The parallel implementation of the Astrée static analyzer

David Monniaux

David.Monniaux@ens.fr http://www.di.ens.fr
Centre national de la recherche scientifique (CNRS)
École normale supérieure, Laboratoire d'Informatique, 45 rue d'Ulm,
75230 Paris cedex 5, France

Abstract. The Astrée static analyzer is a specialized tool that can prove the absence of runtime errors, including arithmetic overflows, in large critical programs. Keeping analysis times reasonable for industrial use is one of the design objectives. In this paper, we discuss the parallel implementation of the analysis.

1 Introduction

The Astrée static analyzer[1] is a tool that analyzes, fully automatically, single-threaded programs written in a subset of the C programming language, sufficient for many typical critical embedded programs. The tool particularly targets control/command applications using many floating-point variables and numerical filters, though it has been successfully applied to other categories of software. It computes a super-set of the possible run-time errors. Astrée is designed for efficiency on large software: hundreds of thousands of lines of code are analyzed in a matter of hours, while producing very few false alarms. For example, some fly-by-wire avionics reactive control codes (70 000 and 380 000 lines respectively, the latter of a much more complex design) are analyzed in 1 h and 10 h 30' respectively on current single-CPU PCs, with *no false alarm* [1,2,9].

Other contributions [10,14,15,16,17,18] have described the *abstract domains* used in Astrée; that is, the data structures and algorithms implementing the symbolic operations over abstract set of reachable states in the program to be analyzed. However, the operations in these abstract domains must be driven by an *iterator*, which follows the control flow of the program to be analyzed and calls the necessary operations. This paper describes some characteristics of the iterator. We first explain some peculiarities of our iteration algorithm as well as some implementation techniques regarding efficient shared data structures. These have an impact on the main contribution of the paper, which is the parallelization technique implemented in Astrée.

Even though Astrée presented good enough performances to be used in practical settings on large-scale industrial code on single-processor systems, we

[1] http://www.astree.ens.fr

designed a parallel implementation suitable both for shared-memory multiprocessor systems and for small clusters of PCs over local area networks. ASTRÉE being focused on synchronous, statically scheduled reactive programs, we used that peculiar form of the program to be analyzed in order to design a very simple, yet efficient, parallelization scheme. We however show that the control-flow properties that enable such parallel analysis are to be found in other kinds of programs, including major classes of programs such as event-driven user interfaces.

Section 2 describes the overall structure of the interpreter and the most significant choices about the iteration strategy. This defines the framework within which we implement our parallelization scheme. We also discuss implementation choices for some data structures, which have a large impact on the simplicity and efficiency of the parallel implementation.

Section 3 describes the parallelization of the abstract interpreter in a range of practical cases.

2 The ASTRÉE Abstract Interpreter

Our static analyzer is structured in several hierarchical layers:

– a denotational-based *abstract interpreter* abstractly executes the instructions in the programs by sending orders to the abstract domains;

– a *partitioning domain* [12] handles the partitioning of traces depending on various criteria; it also operates the partitioning with respect to the call stack;

– a *branching abstract domain* handles forward branching constructs such as forward `goto`, `break`, `continue`;

– a *structure abstract domain* resolves all accesses to complex data structures (arrays, pointers, records...) into may- or must-aliases over *abstract cells* [2, §6.1];

– various numerical domains express different kinds of constraints over those cells; each of these domains can query other domains for information, and send information to those domains (*reduction*).

2.1 A denotational-based interpreter

Contrary to some presentations or examples of abstract interpretation-based static analysis [7] , we did not choose to obtain results through the direct resolution (or over-approximation) of a system of semantic equations, but rather to follow the denotational semantics of the program as in [6, Sect. 13].

