Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software

Bruno Blanchet[1], Patrick Cousot[1], Radhia Cousot[2], Jérôme Feret[1], Laurent Mauborgne[1], Antoine Miné[1], David Monniaux[1], Xavier Rival[1]

[1] CNRS & École normale supérieure, 75005 Paris, France
[2] CNRS & École polytechnique, 91128 Palaiseau cedex, France

Abstract. We report on a successful preliminary experience in the design and implementation of a special-purpose Abstract Interpretation based static program analyzer for the verification of safety critical embedded real-time software. The analyzer is both precise (zero false alarm in the considered experiment) and efficient (less than one minute of analysis for 10,000 lines of code). Even if it is based on a simple interval analysis, many features have been added to obtain the desired precision: expansion of small arrays, widening with several thresholds, loop unrolling, trace partitioning, relations between loop counters and other variables. The efficiency of the tool mainly comes from a clever representation of abstract environments based on balanced binary search trees.

Dedicated to Neil Jones, for his 60th birthday.

1 Introduction

1.1 General-Purpose Static Program Analyzers

The objective of static program analysis is to automatically determine run-time properties, statically, that is, at compile-time. This problem is in general undecidable, so static program analysis relies on approximations as formalized by Abstract Interpretation [8,9]. For example, typable programs do not go wrong but untypable programs do not all go wrong.

In many contexts, such as program transformation, the uncertainty induced by the approximation is acceptable. For example, interval analysis can be used to eliminate useless array bound checks at run-time [7]. The selection rate (i.e. the proportion of potential alarms that are definitively solved either positively or negatively) is often between 80 and 95%, so the optimization is worthwhile. Moreover, it is correct since the remaining 5 to 20% cases for which tests cannot be eliminated at compile-time will be checked at run-time. Sometimes, static program analysis can discover definite errors at compile-time (e.g. uninitialized variables), which is useful for debugging. The objectives of such *general-purpose static program analyzers* are:

1. to fully handle a general purpose programming language (such as Ada or C, including the standard libraries);
2. to require no user-provided specification/annotation (except maybe light ones such as, e.g., ranges of input variables or stubs for library functions the source of which is not available);
3. to be precise enough to provide interesting information for most programs (e.g. information pertinent to static program manipulation or debugging);
4. to be efficient enough to handle very large programs (from a cost of a few megabytes and minutes to gigabytes and hours when dealing with hundreds of thousands of source code lines).

Such general-purpose static program analyzers are very difficult to design because of the complexity of modern programming languages and systems. Some are commercially available and have had successes in repairing failures or preventing them in the early development of programs. Since the coverage is 100%, the false alarms can be handled, e.g., by classical testing methods, thus reducing the need for actual test of absence of run-time error by an economically significant factor of 80 to 95%.

1.2 Program Verification

In the context of safety critical real-time software as found, e.g., in the transportation industry, run-time checking of errors may not be acceptable at all. Hence, the debugging cost is very high and significantly higher than the software development cost itself. In this particular industrial context where correctness is required by safety and criticality, rigorous formal methods should be directly applicable and economically affordable.

Deductive Methods. In practice, deductive methods (see, for example, [1,17]) are hard to apply when lacking a formal specification and when the program size is very large. Indeed, the cost for developing the specification and the proof, even with a proof assistant or a theorem prover, is in general much higher than the cost for developing and testing of the program itself (figures of 600 person-years for $80,000$ lines of C code have been reported). Only critical parts of the software can be checked formally and errors appear elsewhere (e.g. at interfaces). Moreover, for embedded software with a lifetime of ten to twenty years, both the program and its proof have to be maintained over that long period of time.

Software Model Checking. Software model checking (see, for example, [12]) is also hard to apply when lacking a formal specification and when the program size is very large. This is because the cost of developing both the specification and the model of the program can also be very large. Problems such as the difficulty to provide sensible temporal specifications or the state explosion are well-known. On one hand, if the model is not proved correct, then the program correctness check is not rigorous and mostly amounts to debugging. On the other hand, the model can be proved correct, either manually or using deductive methods, but it is then logically equivalent to the correctness proof of the original program [6] and consequently requires an immense effort. Moreover, the program model and its correctness proof have to be maintained with the program itself, which may be a significant additional cost. Finally, abstraction is often required; in this case, software model checking essentially boils down to static program analysis.

Static Program Analysis. Static program analyzers prove program properties by effectively computing an abstract semantics of programs expressed in fixpoint or constraint form. The collected abstract information can then be used as a basis for program manipulation, checking, or partial or total verification. Depending on the considered classes of programs and abstract properties, several types of static program analyzers can be designed, all founded on the Abstract Interpretation theory [5].

1.3 On the Use of General-Purpose Static Program Analyzers

General-purpose static program analyzers require no human intervention and, hence, are very cheap to use, in particular during the initial testing phase and, later, during the program maintenance and modification. However, even for simple specifications, they are hardly useful for formal verification because of their execution time (which is required to get precision but may prevent a routine compiler-like utilization during the initial program development process) and the residual false alarms (excluding full verification). A selection rate of 95%—which is very high when considering a general-purpose static program analyzer—means that a significant part of the code still needs to be inspected manually, which remains a prohibitive cost or, if bypassed, is incompatible with severe safety requirements. Moreover, if the analysis time is of several hours, if not days, the refinement of the analysis, e.g., by inserting correctness assertions, is a very slow process and will not, in general, eliminate all false alarms because of the inherent approximations wired in the analyzer.

1.4 Special-Purpose Static Program Analyzers

Because of undecidability, automaticity, and efficiency requirements, the absence of false alarm can only be achieved by restricting both the considered classes of specifications and programs. This leads to the idea of *special-purpose static program analyzers*. Their objectives are:

1. to handle a restricted family of programs (usually not using the full complexity of modern programming languages);
2. to handle a restricted class of general-purpose specifications (without user intervention except, maybe, light ones such as, e.g., ranges of input or volatile variables);

3. to be precise enough to eliminate all false alarms (maybe through a redesign or, better, a convenient parameterization of the analyzer by a trained end-user that does not need to be a static analysis or Abstract Interpretation specialist);
4. to be efficient enough to handle very large programs (a cost of a few megabytes and minutes for hundreds of thousands of source code lines).

