The ASTRÉE Analyzer*

Patrick Cousot [2], Radhia Cousot [1,3], Jerôme Feret [2], Laurent Mauborgne [2], Antoine Miné [2], David Monniaux [1,2] & Xavier Rival [2]

[1] CNRS
[2] École Normale Supérieure, Paris, France (Firstname.Lastname@ens.fr)
[3] École Polytechnique, Palaiseau, France (Firstname.Lastname@polytechnique.fr)
http://www.astree.ens.fr/

Abstract. ASTRÉE is an abstract interpretation-based static program analyzer aiming at proving automatically the absence of run time errors in programs written in the C programming language. It has been applied with success to large embedded control-command safety critical real-time software generated automatically from synchronous specifications, producing a correctness proof for complex software without any false alarm in a few hours of computation.

1 Introduction

Software development, testing, use, and evolution is nowadays a major concern in many machine-driven human activities. Despite progress in the science of computing and the engineering of software aiming at developing larger and more complex systems, incorrect software is not so uncommon and sometimes quite problematic. Hence, the design of sound and efficient formal program verifiers, which has been a long-standing problem, is a grand challenge for the forthcoming decades.

All automatic proof methods involve some form of approximation of program execution, as formalized by abstract interpretation. They are sound but incomplete whence subject to *false alarms*, that is desired properties that cannot be proved to hold, hence must be signaled as potential problems, even though they do hold at runtime.

Although ASTRÉE addresses only part of the challenge, that of proving the absence of runtime errors in large embedded control-command safety critical real-time software generated automatically from synchronous specifications [1–3], it is a promising first step, in that it was able to make the correctness proof for large and complex software by abstract-interpretation based static analysis [4, 5] in a few hours of computations, without any false alarm.

* This work was supported in part by the French exploratory project ASTRÉE of the Réseau National de recherche et d'innovation en Technologies Logicielles (RNTL).

2 Domain of Application of ASTRÉE

Synchronous C Programs. ASTRÉE can analyze C programs with pointers (including to functions), structures and arrays, integer and floating point computations, tests, loops, function calls, and branching (limited to forward `goto`, `switch`, `break`, `continue`). It excludes `union` types, dynamic memory allocation, unbounded recursive function calls, backward branching, conflicting side effects and the use of C libraries. This corresponds to a clean memory model and semantics as recommended for safety critical embedded real-time synchronous software for non-linear control of very complex control/command systems.

Semantics. The *concrete operational semantics* for the considered subset is that of the international C norm (ISO/IEC 9899:1999) instanced by implementation-specific behaviors depending upon the machine and compiler (e.g., representation and size of integers, IEEE 754-1985 norm for floats and doubles), restricted by user-defined programming guidelines (e.g., whether static variables can or cannot be assumed to be initialized to 0) and finally restricted by program-specific user requirements (such as static assertions). Programs may have a volatile environment where inputs are assumed to be safe (e.g., volatile floats cannot be `NaN`) and may be specified by a trusted configuration file (e.g., specifying physical restrictions on captor values or the maximum number of clock ticks, i.e., of calls to a `wait_for_clock()` function specific to synchronous systems). The *collecting semantics* is the set of partial traces for the concrete operational semantics starting from initial states. The *abstract semantics* is an abstraction of a trace-based refinement of the reachable states.

Specification. The absence of runtime errors is the implicit specification that there is no violation of the C norm (e.g., array index out of bounds), no implementation-specific undefined behaviors (e.g., floating-point division by zero), no violation of the programming guidelines (e.g., arithmetics operators on `short` variables should not overflow the range $[-32768, 32767]$ although, on the specific platform, the result can be well-defined through modular arithmetics), and no violation of the programmer-supplied assertions (which must all be statically verified). It follows that the only possible interrupts are clock ticks, an essential requirement of synchronous programs.

3 Characteristics of ASTRÉE

ASTRÉE is a *program analyzer* (it analyzes directly the program source and not some external specification or program model) which is *static* (the verification is performed before execution), entirely *automatic* (no end-user intervention is needed after parameterization by specialists for adaptation to a category of programs), *semantic-based* (unlike syntactic feature detectors in the spirit of `lint`), *sound* (it covers the whole state space and, contrarily to mere debuggers or bounded-trace software verifiers, never omits a potential error), *terminating* (there is no possibility of non-termination of the analysis), and, in practice, has

shown to be *efficient* (a few hours of computations for hundreds of thousands lines of code).

