} d = 9 \\ 20 \leqslant b \\ b \leqslant s + 20 \end{array}\right\}$	$\left\{\begin{array}{c} \ldots \\ \boldsymbol{11} \leqslant b \\ 1 \leqslant b - s \leqslant 20 \end{array}\right\}$	OK	$\left\{\begin{array}{c} d = 9 \\ 20 \leqslant b \\ b \leqslant s + 20 \end{array}\right\}$
(GB) $GazBurner$	$\{0 \leqslant x \leqslant \ell \leqslant t\}$	$\{0 \leqslant x \leqslant \ell \leqslant t\}$	$> 15min$	$\left\{\begin{array}{c} \boldsymbol{6\ell \leqslant t + 5x} \\ 0 \leqslant x \leqslant 10 \\ x \leqslant \ell, 0 \leqslant \ell \end{array}\right\}$
$SimpleCar$	$\left\{\begin{array}{c} 0 \leqslant s \leqslant 4 \\ s \leqslant \boldsymbol{d \leqslant 4t + s} \end{array}\right\}$	$\left\{\begin{array}{c} 0 \leqslant s \leqslant d \\ 0 \leqslant t \end{array}\right\}$	$> 15mn$	$\left\{\begin{array}{c} 0 \leqslant s \leqslant 4 \\ s \leqslant \boldsymbol{d \leqslant 4t + s} \end{array}\right\}$

Table 1
Invariants for simple numerical automata

4.2 Proving Properties - toy examples

Table 2 shows a comparison between different variants of linear relation analysis: classical LRA (first column), LRA with lookahead widening (second column), and accelerated LRA (third column), while proving properties. These results show that less iteration steps are required to prove the same proof goal when acceleration is used.

Example	Proof Goal	Classical LRA	Lookahead	Accelerated LRA
$swap$	$da \leqslant db + 1$	$delay = 4/6it$	$delay = 4/7it$	$delay = 1/1it$
$subway$	$b \leqslant s \Rightarrow s - b \leqslant 29$	$delay = 1/5it$	$delay = 20/23it$	$delay = 1/4it$
$gazburner$	$6\ell \leqslant t + 50$	$delay = 63/65it$	$delay = 63/66it$	$delay = 1/5it$
$wcet1$	$3k \leqslant 10i + 10$	$delay = 11/12it$	$delay = 10/12it$	$delay = 1/4it$
$wcet2$	$20 \leqslant k1$	$delay = 37/39it$	$delay = 37/40it$	$delay = 5/8it$

Table 2
Toy examples with proof goals

4.3 Proving properties from C programs

File	Time (c2fsm+aspic)	Proved
Apache(`simp1_ok`)	0.5+0.1	No buffer Overflow (c2fsm)
Sendmail(`inner_ok`)	0.4 + 0.1	No buffer Overflow (c2fsm)
Sendmail(`mime_fromqp_arr_ok.c`)	1.4 + 0.1	No buffer Overflow (aspic)
Spam(`loop_ok`)	1+0.1	No Buffer Overflow (aspic)
OpenSER(`parse_config_ok`)	1.2+0.1	No Buffer Overflow (aspic+accel)
Heapsort(`realheapsort`)	2 +0.8	Termination (aspic)
Loops (`nestedLoop`)	0.8+0.1	Termination (aspic+delay4+accel)
`list.c`	1+0.1	AssertOK (aspic+delay4+accel)
`disj_simple.c`	0.5+0.1	AssertOK (aspic+accel)

Table 3
Benchmarks inspired by [18] and [13]

These results show that the performances of `c2fsm` and Aspic are promising (for instance, the computation is much better than InvGen for `heapsort`, and comparable for the rest of the benchmarks). `c2fsm` performs an abstraction precise enough to prove the desired properties. For termination, we used the technique described in [2], which uses Aspic invariants as input to compute affine ranking functions.

5 Related work

There are many other tools for the computation of numerical invariants. Here is a non exhautive list of some recent ones:

- Nbac[6] implements the classical LRA in combination to dynamic partitioning ([16]). Contrary to Aspic, the tool is dedicated to the verification of properties of Lustre programs. The method perfoms forward and backward analysis from a minimal control structure, and the CFG is partionned w.r.t. the analysis results (and the proof goal). Our technique can be used to improve the precision of invariants during each forward/backward analysis.
- Lash[7] and FastER[8] use acceleration techniques to compute, when possible, the exact reachability sets of counter automata. Theoretical results concerning the acceleration of some subclasses of loop have been obtained this last ten years (for difference bound contraints [7], a subclass of affine guarded functions [4,9] and more recently for octagonal relations [6]). However, the tools based on these algorithms are not fully automatic (Lash), or are not guaranteed to terminate (FastER), in particular for nested loops.
- StInG[9], and InvGen[10] use a combination of LRA and Farkas lemma to discover numerical invariants. The main drawback of the method is the use of template invariants, which prevents the analysis to discover any invariant which is not of the right form. To improve the precision, InvGen perfoms an execution of the program to add some additional constraints, which increase the global analysis time.