By handling a family of programs and not only a single program or model of the program, we cope with the evolution over years and the economic cost-effectiveness problems. By restricting the considered class of specifications and, more precisely, considering general-purpose requirements (such as absence of run-time error or unexpected interrupt), we avoid the costly development of specific specifications and can apply the analyzer on legacy software (e.g. decades-old applications the initial specification of which, if any, has not been maintained over time). Moreover, the trade-off between analysis precision and cost can be carefully balanced by the choice of appropriate and reusable abstractions.

1.5 Report on a First Experience on the Design of a Special-Purpose Static Program Analyzer

In this paper, we report on a first experience on the design of a special-purpose static program analyzer. The considered class of software is critical real-time synchronous embedded software written in a subset of C. The considered class of specifications is that of absence of run-time error. This experience report explains the crucial design decisions and dead-ends that lead from a too imprecise and too slow initial implementation of a general-purpose static program analyzer to a completely rewritten, very fast, and extremely precise special-purpose static program analyzer. By providing details on the design and implementation of this static analyzer as well as on its precision (absence of false alarm), execution time, and memory usage, we prove that the approach is technically and economically viable.

2 The Special Purpose of our Analyzer

Because of its critical aspect, the class of software analyzed in this first experience was developed through a rigorous process. In this process, the software is first described using schemata. These schemata are automatically translated into a C code using handwritten macros which compute basic functionalities. This C code, organized in many source files, is the input of the analyzer.

Because of the real-time aspect of the application, the global structure of the software consists in an initialization phase followed by a big global synchronized loop. Because of this structure, nearly all variables in the program depend on each other.

Because of the way the C code is generated, the program contains a lot of global and static variables, roughly linear in the length of the code (about

$1,300$ for $10,000$ LOCs[3]). It follows that memory space cannot be saved by a preliminary analysis of the locality of the program data.

2.1 Restrictions to C Followed by the Software Class

Fortunately, the strong requirements enforced by the development of critical software imply that some difficult aspects of the C language are not used in this class of software. First, there are neither **goto**s nor recursive function calls. The data structures are quite simple: the software does not manipulate recursive data structures, and the only pointers are statically allocated arrays (no dynamic memory allocation). There is no pointer arithmetic except the basic array operations. Because the code does not contain strings either, alias information is trivial.

2.2 Specification to be Verified

The analyzer has to prove the following:

- absence of out-of-bound array indexes;
- logical correctness of integer and floating-point arithmetic operations (essentially, absence of overflow, of division by 0).

So, the analysis consists in over-approximating the set of reachable states.

3 Program Concrete Semantics

The program concrete semantics is a mathematical formalization of the actual execution of the program. A precise definition is necessary to define and prove the soundness of the verifier, checker, or analyzer. For example, in static analysis, the analyzer effectively computes an abstract semantics which is a safe approximation of this concrete semantics. So, the rigorous definition of the program concrete semantics is mandatory for all formal methods.

In practice, the concrete semantics is defined by:

- the ISO/IEC 9899 standard for the C programming language [14] as well as the ANSI/IEEE 754 standard for floating-point arithmetic [2];
- the compiler and machine implementation of these standards;
- the end-user expectations.

Each semantics is a refinement of the previous one where some non-determinism is resolved.

[3] The number of lines of code (LOCs) is counted with the Unix$^{\mathrm{TM}}$ command `wc -l` after stripping comments out and macro preprocessing. Then, the abstractions we consider essentially conserve all variables and LOCs, see Sec. 8.

3.1 The C Standard Semantics

The C standard semantics is often nondeterministic in order to account for different implementations. Here are three examples:

unspecified behaviors are behaviors where the standard provides two or more possibilities and imposes no further requirement on which should be chosen. An example of unspecified behavior is the order in which the arguments to a function are evaluated;

undefined behaviors correspond to unportable or erroneous program constructs on which no requirement is imposed. An example of undefined behavior is the behavior on integer overflow;

implementation-defined behaviors are unspecified behaviors where each implementation documents how the choice is made. An example of implementation-defined behavior is the number of bits in an **int** or the propagation of the high-order bit when a signed integer is right-shifted.

A static analyzer based on the standard C semantics would be sound/correct for all possible conformant compilers. The approach seems unrealistic since the worst-case assumptions to be made by the concrete semantics are not always easy to imagine in case no requirement is imposed and will anyway lead to huge losses of precision, hence, to unacceptably many false alarms. For instance, the C standard does not impose the sizes and precisions of the various arithmetic types, only some minimal sizes, thus the analysis would be very imprecise in case of suspected overflows.

3.2 The Implementation Semantics

A correct compiler for a given machine will implement a refinement of the standard semantics by choosing among the allowed behaviors during the execution. To achieve precision, the design of a static analyzer will have to take into account behaviors which are unspecified (or even undefined) in the norm but are perfectly predictable for a given compiler and a given machine (provided the machine is predictable). For example:

unspecified behaviors: the arguments to a function are evaluated left to right;

undefined behaviors: integer overflow is impossible because of modulo arithmetic (division and modulo by zero are the only possible integer run-time errors);

implementation-defined behaviors: there are 32 bits in an **int** and the high-order bit is copied when right-shifting a signed integer.

3.3 The End-User Semantics

The end-user may have in mind a semantics which is a refinement of the implementation semantics. Examples are:

initialization to zero which is to be performed by the system before launching the program (whereas the C standard requires this for static variables only);

volatile variables for using interface hardware can be assigned a range, so that reads from these variables always return a value in the specified range;

integer arithmetic computations which are subject to overflow (since they represent integer bounded quantities for which modulo arithmetic is meaningless) or not (such as shift operations to extract fields of words on interface hardware for which overflow is meaningless).