ASTRÉE is *multi-abstraction* in that it does not use a canonical abstraction but instead uses an approximate reduced cardinal product [5] of many numerical and symbolic abstract domains. The analyses performed by each abstract domain closely interact to perform mutual reductions. The abstraction is *specializable* in that new abstract domains can be easily included or useless ones excluded to adapt the analysis to a given category of programs. The design of ASTRÉE in `Ocaml` is *modular*. An instance of ASTRÉE is built by selecting `Ocaml` modules from a collection, each implementing an abstract domain. Most abstract domains are *infinitary* and *infinite-height*. We use widening/narrowing to enforce convergence. ASTRÉE is *specialized* to a safe programming style but is also *domain-aware* in that it knows about control/command (e.g., digital filters). Each abstract domain is *parametric* so that the precision/cost ratio can be tailored to user needs by options and/or directives in the code. The *automatic parameterization* enables the generation of parametric directives in the code to be programmed. ASTRÉE can therefore be specialized to perform fully automatically for each specific application domain. This design structure makes ASTRÉE both fast and very *precise*: there are very few or no false alarms when conveniently adapted to an application domain. It follows that ASTRÉE is a *formal verifier* that scales up.

4 Design of ASTRÉE by Refinement

ASTRÉE was designed starting from a simple memory model (with references to abstract variables representing either a single or multiple concrete memory locations) and an interval abstraction ($a \leq X \leq b$ where X is a variable and a, b are constants to be determined by the analysis), which is precise enough to express the absence of runtime errors. The widening uses thresholds [1]. This is extremely fast (if sufficient care has been taken to use good data structures) but quite imprecise. Then, numerous abstract domains were designed and experimented until an acceptable cost/precision ratio was obtained. Sometimes, a more precise domain results in an improvement in both analysis precision *and* time (most often because the number of iterations is reduced).

5 The ASTRÉE Fixpoint Iterator

The fixpoint computation of the invariant post-fixpoint [4] is by structural induction on the program abstract syntax, keeping a minimal number of abstract invariants. Functions are handled by semantic expansion of the body (thus excluding unbounded recursion) while convergence is accelerated by non-monotonic widening/narrowing for loops [4, 2]. It follows that the abstract fixpoint transformer is non-monotonic, which is not an issue as it abstracts monotonic concrete fixpoints [6]. Because abstract domains are themselves implemented using floats, possible rounding errors may produce instabilities in the post-fixpoint

check which can be solved thanks to perturbations [2, 7, 8]. Finally, the specification checking is performed by a forward propagation of the stable abstract post-fixpoint invariant.

6 Examples of Numerical Abstractions in ASTRÉE

The Domain of Octagons. An example of numerical abstract domain is the weakly relational domain of octagons [9, 10, 7] ($\pm X \pm Y \leq a$ where X, Y are variables and a is a constant to be determined by the analysis).

```
volatile int vD, vX;                __ASTREE_volatile_input((vD [0,16]));
void main () {                      __ASTREE_volatile_input((vX [-128,128]));
 int D, X, Y = 0, R, S;
 while (1) {
  X = vX; D = vD;
  S = Y; R = X - S; Y = X;
  if (R <= -D) { Y = S - D; }
  else if (D <= R) { Y = S + D; }
 }}
```

Fig. 1. Rate limiter and configuration file.

For instance [7], at each loop iteration of the rate limiter of Fig. 1, a new value for the entry X is fetched within $[-128, 128]$ and a new maximum rate D is chosen in $[0, 16]$. The program then computes an output Y that tries to follow X but is compelled to change slowly: the difference between Y and its value in the preceding iteration is bounded, in absolute value, by the current value of D. The state variable S is used to remember the value of Y at the last iteration while R is a temporary variable used to avoid computing the difference $X - S$ twice. A relational domain is necessary to prove that the output Y is bounded by the range $[-128, 128]$ of X, which requires the discovery of the invariant $R = X - S$. The octagon abstract domain will discover a weaker property, $R + S \in [-128, 128]$, which is precise enough to prove that $Y \in [-M, M]$ is stable whenever $M \geq 144$. So, by widening, M will be set to the least threshold greater than 144 which is loose but precise enough to prove the absence of runtime errors (indeed ASTRÉE finds $Y \in [-261, 261]$). This example is out of the scope of the interval domain.