Consider the following fragment of the C programming language:

$l ::= x \mid t[e] \mid \ldots$ l-values

$e ::= l \mid e \oplus e \mid \ominus e \mid \ldots$ expressions ($\oplus \in \{+, \star, \ldots\}; \ominus \in \{-, \ldots\}$)

$s ::= \tau\ x; \mid l = e \mid \textbf{if}(e)\{s; \ldots; s\}\textbf{else}\{s; \ldots; s\} \mid \textbf{while}(e)\{s; \ldots; s\}$

L is the set of control states, τ is any type, $\mathbb{L}$ (resp. $\mathbb{E}$) is the set of l-values (resp. expressions). The concrete semantics of a statement s is a function $[\![s]\!] : M \to \mathcal{P}(M) \times \mathcal{P}(E)$ where M (resp. E) is the set of memory states (resp. errors). Given an abstraction of sets of stores defined by an abstract domain $D^\sharp_M$

and a concretization function $\gamma_M : D_M^\sharp \to \mathcal{P}(M)$, we can derive an approximate abstract semantics $[\![P]\!]^\sharp : D_M^\sharp \to D_M^\sharp$ of program fragment P by following the methodology of abstract interpretation [8].

The soundness of $[\![P]\!]^\sharp$ can be stated as follows: if $[\![P]\!](\rho) = (m_0, e_0)$ and $\rho \in \gamma_M(d^\sharp)$, then $m_0 \subseteq \gamma_M([\![P]\!]^\sharp(d^\sharp))$ (resp. for the error list). The principle of the interpreter is to compute $[\![P]\!]^\sharp(d^\sharp)$ by induction on the syntax of P. $D_M^\sharp$ should provide abstract counterparts (**assign**, **new_var**, **del_var**, **guard**) to the concrete operations (assignment, variable creation and deletion, condition test). For instance, **assign** should be a function in $\mathbb{L} \times \mathbb{E} \times D_M^\sharp \to D_M^\sharp$, that inputs an l-value, an expression and an abstract value and returns a sound over-approximation of the set of stores resulting from the assignment: $\forall l \in \mathbb{L}, \forall e \in \mathbb{E}$, $\forall d^\sharp \in D_M^\sharp$, $\{\rho[[\![l]\!](\rho) \mapsto [\![e]\!](\rho)] \mid \rho \in \gamma_M(d^\sharp)\} \subseteq \gamma_M(\mathbf{assign}(l, e, d^\sharp))$. Soundness conditions for the other operations (**guard**, **new_var**, **del_var**) are similar.

$$
\begin{aligned}
[\![l = e;]\!]^\sharp(d^\sharp) &= \mathbf{assign}(l, e, d^\sharp) \quad \text{where } l \in \mathbb{L}, e \in \mathbb{E} \\
[\![\{\tau\, x; s_0\}]\!]^\sharp(d^\sharp) &= \mathbf{del\_var}(\tau\, x, [\![s_0]\!]^\sharp(\mathbf{new\_var}(\tau\, x, d^\sharp))) \\
[\![\mathbf{if}(e)\, s_0\, \mathbf{else}\, s_1;]\!]^\sharp(d^\sharp) &= [\![s_0]\!]^\sharp(\mathbf{guard}(e, \mathtt{t}, d^\sharp)) \sqcup [\![s_1]\!]^\sharp(\mathbf{guard}(e, \mathtt{f}, d^\sharp)) \\
[\![\mathbf{while}(e)\, s_0]\!]^\sharp(d^\sharp) &= \mathbf{guard}(e, \mathtt{f}, \mathrm{lfp}^\sharp \phi^\sharp) \\
&\qquad \text{where: } \phi^\sharp : x^\sharp \in D_M^\sharp \mapsto d^\sharp \sqcup [\![s_0]\!]^\sharp(\mathbf{guard}(e, \mathtt{t}, x^\sharp))
\end{aligned}
$$

The function $\mathrm{lfp}^\sharp$ computes a post-fixpoint of any abstract function (i.e., approximation of the concrete least-fixpoint). While the actual scheme implemented is somewhat complex, it is sufficient to say that $\mathrm{lfp}^\sharp f^\sharp$ outputs some $x^\sharp$ such that $f^\sharp(x^\sharp) \sqsubseteq^\sharp x^\sharp$ for some decidable ordering $\sqsubseteq^\sharp$ such that $\forall x^\sharp, y^\sharp\ x^\sharp \sqsubseteq^\sharp y^\sharp \implies \gamma(x^\sharp) \subseteq \gamma(y^\sharp)$. This abstract fixpoint is sought by the iterator in "iteration mode": possible warnings that could occur within the code are not displayed when encountered. Then, once $L^\sharp = \mathrm{lfp}^\sharp f^\sharp$ is computed — an invariant for the loop body —, the iterator analyzes the loop body again and displays possible warnings. As a supplemental safety measure, we check again that $f^\sharp(L^\sharp) \sqsubseteq^\sharp L^\sharp$.[2]

$\sqcup$ is an abstraction of the concrete union $\cup$: $\forall {d_1}^\sharp, {d_2}^\sharp, \gamma({d_1}^\sharp) \cup \gamma({d_2}^\sharp) \subseteq \gamma({d_1}^\sharp \sqcup {d_2}^\sharp)$.