There are many C parsers and experimental compilers, including CIL [21], LLVM [19] and Suif [25], and all of them could be used as a replacement for the `c2fsm` parser. However, none of them is able to extract an automaton from their intermediate representation, let alone do the complex approximations and transformations that are necessary prior to static analysis. In fact, since these tools are geared toward compilation, they tend to represent their input program as faithfully as possible.

References

[1] C Alias, A Darte, P Feautrier, and L Gonnord. Bounding the computational complexity of flowchart programs with multi-dimensional rankings. Research Report 7235, INRIA, March 2010.

[6] http://pop-art.inrialpes.fr/~bjeannet/nbac/index.html

[7] http://www.montefiore.ulg.ac.be/~boigelot/research/lash

[8] http://www.lsv.ens-cachan.fr/fast/

[9] http://theory.stanford.edu/~srirams/Software/sting.html

[10] http://www.model.in.tum.de/~guptaa/invgen/

[2] C Alias, A Darte, P Feautrier, and L Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *To appear in International Symposium on Static Analysis (SAS)*, 2010.

[3] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: Fast acceleration of symbolic transition systems. In *CAV'03*, pages 118–121, Boulder (Colorado), July 2003. Springer-Verlag.

[4] B. Boigelot. Symbolic methods for exploring infinite state spaces. Phd thesis, Université de Liège, 1999.

[5] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *In Proceedings of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141, 1993.

[6] M Bozga, C Girlea, and R Iosif. Iterating Octagons. In Anna Kowalewski, Stefan; Philippou, editor, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, March 2009.

[7] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In *CAV'98*, Vancouver (B.C.), 1998. LNCS 1427, Springer Verlag.

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages, POPL'78*, Tucson (Arizona), January 1978.

[9] A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Foundations of Software Technology and Theoritical Computer Science (FSTTCS'2002)*, pages 145–156, Kanpur, India, December 2002. Springer.

[10] L Gonnord and N Halbwachs. Combining widening and acceleration in linear relation analysis. In *13th International Static Analysis Symposium, SAS'06*, Seoul, Korea, August 2006.

[11] L Gonnord and N Halbwachs. Abstract acceleration to improve precision of linear relation analysis. Research report, Verimag, 03 2010.

[12] D. Gopan and T. Reps. Lookahead widening. In *CAV'06*, Seattle, 2006.

[13] A Gupta and A Rybalchenko. Invgen: An efficient invariant generator. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 634–640. Springer, 2009.

[14] N. Halbwachs. Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Thèse de troisième cycle, University of Grenoble, March 1979.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[16] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In A. Cortesi and G. Filé, editors, *Static Analysis Symposium, SAS'99*, Venice (Italy), September 1999. LNCS 1694, Springer Verlag.

[17] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV 2009)*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667, Grenoble, France, June 2009. Springer.

[18] K Ku, T Hart, M Chechik, and D Lie. A buffer overflow benchmark for software model checkers. In *Automated Software Engineering (ASE)*, pages 389–392, Atlanta,USA, 2007. IEEE/ACM.

[19] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[20] J. Leroux. Algorithmique de la vérification des systèmes à compteurs – approximation et accélération – implémentation dans l'outil Fast. Phd thesis, École Normale Supérieure de Cachan, December 2003.

[21] G C. Necula, S McPeak, S Rahul, and W Westley. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228. Springer-Verlag, 2002. LNCS 2304.

[22] S Sankaranarayanan, H Sipma, and Z Manna. Constraint-based linear relations analysis. In *International Symposium on Static Analysis, SAS'2004*, pages 53–68. LNCS 3148, Springer Verlag, 2004.

[23] G. Jr Steele and S.P. Harbison. *C: A Reference Manual.* Prentice Hall, 1984.

[24] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[25] R Wilson, R French, C Wilson, S Amarasinghe, J Anderson, S Tjiang, S-W Liao, C-W Tseng, M Hall, M Lam, and J Henessy. The suif compiler system: a parallelizing and optimizing research compiler. Research Report CSL-TR-94-620, Stanford University, Computer Research Laboratory, 1994.