Heterogeneous Structural Abstraction. The use of the domain of octagons in ASTRÉE is an example of heterogeneous abstraction which depends upon the program structure and is not the same at each program point. Indeed, the octagonal abstraction would be too costly to handle, e.g., thousands of global variables at each program point. The domain of octagons is therefore parameterized by packs of variables attached to blocks/functions by way of directives. These packs specify which variables should be candidate for octagonal analysis in the given block/function. The determination of accurate packs would require

a huge amount of work, if done by hand. Therefore the packing parameterization is automatized using context-sensitive syntactic criteria. Experimentations show that the average pack size is usually of order of 3 or 4 variables while the number of packs grows linearly with the program size. It follows that precise abstractions are performed only when needed, which is necessary to scale up.

Floating-Point Interval Linear Form Abstraction. A general problem with relational numerical domains is that of floating point numbers. Considering them as reals (as usually done with theorem provers) or fixed point numbers (as in CBMC [11]) would not conform to the norm whence would be unsound. Using rationals or other symbolic reals in the abstract domains would be too costly. The general approach [7, 8] has been to define the concrete semantics of floating point computations in the reals (taking the worst possible rounding errors explicitly into account), to abstract with real numbers but to implement, thanks to a further sound over-approximation, using floats. For example the float expression $(x + y) + z$ is evaluated as in the reals as $x + y + z + \varepsilon_1 + \varepsilon_2$ where $|\varepsilon_1| \leq \epsilon_{\text{rel}}.|x + y| + \epsilon_{\text{abs}}$ and $|\varepsilon_2| \leq \epsilon_{\text{rel}}.|x + y + \varepsilon_1 + z| + \epsilon_{\text{abs}}$. The real ε_1 encodes rounding errors in the atomic computation $(x + y)$, and the real ε_2 encodes rounding errors in the atomic computation $(x + y + \varepsilon_1) + z$. The parameters ϵ_{rel} and ϵ_{abs} depends on the floating-point type being used in the analyzed program. This *linearization* [7, 8] of arbitrary expressions is a correct abstraction of the floating point semantics into interval linear forms $[a_0, b_0] + \sum_{k=1}^{n} [a_k, b_k]X_k$. This approach separates the treatment of rounding errors from that of the numerical abstract domains.

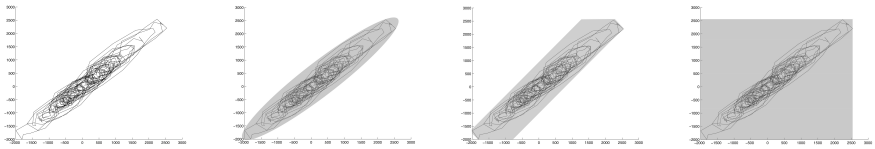

Fig. 2. Filter trace Ellipsoid abstraction Octagon abstraction Interval abstraction

The Simplified Filter Abstract Domains. The simplified filter abstract domains [13] provide examples of domain-aware abstractions. A typical example of simplified filter behavior is traced in Fig. 2 (tracing the sequence `D1` in Fig. 3). Interval and octagonal envelops are unstable because they are rotated and shrunk a little at each iteration so that some corner always sticks out of the envelop. However, the ellipsoid of Fig. 2 is stable. First, filter domains use dynamical linear properties that are captured by the other domains such as the range of input variables (`x1` and `y1` for the example of Fig. 3) and symbolic affine equalities with interval coefficients (to model rounding errors) such as $\texttt{t1} \in [1 - \varepsilon_1, 1 + \varepsilon_1].\texttt{x1} + [b1[0] - \varepsilon_2, b1[0] + \varepsilon_2].\texttt{D1[0]} - [b1[1] - \varepsilon_3, b1[1] + \varepsilon_3].\texttt{D1[1]} + [-\varepsilon, \varepsilon]$ for the example of Fig. 3 (where ε_1, ε_2, and ε_3 describe relative error contributions and ε describes an absolute error contribution). These symbolic equalities are captured either by *linearization* (see Sect. 6), or by symbolic constant propagation (see

```
float A1[3] = { 1, 0.5179422053046, 1.0 };
float b1[2] = { 1.470767736573, 0.5522073405779 };
float A2[3] = { 1, 1.633101801841, 1.0 };
float b2[2] = { 1.742319554830, 0.820939679242 };
float D1[2], D2[2]; float P, X; volatile float E;
void iir4(float *x, float *y)
{ float x1, y1, t1, t2;
  x1 = 0.0117749388721091* *x; t1 = x1 + b1[0]*D1[0] - b1[1]*D1[1];
  y1 = A1[0]*t1 - A1[1]*D1[0] + A1[2]*D1[1]; D1[1] = D1[0]; D1[0] = t1;
  t2 = y1 + b2[0]*D2[0] - b2[1]*D2[1];
  *y = A2[0]*t2 - A2[1]*D2[0] + A2[2]*D2[1]; D2[1] = D2[0];  D2[0] = t2;
  __ASTREE_log_vars((P,y1,x1,t2;ellipse))
}
int main () { while (1) { X = E; iir4(&X,&P); __ASTREE_log_vars((P)); }}
```