An abstract domain handles the call stack; currently in Astrée, it amounts to partitioning states by the full calling context [12]. Astrée does not handle recursive functions;[3] this is not a problem with critical embedded code, since

[2] Let us note that the computationally costly part of the analysis is finding the loop invariant, rather than checking it. P. Cousot suggested the following improvement over our existing analysis: using different implementations for finding the invariant and checking it (at present, the same program does both). . For instance, the checking phase could be a possibly less efficient version, whose safety would be formally proved. However, since all abstract domains and most associated algorithms would have to be implemented in that "safe" analyzer, the amount of work involved would be considerable and we have not done it at this point. Also, as discussed in Sect. 3.2, both implementations would have to yield identical results, which means that the "safe" analysis would have to mimic the "unsafe" one in detail.

[3] More precisely, it can analyze recursive programs, but analysis may fail to terminate. If analysis terminates, then its results are sound.

programming guidelines for such systems generally prohibit the use of recursive functions. Functions are analyzed as if they were inlined at the point of call. Multiple targets for function pointers are analyzed separately, and the results merged with $\sqcup$; see §3 for an application to parallelization.

Other forms of branches are dealt with by an extension of the abstract semantics. Explicit gotos are rarely used in C, except as forward branches to error handlers or for exiting multiple loops; however, semantically similar branching structures are very usual and include **cases** structures, **break** statements and **return** statements. Indeed, a return statement **return** e carries out two operations: first, it evaluates e and stores the value as the function result; then, it terminates the current function, i.e. branches to the end of the function. In this paper, we only consider the case of forward-branching **goto**'s; the other constructs then are straightforward extensions.

We extend the syntax of statements with a goto statement **goto** l where l is a program point (we implicitly assume that there is a label before each statement in the program). The execution of a statement s may yield either a new memory state or a branching to a point after s. Therefore, we lift the definition of the semantics into a function $[\![s]\!] : (M \times (L \to \mathcal{P}(M))) \to (\mathcal{P}(M) \times (L \to \mathcal{P}(M)) \times E)$. The concrete states before and after each statement no longer consist solely of a set of memory states, but of a set of memory states for the "direct" control-flow as well as a set of memory states for each label l, representing all the memory states that have occurred in the past for which a forward branch to l was requested.

The concrete semantics of **goto** l is defined by: $[\![\textbf{goto } l;]\!](I_i, \phi_i) = (\bot, \phi_i[l \mapsto \phi_i(l) \cup I_i])$ and the concrete semantics of a statement s at label l is defined from the semantics without branches as: $[\![l : s;]\!](I_i, \phi_i) = (\{I_i\} \cup \phi_i(l), \phi_i)$. The definition of the abstract semantics can be extended in the same way. We straightforwardly lift the abstract semantics of a statement s into a function $[\![s]\!]^\sharp : D^\sharp_M \times (L \to D^\sharp_M) \to D^\sharp_M \times (L \to D^\sharp_M)$.

2.2 Rationale and efficiency issues

The choice of the denotational approach was made for two reasons :

- Iteration and widening techniques on general graph representations of programs are more complex. Essentially, these techniques partly have to reconstruct the natural control flow of the program so as to obtain an efficient propagation flow [3,11]. Since our programs are block-structured and do not contain backward `goto`'s, this flow information is already present in their syntax; there is no need to reconstruct it. [6, Sect. 13]
- It minimizes the amount of memory used for storing the abstract environments. While our storage methods maximize the sharing between abstract environments, our experiments showed that storing an abstract environment for each program point (or even each branching point) in main memory was too costly. Good forward/backward iteration techniques do not need to store environments at that many points, but this measurement still was an indication that there would be difficulties in implementing such schemes.