For meaningful analysis results, one has to distinguish between cases where the execution of the program proceeds or not after hitting undefined or implementation-defined behaviors. In the former case, we take into account the implementation-defined execution; in the latter, we consider the trace to be interrupted. Let us take two examples:

- In a context where $\mathtt{x} \in [0, \mathtt{maxint}]$ is an unsigned integer variable, the analysis of the assignment $\mathtt{y} := 1/\mathtt{x}$ will signal a logical error in case $\mathtt{x} = 0$. In the considered implementation, integer divisions by zero always generate a system exception that aborts the normal execution of the program. Hence we consider that the execution can only go on when there is no run-time error with $\mathtt{y} \in [1/\mathtt{maxint}, 1]$. In that case, the implementation and intended concrete semantics do coincide;
- In a context where $\mathtt{x} \in [0, \mathtt{maxint}]$ is an integer variable, the analysis of the assignment $\mathtt{y} := \mathtt{x} + 1$ will signal a logical error in case $\mathtt{x} = \mathtt{maxint}$. Since the implementation does not signal any error, the end-user can consider the logical error as a simple warning and choose to go on according to several possible concrete semantics:

  Implementation concrete semantics: from an implementation point of view, the execution goes on in all cases $\mathtt{x} \in [0, \mathtt{maxint}]$, that is with $\mathtt{y} \in \{-\mathtt{maxint} - 1\} \cup [1, \mathtt{maxint}]$ (since with modulo arithmetic, the implementation does not signal the potential logical error).

  This choice may cause the later analysis to be polluted by the logically infeasible cases ($\mathtt{y} = -\mathtt{maxint} - 1$ in our example). Such a behavior is in fact intentional in certain parts of the program (such as to extract fields of unsigned integers to select volatile quantities provided by the hardware which is logically correct with wrapping);

  Logical concrete semantics: from a purely logical point of view, the execution goes on with error-free cases $\mathtt{x} \in [0, \mathtt{maxint} - 1]$, that is with $\mathtt{y} \in [1, \mathtt{maxint}]$ (as if the implementation had signaled the logical error).

  One can think that this point of view would be implementation correct for error-free programs (assuming programs will not be run until all logical warnings are shown to be impossible). This is not the case if the programmer makes some explicit use of the hardware characteristics (such as modulo arithmetic). For example, the correctness of some program constructs (such as field extraction) relies on the absence of overflow in modulo arithmetic and, so, ignoring this fact would lead to the erroneous conclusion that the subsequent program points are unreachable!

> Because some constructs (such as signed integer arithmetic) require to take a logical concrete semantics and others (such as field extraction from unsigned integers) require to explicitly rely on the implementation concrete semantics, the analyzer has to be parameterized so as to leave the final choice to the end-user (who can indicate to the analyzer which semantics is intended through a configuration file, for example on a per-type and per-operator basis).

4 Preliminary Manipulations of the Program

To reduce the later cost of the static analysis, we perform a few preliminary manipulations of the program. Since the program uses C macros and the semantics of macros in C is not always clear, macros are expanded before the analysis, so the analyzed program is the pre-processed program. Then, all input files are gathered into a single source file. Because the program is automatically generated, it has numerous symbolic constants, so, a classical constant propagation is performed. Note that floating-point constants must be evaluated with the same rounding mode as at run-time, in general to the nearest, whereas during the analysis, interval operations will always be over-approximated: we consider the worst-case assumptions for the rounding mode, to make sure that the computed interval is larger than the actual one. The constant propagation is extended to the partial evaluation [13] of constant expressions including, in particular, accesses to constant arrays with a constant index. This was particularly useful for arrays containing indirections to hardware addresses for interfaces.

Other manipulations can be specified in a configuration file. We can specify volatile variables. Volatile variables should in fact be mentioned as such in the source file; however, they are sometimes omitted from the source because the compiler does not optimize memory accesses, so volatile declarations have no effect on the compilation. We can also specify for volatile variables a range that represents, for instance, the value of sensors. The analyzer then makes the assumption that all accesses to these variables return a value in the indicated range. The first manipulation pass inserts the range as a special kind of initializer for the considered variables. The resulting syntax is then an extension of the C syntax that is taken as input by the other phases of the analyzer.

The user can also declare functions to be ignored. These functions are then given an empty code. (If they were not already defined, then they are defined with an empty code. If they were already defined, then their code is removed.) This declaration has two purposes. The first one is to give a code to built-in system calls that do not influence the rest of the behavior of the program. The second one is to help finding the origin of errors detected by the analyzer: ignoring declarations can be used to simplify the program, and see if the analyzer still finds the error in the simplified program. This usage was not intended at the beginning, but it proved useful in practice.

5 Structure of the Analyzer

To over-approximate the reachable states of a well-structured program, the analyzer proceeds by induction on the program syntax. Since the number of global variables is large (about $1,300$ and $1,800$ after array expansion, see Sec. 6.3) and the program is large (about $10,000$ LOCs), an abstract environment cannot be maintained at each program point as usual in toy/pedagogical analyzers [4]. Instead, the analysis proceeds by induction on the syntax with one current abstract environment only. Loops are handled by local fixpoint computations with widenings and narrowings. During this iteration, an abstract environment is maintained at the head of the loop only. So, the number of environments which has to be maintained is of the order of the level of nesting of loops. After the fixpoint is reached, an additional iteration is performed so that all runtime errors can be detected even if environments are not recorded at each program point. Nevertheless, special precaution must be taken for implementing these environments efficiently, as discussed below in Sec. 6.2. Moreover, there are only few user-defined procedures and they are not recursive, so they can be handled in a polyvariant way (equivalent to a call by copy).

6 Special-Purpose Abstract Domains

6.1 Iterative Construction of the Analyzer

We started with classical analyzes (e.g. interval analysis for integer and floating-point arithmetics, handling of arrays by abstraction into a single element, etc.), which, as expected from a preliminary use of a commercial general-purpose analyzer by the end-user, lead to unacceptable analysis times and too many false alarms.

The development of the analyzer then followed cycles of iterative refinements. A version of the analyzer is run, outputting an abstract execution trace as well as a list of the alarms. Each alarm (or, rather, group of related alarms) is then manually inspected with the help of the abstract trace. The goal is to differentiate between legitimate alarms, coming for instance from insufficient specification of the inputs of the program, and false alarms arising from lack of analysis precision. When a lack of precision is detected, its causes must be probed. Once the cause of the loss of precision is understood, refinements for the analysis may be proposed.