Fig. 3. Fourth order Infinite Impulse Response (IIR) filter [12].

Sect. 7). Then, simplified filter domains infer non linear properties and compute bounds on the range of output variables (`t1` and `t2` in Fig. 2). For the example of Fig. 3, ASTRÉE over-approximates the interval of variation of `D2[0]` by $[-6890.23, 6890.23]$, which is precise enough to prove the absence of overflow.

On the Limits of User-Provided Assertions. The filter ellipsoidal abstraction illustrates the limits of user provided assertions. Even if the user injects the correct bounds, as an interval information, for all filter outputs, the interval domain cannot exploit them as they are not stable. To reach zero false alarm, the abstract domains should be able to express a loop invariant which is strong enough to be inductive and to imply the absence of runtime errors. User assertions are therefore useful only when they refer to assertions expressible in the abstract domains of a static analyzer. They are mainly useful to provide widening/narrowing limits but techniques such as widenings with thresholds are even more convenient.

On the Limits of Automatic Refinement. The filter ellipsoidal abstraction shows the limits of automatic refinement strategies based on counter-examples. From a finite series of counter-examples to the stability of intervals or octagons, the refinement procedure would have to automatically discover the ellipsoidal abstraction and infer the corresponding sophisticated data representations and algorithms.

The Arithmetic-Geometric Progression Abstract Domain. In synchronous programs, the arithmetic-geometric progression abstract domain [14] can be used to estimate ranges of floating point computations that may diverge in the long run due to rounding errors (although they may be stable in the reals) thanks to a relation with the clock, which, for physical systems that cannot run for ever, must be bounded. In the example of Fig. 4, the bound of `B` is:

$$|\mathtt{B}| \leq a * ((20. + b/(a-1)) * (a)^{\mathrm{clock}} - b/(a-1)) + b \leq 30.7191369175$$

where $a = 1.00000011921$ and $b = 5.87747175411e-39$.

```
volatile float E,T;                __ASTREE_volatile_input((T[-1.0,1.0]));
float A,B,X;                       __ASTREE_volatile_input((E[-20.0,20.0]));
int main () {                      __ASTREE_max_clock((3600000));
 while (1) {
  if (T>0){X = E;}
  else {X = B;}
  B = B-(((2.0*B)-A-X)*0.005);
  A = X;
  __ASTREE_wait_for_clock(());
 }}
```

Fig. 4. Geometric progression and configuration file.

Relative Precision of Abstract Domains. The arithmetic-geometric progression abstract domain provides an example of sophisticated domain that can advantageously replace a simpler domain, specifically the *clock domain* [1], formerly used in ASTRÉE, relating each variable X to the bounded clock C (incremented on clock ticks) as intervals for $X - C$ and $X + C$ to indirectly bound X from the user-provided bound on the clock C.

7 Examples of Symbolic Abstractions in ASTRÉE

Memory Abstraction. A first example of symbolic domain in ASTRÉE is the memory abstraction model shortly described in [2].

The Symbolic Constant Domain. The symbolic constant domain [7, 8] is reminiscent of Kildall's constant propagation abstract domain, that is the smash product of infinitely many domains of the form $\bot \sqsubset e \sqsubset \top$, but memorizes symbolic expressions e instead of numerical constants. It keeps track of symbolic relations between variables and performs simplifications (such as simplifying `Z=X; Y=X-Z` into `Z=X; Y=0`, which does appear in mechanically generated programs). Such relational information is essential for the interval abstract domain (e.g., to derive that $Y = 0$). Again the abstract domain is parameterized (e.g., by simplification strategies).