We measured the memory required for storing the local invariants at part or all of the program points, for three industrial control programs representative of those we are interested in; see the table below. We performed several measurements, depending on whether invariant data was saved at all statements, at the beginning and end of each block, and at the beginning and end of each function.

For each program and measurement, we provide two figures: from left to right, the peak memory observed during the analysis, then the size of the serialized invariant (serialization is performed for saving to files or for parallelization purposes, and preserves the sharing property of the internal representation).

Benchmarks (see below) show that keeping local invariants at the boundaries of every block in main memory is not practical on large programs; even restricting to the boundaries of functions results in a major overhead. A database system for storing invariants on secondary storage could be an option, but Brat and Venet have reported significant difficulties and complexity with that approach [4]. Furthermore, such an approach would complicate memory sharing, and perhaps force the use of solutions such as "hash-consing", which we have avoided so far.

Memory requirements are expressed in megabytes; analyses were run in 64-bit mode on a dual Opteron 2.2 GHz machine with 8 Gb RAM.[4] On many occasions, we had to abort the computation due to large memory requirements causing the system to swap.

	Program 1		Program 2		Program 3	
# of lines of C code	67,553		232,859		412,858	
# of functions	650		1,900		2,900	
Save at all statements	3300	688	> 8000	swap	> 8000	swap
Save at beginning / end of blocks	2300	463	> 8000	swap	> 8000	swap
Save at beginning / end of functions	690	87	2480	264	4800	428
Save main loop invariant only	415	15	1544	53	2477	96
No save	410		1544		2440	

Benchmarks courtesy of X. Rival.

Memorizing invariants at the head of loops (the least set of invariants we can keep so as to be able to compute widening chains) thus entails much smaller memory requirements than a naïve graph-based implementation; the latter is intractable on reasonably-priced current computers on the class of large programs that we are interested in. It is possible that more complex memorization schemes may make graph-based algorithms tractable, but we did not investigate such schemes because we had an efficient and simple working system.

Regarding efficiency, it soon became apparent that a major factor was the efficiency of the $\sqcup$ operation. In a typical program, the number of tests will be roughly linear in the length of the code. In the control programs that Astrée targets, the number of state variables (the values of which are kept across iterations) is also roughly linear in the length l of the code. This means that if the

[4] Memory requirements are smaller on 32-bit systems.

$\sqcup$ operation takes linear time in the number of variables — an apparently good complexity —, an iteration of the analyzer takes $\Theta(l^2)$ time, which is prohibitive. We therefore argue that *what matters is the complexity of* $\sqcup$ *with respect to the number of updated variables*, which should be almost linear: if only n_1 (resp. n_2) variables are touched in the `if` branch (resp. `else` branch), then the overall complexity should be at most roughly $O(n_1 + n_2)$.

We achieve such complexity with our implementation using balanced trees with strong memory sharing and "short-cuts" [1, §6.2]. Experimentation showed that memory sharing was good with the rough physical equality tests that we implement, without the need for much more costly techniques such as hash consing. Indeed, experiments show that considerable sharing is kept after the abstract execution of program parts that modify only parts of the global state (see the Δ-compression in §3.2). Though simple, this memory-saving technique is fragile; data sharing must be conserved by all modules in the program. This obligation had an impact on the design of the communications between parallel processes.

3 Parallelization

In iteration mode, we analyze tests in the following way: $[\![\mathbf{if}(e)\, s_0\, \mathbf{else}\, s_1;]\!]^\sharp(d^\sharp)$ $= [\![s_0]\!]^\sharp(\mathbf{guard}(e, \mathtt{t}, d^\sharp)) \sqcup [\![s_1]\!]^\sharp(\mathbf{guard}(e, \mathtt{f}, d^\sharp))$. The analyses of s_0 and s_1 may be conducted in total separation, in different threads or processes, or even on different machines. Similarly, the semantics of an indirect function call may be approximated as: $[\![\texttt{(*f)();}]\!]^\sharp(a^\sharp) = \bigsqcup_{g \in [\![\mathtt{f}]\!](a^\sharp)} [\![g]\!]^\sharp(a^\sharp)$: g ranges on all the possible code blocks to which `f` may point.