Various refinements of the analyzer were related to memory and time efficiency which were improved either by redesign of data structures and algorithms or by selecting coarser abstract domains.

These refinements are reported below. Some are specific to the considered class of programs, but others are of general interest to many analyzers—such as the use of functional maps as discussed in the following section Sec. 6.2.

6.2 Efficient Implementation of Abstract Environments through Maps

One of the simplest abstract domains is the domain of intervals [7]: an abstract environment maps each integer or real variable $\mathtt{x} \in V$ to an interval $\mathtt{X} \in I$. The abstract semantics of arithmetic operations are then ordinary interval arithmetic. The least upper bound and widening operations operate point-wise (i.e. for each variable). More generally, we shall consider the case where the abstract environment is a mapping from V to any abstract domain I.

A naive implementation of this abstract domain represents abstract environments as arrays of elements of I. If destructive updates are allowed in abstract transfer functions (i.e. the environment representing the state before the operation can be discarded), the abstract functions corresponding to assignments are easy to implement; if not, a new array has to be allocated.

For all its simplicity, this approach suffers from two drawbacks:

- it requires many array allocations; this can strain the memory allocation system, although most of the allocated data is short-lived;
- more annoyingly, its complexity on the class of programs we are considering is prohibitive: the cost of a least upper bound operation, which is performed for each test construct in the program, is linear in the number of variables; on the programs we consider, the number of static variables is linear in the length of the program, thus leading to a quadratic cost.

A closer look at the program shows that most least upper bound operations are performed between very similar environments; that is, environments that differ in a small number of variables, corresponding to the updates done in the two branches of a test construct. This suggests a system that somehow represents the similarities between environments and optimizes the least upper bound operation between similar environments.

We decided to implement the mappings from V to I as balanced binary search trees that contain, at each node, the name of a variable and its abstract value. This implementation is provided by the functional map module Map of Objective Caml [15]. The access time to the environment for reading or updating an element is logarithmic with respect to the number of variables (whereas arrays, for instance, would yield a constant time access).

A salient point is that all the operations are then performed fully functionally (no side effect) with a large sharing between the data structures describing different related environments. The functional nature allows for straightforward programming in the analyzer—no need to keep track of data structures that may or may not be overwritten—and the sharing keeps the memory usage low.

Functional maps also provide a very efficient computation of binary operations between similar environments, when the operation $o : I \times I \to I$ satisfies $\forall x \in I,\ o(x,x) = x$. This is true in particular for the least upper bound and the widening. More precisely, we added the function map2 defined as follows: if f_1 and $f_2 : V \to I$ and $o : I \times I \to I$ satisfies $\forall x \in I,\ o(x,x) = x$, then $\mathsf{map2}(o, f_1, f_2) = x \mapsto o(f_1(x), f_2(x))$. This function is implemented by walking recursively both trees representing f_1 and f_2; when f_1 and f_2 share a common subtree, the result is the same subtree, which can be returned immediately. The function map2 has to traverse only the nodes that differ between f_1 and f_2—which correspond to paths from the root to the modified variables. This strategy leads to a time complexity $\mathcal{O}(m \log n)$ where m the number of *modified* variables between f_1 and f_2, and n is the total number of variables in the environment ($\log n$ is the maximum length of a path from the root to a variable). When only a few variables in the functional map have different values (for example, when merging two environments after the end of a test), a very large part of the computation can be optimized away thanks to this technique.

In conclusion, functional maps implemented using balanced binary search trees decrease tremendously the practical complexity of the analyzer.

6.3 Expansion of Small Arrays

When considering an array variable, one can simply *expand* it in the environment, that is to say, consider one abstract element in I for each index in the array. One can also choose to *smash* the array elements into one abstract element that represent all the possible values for all indices.

When dealing with large arrays, smashing them results in a smaller memory consumption. Transfer functions on smashed arrays are also much more efficient. For example, the assignment `tab[i] :=` **exp** with `i` $\in [0, 99]$ leads to 100 abstract assignments if `tab` is expanded, and only one if `tab` is smashed.

Expanded arrays, however, are much more precise than smashed ones. Not only they can represent heterogeneous arrays—such as arrays of non-zero float elements followed by a zero element that marks the end of the array—but they result in less *weak updates*[4]. For example, if the two-elements array `tab` is initialized to zero, and then assigned by `tab[0] := 1; tab[1] := 1`, smashing the array will result in weak updates that will conclude that `tab[i]` $\in [0, 1]$. The precision gain of expanded arrays is particularly interesting when combined with semantics loop unrolling (see Sec. 6.5).

To address the precision/cost trade-off of smashed vs. expanded arrays, the analyzer is parameterized so that the end-user can specify in a configuration file which arrays should be expanded either by providing an array size bound (arrays of size smaller than the bound are expanded) and/or by enumerating them nominatively.

6.4 Staged Widenings with Thresholds

The analyzer is parameterized by a configuration file allowing the user to specify refinements of the abstract domains which are used by the analyzer. An example is the *staged widenings with thresholds.*

[4] A weak update denotes an assignment where some variables may or may not be updated, either because the assigned variable is not uniquely determined by the analyzer, or because the assigned variable is smashed with some others.

The classical widening on intervals is $[a, b] \bigtriangledown [c, d] = [(c < a ? -\infty : a), (d > b ? +\infty : b)]$[5] [8]. It is known since a long time that interval analysis with this widening is less precise than sign analysis since, e.g., $[2, +\infty] \bigtriangledown [1, +\infty] = [-\infty, +\infty]$ whereas sign analysis would lead to $[0, +\infty]$ (or $[1, +\infty]$ depending on the chosen abstract domain [9]). So, most widenings on intervals use additional thresholds, such as -1, 0 and +1. The analyzer is parameterized by a configuration file allowing the user to specify thresholds of his choice.