The Boolean Relation Domain. The Boolean relation domain [2] copes with the use of booleans in the control of synchronous programs. It is a reduced cardinal power [5] with boolean base implemented as a decision tree with sharing (à la BDD) and exponents at the leaves. It is parametric in the maximum number of boolean variables and in the packs of variables which are involved at the leaves. The maximum number 3 was determined experimentally and the packing is automatized.

The Expanded Filter Abstract Domains. The expanded filter abstract domains associate recursive sequence definitions to tuples of numerical variables automatically detected by the analysis [13]. For instance, a second order filter is encoded by a recursive definition of the form $S_{n+2} = a.S_{n+1} + b.S_n +$

$c.E_{n+2} + d.E_{n+1} + e.E_n$ ((E_n) denotes an input stream and (S_n) denotes an output stream). The second order filter domain relates abstract values M to the quadruples of variables (V, W, X, Y) detected by the analysis. This symbolic property means that there exists a positive integer p and a recursive sequence satisfying $S_{n+2} = a.S_{n+1} + b.S_n + c.E_{n+2} + d.E_{n+1} + e.E_n$, for any positive integer n, such that:

- $V = S_{p+1}$, $W = S_p$, $X = E_{p+1}$, and $Y = E_p$;
- the abstract value M gives an abstraction of the values S_0 and S_1 and of the sequence (E_n).

We compute a bound on the value of V, by unfolding the recursive definition several times (so that we describe the contribution of the last inputs very accurately). The contributions of rounding errors and of previous inputs are bounded by using a simplified filter domain (the ellipsoid domain [13] in our example).

Trace Partitioning. Trace partitioning [15] is a local parametric symbolic abstraction of sets of traces, which is a local refinement of reachable states. By relative completeness of Floyd's proof method, it is useless to reason on traces and sets of states should be precise enough. However, this greatly simplifies the abstraction which would otherwise require to establish more relations among variables. Examples are loop unrolling, case analysis for tests, etc.

8 Performances of ASTRÉE.

ASTRÉE has been applied with success to large embedded control-command safety critical real-time software generated automatically from synchronous specifications, producing a correctness proof for complex software without any false alarm in a few hours of computations (see Fig. 5).

Nb of lines	70 000	226 000	400 000
Number of iterations	32	51	88
Memory	599 Mb	1.3 Gb	2.2 Gb
Time	46mn	3h57mn	11h48mn
False alarms	0	0	0

Fig. 5. Performance of ASTRÉE (64 bits monoprocessor).

9 ASTRÉE Visualisator

According to the desired information, it is possible to serialize the invariants and alarms attached by ASTRÉE to program points, blocks, loops or functions and to vizualize them, per abstract domain, using a graphical vizualisator to navigate in the program invariants as shown on Fig. 6.