3.1 Dispatch points

In usual programs, most tests split the control flow between short sequences of execution; the overhead of analyzing such short paths in separate processes would therefore be considerable with respect to the length of the analysis itself. However, there exist wide classes of programs where a few tests (at "dispatch points") divide the control flow between long executions. In particular, there exist several important kinds of software consisting in a large event loop: the system waits for an event, then a "dispatcher" runs an appropriate (often fairly complex) handler routine:

Initialization
while true do
  wait for a request r
  dispatch according to the type of r **in**
    handler for requests of the first type
    handler for requests of the second type
    ...
done

This program structure is quite frequent in network services and traditional graphical user interfaces (though nowadays often wrapped inside a callback mechanism):

Initialization
while true do
  wait for an event e
  dispatch according to e **in**
    event handler 1
    event handler 2
    ...
done

Many critical embedded programs are also of this form. We analyze reactive programs that, for the most part, implement in software a directed graph of numeric filters. Those numeric filters are in general the discrete-time counterparts of hardware, continuous-time components, with various sampling rates. The system is thus made of n components, each clocked with a period $p_i \cdot p$, where $1/p$ is a master clock rate (say, 1 kHz). It is statically scheduled as a succession of "sequencers" numbered from 0 to $N-1$ where N is the least common multiple of the p_i. A task of period $p_i.p$ is scheduled in all sequencers numbered $k.p_i + c_i$. c_i may often be arbitrarily chosen; judicious choices of c_i allow for static load balancing, especially with respect to worst-case execution time (all sequencers should complete within a time of p). Thus, the resulting program is of the form:

Initialization
while true do
  wait for clock tick ($1/p$ Hz)
  dispatch according to i **in**
    sequencer 0
    sequencer 1
    ...
  $i := i + 1 \pmod N$
done

Our analysis is imprecise with respect to the succession of values of i; indeed, it soon approximates i by the full interval $[0, N-1]$. This is equivalent to saying that any of the sequencers is nondeterministically chosen. Yet, due to the nature of the studied system, this is not a hindrance to proving the absence of runtime errors: each of the n subcomponents should be individually sound, whatever the sampling rate, and the global stability of the system does not rely on the sampling rates of the subcomponents, within reasonable bounds.

Our system could actually handle parallelization at any place in the program where there exist two or more different control flows, by splitting the flows between several processors or machines; it is however undesirable to fork processes or launch remote analyses for simple blocks of code. Our current system decides the splitting point according to some simple ad-hoc criterion, but we

could use some more universal criteria. For instance, the analyzer, during the first iteration(s), could measure the analysis times of all branches in `if`, `switch` or multi-aliases function calls; if a control flow choice takes place between several blocks for which the analysis takes a long time, the analysis could be split between several machines in the following iterations. To be efficient, however, such a system would have to do some relatively complex workload allocation between processors and machines; we will thus only implement it when really necessary.

3.2 Parallelization implementation

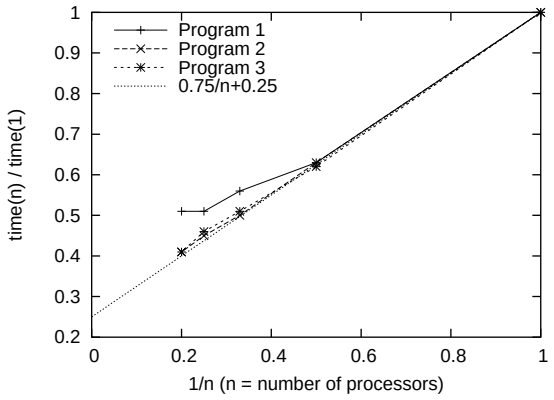

Fig. 1. Parallelization performances (dual-2.2 GHz Opteron machine + 2 GHz AMD64 machines).