The thresholds can be chosen by understanding the origin of the loss of precision. A classical example is (n is a given integer constant):

```
int x;
x := 0;
while x <> $n$ do
  x := x + 1;
end while
```

(whereas writing x $<$ n would allow the narrowing to capture the bound n [7,8]). A more subtle example is:

```
volatile boolean b;
int x;
x := 0;
while true do
  if b then
    x := x + 1;
    if x > $n$ then
      x := 0;
    end if
  end if
end while
```

In both cases, a widening at the loop head will extrapolate to $+\infty$ and the later narrowing will not recover the constant n bound within the loop body. This was surprising, since this worked well for the following piece of code:

```
int x;
x := 0;
while true do
  x := x + 1;
  if x > $n$ then
    x := 0;
  end if
end while
```

In the first case however, the test:

```
if x > $n$ then
  x := 0;
end if
```

[5] (true?$a : b$) $= a$ whereas (false?$a : b$) $= b$.

may not be run at each iteration so once the analyzer over-estimates the range of `x`, it cannot regain precision even with this test. A solution would be to ask for a user hint in the form of an assertion $\mathtt{x} \leq n$. An equivalent hinting strategy is to add the constant n as a widening threshold. In both cases, the widening will not lead to any loss of precision. Another threshold idea, depending on the program, is to add arithmetic, geometric or exponential progressions known to appear in the course of the program computations.

6.5 Semantic Loop Unrolling

Although the software we analyze always starts with some initialization code before the main loop, there is still some initialization which is performed during the first iteration of that main loop. The mechanism used in the software is a global boolean variable which is true when the code is in the first iteration of the main loop.

If we try to compute an invariant at the head of the loop and the domain is not relational, then this boolean can contain both the values true and false and we cannot distinguish between the first iteration and the other ones. To solve this problem, we applied semantic loop unrolling.

Semantic loop unrolling consists, given an unrolling factor n, in computing the invariants I_0 which is the set of possible values before the loop, then I_k, $1 \leq k < n$ the set of possible values after exactly k iterations, and finally J_n the set of possible values after n or more iterations. Then, we merge $I_0, \ldots, I_{n-1}, J_n$ in order to get the invariant at the end of the loop. Another point of view is to analyze the loop **while** B **do** C as **if** B **then** (C; **if** B **then** (... **if** B **then** (C; (**while** B **do** C))...)). Such a technique is more precise than the classical analysis of while loops when the abstract transfer functions are not fully distributive or when we use widenings.

In our case, the first iteration of the main loop is an initialization phase that behaves very differently than subsequent iterations. Thus, by setting $n = 1$, the invariant J_n is computed taking into account initialized values only so we can get a more precise result and even suppress some false alarms.

6.6 Trace Partitioning

The reason why semantic loop unrolling is more precise is that, for each loop unrolling, a new set of values is approximated. So, instead of having one set of values, we have a collection of sets of values which is more precise than their union because we cannot represent this union exactly in the abstract domain. We could be even more precise if we did not merge the collection of sets of values at the end of the loop but later.

Consider, for example, the following algorithm which computes a linear interpolation:

```
t = {-10, -10, 0, 10, 10};
c = {0, 2, 2, 0};
```

```
d = {-20, -20, 0, 20};
i := 0;
while i < 3 and x ≥ t[i+1] do
  i := i+1;
end while
r := (x - t[i]) × c[i] + d[i];
```

The resulting variable r ranges in $[-20, 20]$, but if we perform a standard interval analysis the result will be $[\min(-20, 2x^{-}-40), \max(20, 2x^{+}+40)]$ (where x is in $[x^{-}, x^{+}]$). This information is not precise enough because interval analysis is not distributive. It is the case even with semantic loop unrolling because, when we arrive at the statement where r is computed, all unrollings are merged and we have lost the relationship between i and x.

Trace partitioning consists in delaying the usual mergings which might occur in the transfer functions. Such mergings happen at the end of the two branches of an **if**, or at the end of a **while** loop when there is semantic loop unrolling. Control-based trace partitioning was first introduced by [11]. Trace partitioning is more precise for non-distributive abstract domains but can be very expensive as it multiplies the number of environments by 2 for each **if** that is partitioned and by k for each loop unrolled k times. And this is even worse in the case of trace partitioning inside a partitioned loop.

So, we improved [11] techniques to allow the partition to be temporary: the merging is not delayed forever but up to a parameterizable point. It worked well to merge partitions created inside a function just before return points, and partitions created *inside* a loop at the end of the loop. This notion of merging allowed the use of trace partitioning even inside the non-unrolled part of loops. In practice, this technique seems to be a good alternative to the more complex classical reduced cardinal power of [9].

6.7 Relation between Variables and Loop Counters

As explained in the beginning of the section, non-relational domains, such as the interval domain, can be efficiently implemented. However, non-relational invariants are sometime not sufficient, even for the purpose of bounding variable values. Consider the following loop:

```
volatile boolean b;
i := 0;
x := 0;
while i < 100 do
  x := x + 1;
  if b then
    x := 0;
  end if
  i := i + 1;
end while
```

In order to discover that $\mathtt{x} < 100$, one must be able to discover the invariant relation $\mathtt{x} \leq \mathtt{i}$. Staged widenings are ineffective here because x is never compared explicitly to 100. Switching to fully relational abstract domains (such as polyhedra, or even linear equality) is clearly impossible due to the tremendous amount of global live variables in our application (in fact, even non-relational domains would be too costly without the representation technique of Sec. 6.2).

Our solution is to consider only relations between a variable and a loop counter δ (either explicit in a **for** loop or implicit in a **while** loop). We denote by Δ the interval of the counter δ (Δ is either determined by the analysis or specified by the end-user in the configuration file, e.g., because the application is designed to run for a certain maximum amount of time). Instead of mapping each variable x to an interval X, our enhanced invariants map each variable x to three intervals: X, $\mathtt{X}^+$ and $\mathtt{X}^-$ which are, respectively, the possible values for x, for $\mathtt{x}+\delta$, and for $\mathtt{x}-\delta$. When too much information on the interval X is lost (after a widening, for example), $\mathtt{X}^+$, $\mathtt{X}^-$, and Δ are used to recover some information using a so-called *reduction operator* (see Sec. 6.8 below), which replaces the interval X by the interval $\mathtt{X} \cap (\mathtt{X}^+ - \Delta) \cap (\mathtt{X}^- + \Delta)$. This is a simple abstract way of capturing the evolution of the value of the variables over time (abstracted by the loop counter).