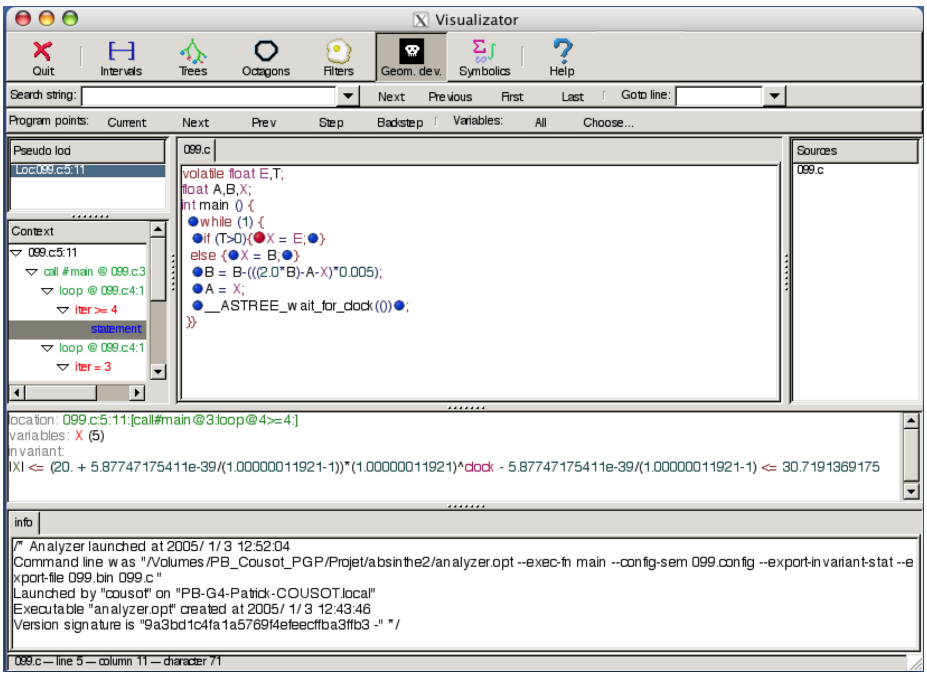

Fig. 6. Vizualisator.

10 Conclusion and Future Work

Abstract interpretation-based static analyzers have recently shown to scale-up for different programming languages and different areas of application [16, 17]. ASTRÉE is certainly the first static analyzer able to *fully prove automatically* the absence of runtime errors in real-life large industrial synchronous programs. It is therefore a verifier (as opposed to a debugger or testing aid). In case of erroneous source programs, an assistance to error source localization is presently being incorporated in ASTRÉE thanks to backward analyses. By extension of the abstract domains, ASTRÉE can be extended beyond synchronous programs.

To go beyond and generalize to more complex memory models and asynchronous programs will necessitate a complete redesign of the basic memory abstraction and fixpoint iterators. This will be the object of ASTRÉE successors.

Acknowledgements We warmly thank Bruno Blanchet for his contribution to ASTRÉE.

Bibliography

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In Mogensen,

T., Schmidt, D., Sudborough, I., eds.: The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. LNCS 2566. Springer (2002) 85–108
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. ACM SIGPLAN '2003 Conf. PLDI, San Diego, ACM Press (2003) 196–207
3. Mauborgne, L.: ASTRÉE: Verification of absence of run-time error. In Jacquart, P., ed.: Building the Information Society. Kluwer Academic Publishers (2004) 385–392
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4^{th} ACM POPL. (1977) 238–252
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6^{th} ACM POPL. (1979) 269–282
6. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. of Logic and Comput. **2** (1992) 511–547
7. Miné, A.: Weakly Relational Numerical Abstract Domains. Thèse de doctorat en informatique, École polytechnique, Palaiseau, France (2004)
8. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In Schmidt, D., ed.: Proc. 30^{th} ESOP '2004, Barcelona. LNCS 2986, Springer (2004) 3–17
9. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In Danvy, O., Filinski, A., eds.: Proc. 2^{nd} Symp. PADO '2001. Århus, 21–23 May 2001, LNCS 2053, Springer (2001) 155–172
10. Miné, A.: The octagon abstract domain. In: AST 2001 in WCRE 2001. IEEE, IEEE CS Press (2001) 310–319
11. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In Jensen, K., Podelski, A., eds.: Proc.10th Int. Conf. TACAS '2004, Barcelona. LNCS 2988, Springer (2004) 168–176
12. Aamodt, T., Chow, P.: Numerical error minimizing floating-point to fixed-point ANSI C compilation. In: 1st Workshop on Media Processors and DSPs. (1999) 3–12
13. Feret, J.: Static analysis of digital filters. In Schmidt, D., ed.: Proc. 30^{th} ESOP '2004, Barcelona. LNCS 2986, Springer (2004) 33–48
14. Feret, J.: The arithmetic-geometric progression abstract domain. In Cousot, R., ed.: Proc. 6^{th} VMCAI '2005, Paris. LNCS 3385, Springer (2004) 2–58
15. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In Sagiv, M., ed.: Proc. 31^{th} ESOP '2005, Edinbugh. This volume of LNCS., Springer (2005)
16. Alt, M., C., F., Martin, F., Wilhelm, R.: Cache behavior prediction by abstract interpretation. In Cousot, R., Schmidt, D., eds.: Proc. 3^{rd} Int. Symp. SAS '96. Aachen, 24–26 sept. 1996, LNCS 1145. Springer (1996) 52–66
17. Venet, A., Brat, G.: Precise and efficient static array bound checking for large embedded C programs. In: Proc. ACM SIGPLAN '2004 Conf. PLDI, Washington DC, ACM Press (2004) 231–242