Instead of analyzing all dispatch branches in sequence, we split the workload between p several processors (possibly in different machines). We replace the iterative computation of $f^\sharp(X^\sharp) = [\![P_1]\!]^\sharp(X^\sharp) \sqcup ([\![P_2]\!]^\sharp(X^\sharp \sqcup \ldots [\![P_n]\!]^\sharp(X^\sharp))\ldots)$ by a parallel computation $f^\sharp(X^\sharp) = \bigsqcup_{i=1}^{p}(\bigsqcup_{k\in\pi_i} [\![P_k]\!]^\sharp(X^\sharp))$ where the π_k are a partition of $\{1,\ldots,n\}$. Let us note τ_j the time needed to compute $[\![P_j]\!]^\sharp$. $l_k = \sum_{j\in\pi_k} \tau_j$ is the time spent by processor i.

For maximal efficiency, we would prefer that the l_i should be close to each other, so as to minimize the synchronization waits. Unfortunately, the problem of optimally partitioning into the π_k is NP-hard even in the case where $p = 2$ [13]. If the τ_i are too diverse, randomly shuffling the list may yield improved performance. In practice, the real-time programs that we analyze are scheduled so that all the P_i have about the same worst-case execution time, so as to ensure maximal efficiency of the embedded processor; consequently, the τ_i are reasonably close together and random shuffling does not bring significant improvement; in fact, in can occasionally reduce performances.

For large programs of the class we are interested in, the analysis times (Fig. 1) for n processors is approximately $0.75/n + 0.25$ times the analysis time on one processor; thus, clusters of more than 3 or 4 processors are not much interesting:

	Prog 1	Prog 2	Prog 3
# lines	67,553	232,859	412,858
1 CPU	26'28"	5h 55'	11h 30'
2 CPU	16'38"	3h 43'	7h 09'
3 CPU	14'47"	2h 58'	5h 50'
4 CPU	13'35"	2h 38'	5h 06'
5 CPU	13'26"	2h 25'	4h 44'

Venet and Brat also have experimented with parallelization [19, §5], with similar conclusions; however, the class of programs to be analyzed and the expected precisions of their analysis are too different from ours to make direct comparisons meaningful.

Because it is difficult to determine the τ_i in advance, ASTRÉE features an optional randomized scheduling strategy, which reduces computation times on our examples by 5%, with computation times on 2 CPUs $\simeq$58% of those on 1.

We reduced transmission costs by sending only the differences between abstract values at the input and the output — when the remote computation is $[\![\mathtt{P}]\!]^\sharp(d^\sharp)$, only answer the difference between $d^\sharp$ and $[\![\mathtt{P}]\!]^\sharp(d^\sharp)$. This difference is obtained by physical comparison of data structures, excluding shared subtrees (Sect. 2.2). The advantage of that method is twofold:

- Experimentally, such "Δ-compression" results in transmissions of about 10% of the full size on our examples. This reduces transmission costs on networked implementations.
- Recall that we make analysis tractable by sharing data structures (Sect. 2.2). We however enforce this sharing by simple pointer comparisons (i.e. we do not construct another copy of a node if our procedure happens to have the original node at its disposal), which is fast and simple but does not guarantee optimal sharing. Any data coming from the network, even though logically equal to some data already present in memory, will be loaded at a different location; thus, one should avoid merging in redundant data. Sending only the difference back to the master analyzer thus dramatically reduces the amount of unshared data created by networked merge operations.

We request that the $\sqcup$ operator should be associative and commutative, so that $f^\sharp$ does not depend on the chosen partitioning. Such a dependency would be detrimental for two reasons:

- If the subprograms $P_1, \ldots, P_n$ are enclosed within a loop, the nondeterminism of the abstract transfer function $f^\sharp$ complicates the analysis of the loop. As we said in Sect. 2.1, we use an "abstract fixpoint" operator $\mathrm{lfp}^\sharp$ that terminates when it finds $L^\sharp$ such that $f^\sharp(L^\sharp) \sqsubseteq L^\sharp$. Because this check is performed at least twice, it would be undesirable that the comparisons yield inconsistent results.