From a practical point of view, this domain is implemented as a non-relational domain using the data-structures of Sec. 6.2. It greatly increases the precision of the analysis for a small speed and memory overhead factor.

6.8 Reduction and its Interaction with Widening

In order to take into account the relations between program variables and the loop counters, we use a reduction operator ρ which is a conservative endomorphism (i.e. such that $\gamma(d) \subseteq \gamma(\rho(d))$). The way this reduction is used has a great impact, not only on accuracy, but also on complexity: on the first hand it is crucial to make the reduction before computing some abstract transfer functions (testing a guard for instance) to gain some precision; on the other hand, the cost of the reduction must not exceed the cost of the abstract transfer function itself.

Our choice was to perform reductions on the fly inside each abstract primitive. This allows us to focus the reduction on the program variables which need to be reduced. It is very simple for unary operators. As for the binary operators, we detect which part of the result must be reduced thanks to the functional map implementation, which leads to a sub-linear implementation of the reduction—which coincides with the amortized cost of abstract transfer functions.

The main problem with this approach is that the reduction may destroy the extrapolation constructed by widening[6]. Usually, the reduction operator cannot be applied directly to the results of a widening. Some solutions already existed, but they were not compatible with our requirement of having a sub-linear implementation of the reduction.

[6] The reader may have a look at the Fig. 3 of [16] to have an illustration of this problem in the context of a relational domain.

To solve this problem, we require non-standard conditions on the reduction: we especially require that there are no cyclic propagation of information between abstract variables. We replace the interval $\mathtt{X}$ by the interval $\mathtt{X}\cap(\mathtt{X}^{+}-\mathit{\Delta})\cap(\mathtt{X}^{-}+\mathit{\Delta})$, but we do not propagate the other way round (for instance, we replace neither $\mathtt{X}^{+}$ by $\mathtt{X}^{+}\cap(\mathtt{X}+\mathit{\Delta})$, nor $\mathtt{X}^{-}$ by $\mathtt{X}^{-}\cap(\mathtt{X}-\mathit{\Delta})$). This allows extrapolation to be first performed on the intervals $\mathit{\Delta}$, $\mathtt{X}^{+}$, and $\mathtt{X}^{-}$. Once the iterate of the intervals $\mathit{\Delta}$, $\mathtt{X}^{+}$, and $\mathtt{X}^{-}$ have become stable, the extrapolation of the interval $\mathtt{X}$ is not disturbed anymore.

6.9 On the Analysis of Floating-Point Arithmetic

A major difficulty of the analysis of floating-point arithmetic is the rounding errors, both in the analyzed semantics and in the analyzer itself. One has to consider that:

- transfer functions should model floating-point arithmetic, that is to say (according to the IEEE standard [2]), infinite-precision real arithmetic followed by a rounding phase;
- abstract operators should be themselves implemented using floating-point arithmetic (for efficiency, arbitrary precision floating-point, rational, and algebraic arithmetics should be prohibited).

In particular, special care has to be taken since most classical mathematical equalities (associativity, distributivity, etc.) no longer hold when the operations are translated into floating-point; it is necessary to know at every point if the quantities dealt with are lower or upper bounds.

Interval arithmetic is relatively easy. Operations on lower bounds have to be rounded towards $-\infty$, operations on upper bounds towards $+\infty$. A complication is added by the use of `float`—IEEE single precision—variables in the analyzed programs: abstract operations on these should be rounded in IEEE single precision arithmetic.

7 Dead-Ends

The analyzer went through three successive versions because of dead-ends and to allow for experimentation on the adequate abstractions. In this section, we discuss a number of bad initial design decisions which were corrected in the later versions.

Syntax. An initial bad idea was to use a program syntax tailored to the considered class of automatically generated programs. The idea was to syntactically check for potential errors, e.g., in macros. Besides the additional complexity, it was impossible to test the analyzer with simple examples. Finally, expanding the macros and using a standard C grammar with semantic actions to check for local restrictions turned out to be more productive.

Functional Array Representation of the Environment. The first versions of the analyzer used Caml arrays to represent abstract environments. As discussed in Sec. 6.2, the idea that $\mathcal{O}(1)$ access time to abstract values of variables makes non-relational analyzes efficient turned to be erroneous.

Liveness Analysis. Because of the large number of global variables, liveness analysis was thought to be useful to eliminate useless updates in abstract environments represented as arrays. The gain was in fact negligible. Similar ideas using classical data-flow analysis intermediate representations such as use-definition chains, single static assignment, etc. would probably have also been ineffective. The key idea was to use balanced trees as explained in Sec. 6.2.

Open Floating-Point Intervals. The first version of the analyzer used closed and open floating-point intervals. For soundness, the intervals had to be overestimated to take rounding errors into account, as explained in Sec. 6.9, which makes the analysis very complex with no improvement in precision, so, the idea of using open intervals was abandoned.

Relational Floating-Point Domains. Most literature consider only relational domains over fields, such as rationals or reals, and do not address the problem of floating-point. With some care, one could design a sound approximation of real arithmetic using floating-point arithmetic: each computation is rounded such that the result is always enlarged, in order to preserve soundness. Then, each abstract floating-point operator can be implemented as an abstract operator on reals, followed by an abstract rounding that simply adds to the result an interval representing the absolute error—or, more precisely, the *ulp* [10]. However, this crude approach of rounding can cause the abstract element to drift at each iteration, which prevents its stabilization using widenings. No satisfying solution has been found yet to address this problem, as well as the time and memory complexity inherent to relational domains, so, they are not used in our prototype.

Case Analysis. Case analysis is a classical refinement in static analysis. For example [9, Sec. 10.2] illustrates the reduced cardinal power of abstract domains by a case analysis on a boolean variable, the analysis being split on the true and false cases. Implementations for several variables can be based on BDDs. The same way abstract values can be split according to several concrete values of the variables (such as intervals into sub-intervals). This turned out to be ineffective since the costs can explode exponentially as more splittings are introduced to gain in precision. So, case analysis was ultimately replaced by trace partitioning, as discussed in Sec. 6.6.

On Prototyping. The first versions of the analyzer can be understood as initial prototypes to help decide on the abstractions to be used. The complete rewriting of the successive versions by different persons avoided the accumulation of levels, corrections, translations which over time can make large programs tangled and inefficient.