– For debugging and end-user purposes, it is undesirable that the results of the analysis could vary for the same analyzer and inputs because of runtime vagaries. [5]

In this case of a loop around the $P_1, \ldots, P_n$, we could have alternatively used asynchronous iterations [5]. To compute $\mathrm{lfp} f^\sharp$, one can use a central repository $X^\sharp$, initially containing $\bot$; then, any processor i computes $f_i^\sharp(X^\sharp) = \bigsqcup_{k \in \pi_i} [\![P_k]\!]^\sharp(X^\sharp)$ and replaces $X^\sharp$ with $X^\sharp \triangledown f_i^\sharp(X^\sharp)$. If the scheduling is fair (no $[\![P_k]\!]$ is ignored indefinitely), such iterations converge to an approximation of the least fixpoint of $X \mapsto \cup_k [\![P_k]\!](X)$. However, we did not implement our analyzer this way. Apart from the added complexity, the nondeterminism of the results was undesirable.

4 Conclusion

We have investigated both theoretical and practical matters regarding the computation of fixpoints and iteration strategies for static analysis of single-threaded, block-structured programs, and proposed methods especially suited for the analysis of large synchronous programs: a denotational iteration scheme, maximal data sharing between abstract invariants, and parallelization schemes. In several occasions, we have identified possible extensions of our system.

Two major problems we have had to deal with were long computation times, and, more strikingly, large memory requirements, both owing to the very large size of the programs that we consider. Additionally, we had to keep a very high precision of analysis over complex numerical properties in order to be able to certify the absence of runtime errors in the industrial programs considered.

We think that several of these methods will apply to other classes of programs. Parallelization techniques, perhaps extended, should apply to wide classes

[5] For the same reasons, care should be exercised in networked implementations so that different platforms output the same analysis results on the same inputs. Subtle problems may occur in that respect; for instance, there may be differences between floating-point implementations. We use the native floating-point implementation of the analysis platform; even though all our host platforms are IEEE-compatible, the exact same analysis code may yield different results on various platforms, because implementations are allowed to provide more precision that requested by the norm. For example, the IA32™(Intel Pentium™) platform under Linux™(and some other operating systems) computes by default internally with 80 bits of precision upon values that are specified to be 64-bit IEEE double precision values. Thus, the result of computations on that platform may depend on the register scheduling done by the compiler, and may also differ from results obtained on platforms doing all computations on 64-bit floating point numbers (PowerPC™, and even IA32™and AMD64™with some code generation and system settings). Analysis results, in all cases, would be sound, but they would differ between implementations, which would be undesirable for the sake of reproducibility and debugging, and also for parallelization, as explained here. We thus force (if possible) the use of double (and sometimes single) precision IEEE floating-point numbers in all computations within the analyzer.

of event-driven programs; loop iteration techniques should apply to any single-threaded programs; data sharing and "union" optimizations should apply to any static analyzer. We also have identified various issues of a generic interest with respect to widenings and narrowings.

Acknowledgments: We wish to thank P. Cousot and X. Rival, as well as the rest of the ASTRÉE team.

References

1. B. Blanchet et al. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, number 2566 in LNCS, pages 85–108. Springer, 2002.
2. B. Blanchet et al. A static analyzer for large safety-critical software. In *PLDI*, 2003.
3. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *LNCS*, 735:128, 1993.
4. G. Brat and A. Venet. Precise and scalable static program analysis of nasa flight software. In *IEEE Aerospace Conference*, 2005.
5. P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Technical Report 88, IMAG Lab., 1977.
6. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
7. P. Cousot and R. Cousot. Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, Los Angeles, CA, January 1977.
8. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Prog.*, 2-3(13):103–179, 1992.
9. P. Cousot et al. The ASTRÉE analyzer. In *ESOP*, volume 3444 of *LNCS*, 2005.
10. J. Feret. Static analysis of digital filters. In *ESOP*, volume 2986 of *LNCS*, 2004.
11. S. Horwitz, A. J. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.
12. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, volume 3444 of *LNCS*, 2005.
13. S. Mertens. The easiest hard problem: Number partitioning. In A.G. Percus, G. Istrate, and C. Moore, editors, *Computational Complexity and Statistical Physics*, page 8. Oxford University Press, 2004.
14. A. Miné. The octagon abstract domain. In *AST 2001*, IEEE, 2001.
15. A. Miné. A few graph-based relational numerical abstract domains. In *SAS*, volume 2477 of *LNCS*. Springer, 2002.
16. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, volume 2986 of *LNCS*. Springer, 2004.
17. A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, 2004.
18. A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO*, volume 2053 of *LNCS*, 2001.
19. A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *PLDI*, 2004.