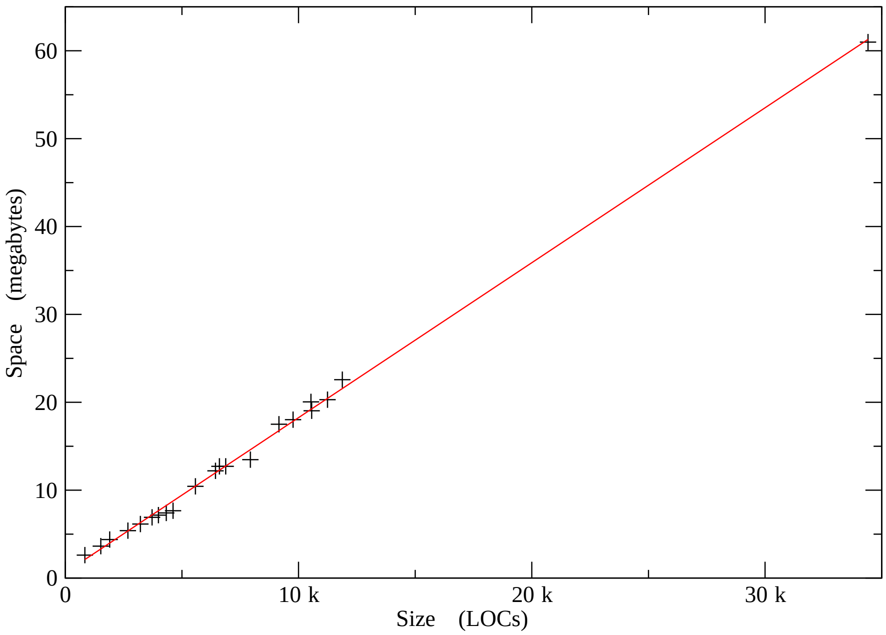

Fig. 1. Memory usage as a function of the subject program size.

8 Performances

The whole problem of static analysis is to find the right cost-performance balance. In program verification, the precision is fixed to zero (false-)alarm. When the zero alarm precision problem is solved, it remains to estimate the performance of the analysis. To estimate memory and timing performances, we made various analysis experiments with slices of the program as well as the synchronous product of the program several times with itself.

The memory used to store an abstract environment grows linearly with the number of variables, which is itself proportional, for the considered applications, to the size of the program itself ℓ. Due to nested loops, loop unrolling, and trace partitioning, the analyzer may need to store several abstract environments during the analysis. However, thanks to the use of functional maps, a huge part of these environment is shared, thus reducing the memory consumption. We have found experimentally (Fig. 1[7]) that the peak memory consumption of the analyzer is indead $\mathcal{O}(\ell)$. For our application, it is of the order of a few megabytes, which is not a problem for modern computers.

[7] The best curve fitting [3] with formula $y = a + bx$ and a tolerance of 10^{-6} yields $a = 0.623994$ and $b = 0.00176291$ with association gauged by Chi-square: 6.82461, Correlation coefficient: 0.951324, Regression Mean Square (RMS) per cent error: 0.0721343 and Theil Uncertainty (U) coefficient: 0.0309627.

Thanks to the use of functional maps (Sec. 6.2), the amortized cost of the elementary abstract operations can be estimated to be at most of the order of the time to access abstract values of variables, which is $\mathcal{O}(\ln v)$, where v is the number of program variables. In the programs considered in our experiment, the number of program variables is itself proportional to the number ℓ of LOCs. It follows that the cost of the elementary abstract operations is $\mathcal{O}(\ln \ell)$. A fixpoint iteration sweeps over the whole program. Because the abstract analysis of procedures is semantically equivalent to an expansion (Sec. 5), each iteration step of the fixpoint takes $\mathcal{O}(\ell' \times \ln \ell \times p \times i')$ where ℓ' is the number of LOCs after procedure expansion, p is a bound to the number of abstract environments that need to be handled at each given program point[8], and i' is a bound to the number of inner fixpoint iterations. The fixpoint computation is then of the order $\mathcal{O}(i \times p \times i' \times \ell' \times \ln \ell)$ where i is a bound to the number of iterations.

We now estimate the bounds p, i, and i'. The number p only depends on end-user parameters. The numbers i and i' are at worst $\mathcal{O}(l \times t)$ where t denotes the number of thresholds, but are constant in practice. So, the execution time is expected to be of the order of $\mathcal{O}(\ell' \times \ln \ell)$. Our hypotheses are confirmed experimentally by best curve fitting [3] the analyzer execution time on various experiments. The fitting formula[9] $y = ax$ yields $a = 0.000136364$, as shown in Fig. 2.

The procedure expansion factor giving ℓ' as a function of the program size ℓ has also been determined experimentally, see Fig. 3. The best curve fitting with formula[10] $y = a \times x \times (\ln x)^b$ yields $a = 0.927555$, $b = 0.638504$. This shows that, for the considered family of programs, the polyvariant analysis of procedures (equivalent to a call by copy semantics), which is known to be more precise than the monovariant analysis (where all calls are merged together), has a reasonable cost.

By composition, we get that the execution time of the analyzer is $\mathcal{O}(\ell(\ln \ell)^a)$ where ℓ is the program size. This is confirmed experimentally by curve fitting the analyzer execution time for various experiments. The non-linear fitting formula[11] $y = a + bx + cx(\ln x)^d$ yields $a = 2.2134 \times 10^{-11}$, $b = 5.16024 \times 10^{-08}$, $c = 0.00015309$ and $d = 1.55729$, see Fig. 4.

The memory and time performances of the analyzer, as extrapolated in Fig. 5, show that extreme precision (no alarm in the experiment) is not incompatible with efficiency. Therefore we can expect such specific static analyzers to be routinely usable for absence of run-time errors verification during the program de-

[8] This number only depends on loop unrolling and trace partitioning.

[9] with association gauged by Chi-square: 239.67, Correlation coefficient: 0.941353, RMS per cent error: 0.515156 and Theil U coefficient: 0.0628226 for a tolerance of 10^{-6}.

[10] with association gauged by Chi-square: 7.00541×10^{07}, Correlation coefficient: 0.942645, RMS per cent error: 0.113283 and Theil U coefficient: 0.0464639 at tolerance 10^{-6}.

[11] with association gauged by Chi-square: 40.1064, Correlation coefficient: 0.956011, RMS per cent error: 0.0595795 and Theil U coefficient: 0.0248341 for a tolerance of 10^{-6}.

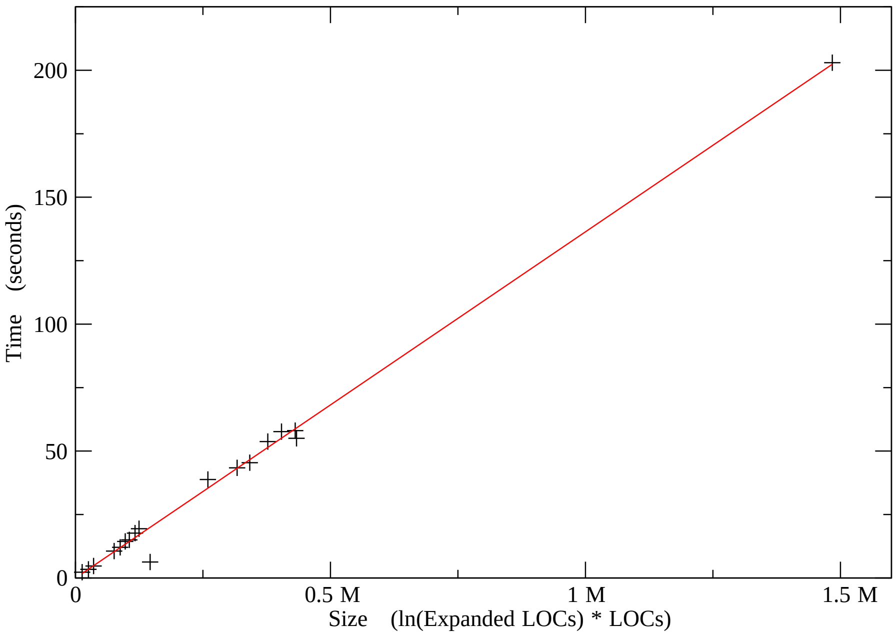

Fig. 2. Execution time as a function of $\ell' \times \ln \ell$, where ℓ' is the expanded subject program size and ℓ its size.

velopment, test, and maintenance processes. Thanks to parameterization, the end-user can easily adjust the analysis to cope with small modifications of the program.

9 Conclusion

When first reading the program, we were somewhat pessimistic on the chances of success of the zero false alarm objective since the numerical computations which, not surprisingly for a non-linear control program, represent up to 80% of the program, looked both rather complex and completely incomprehensible for the neophyte. The fact that the code is mostly machine-generated did not help. Using complex numerical domains (such as polyhedral domains) would have been terribly costly. So, the design criterion was always the simpler, i.e., the most abstract, the better, i.e., the most efficient.

Because of undecidability, human hinting is necessary to analyze programs without false alarm:

- in deductive methods this is done by providing inductive arguments (e.g. invariants) as well as hints for the proof strategy;
- in model-checking, this is done by providing the finite model of the program to be checked;

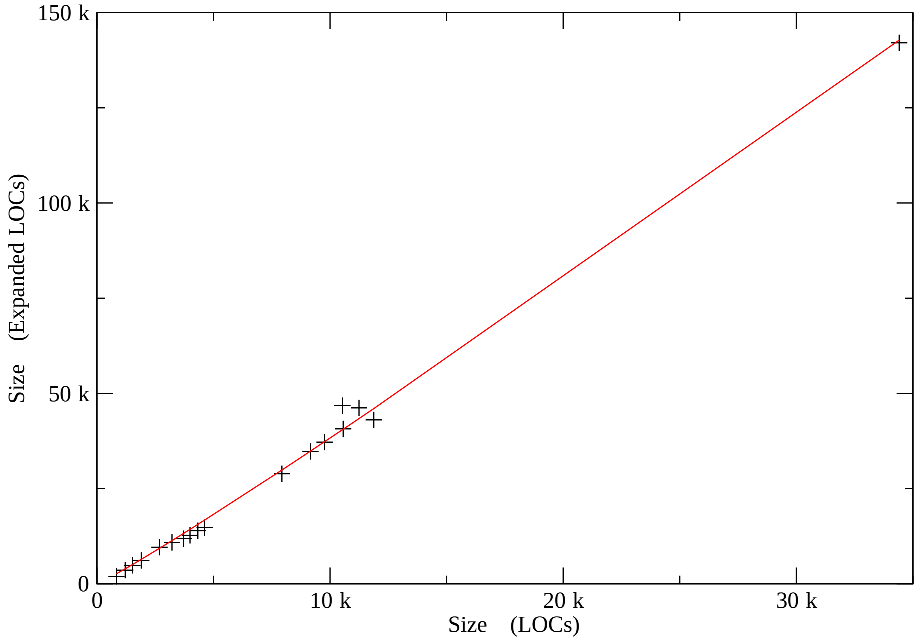

Fig. 3. Procedure-expanded program size ℓ' as a function of the subject program size ℓ.

- in static program analysis, we have shown on a non-trivial example that this can be done by providing hints on the local choice of the abstract domains and widenings.

In all cases, some understanding of the verification technique is necessary. We have the feeling that hints to a parameterized analyzer are much easier to provide than correct invariants or program models. Once specialists have designed the domain-specific static analyzer in a parameterized way, the local refinement process is very easy to specify by end-users who are not specialists in static program analysis.

We have serious doubts on the fact that this refinement process can be fully automated. A counter-example based refinement to handle false alarms would certainly be able only to refine abstract domains, abstract element by abstract element, where these abstract elements directly refer to concrete values. In such an approach, the size of the refined analysis would grow exponentially. Clearly, a non-obvious inference step and a significant rewriting of the analyzer are required to move from examples to abstraction techniques such as partitioning or the relational domain handling the loop counters.

So, our approach to get zero false alarm was to design a special purpose analyzer which is parameterized to allow for casual end-users to choose for the

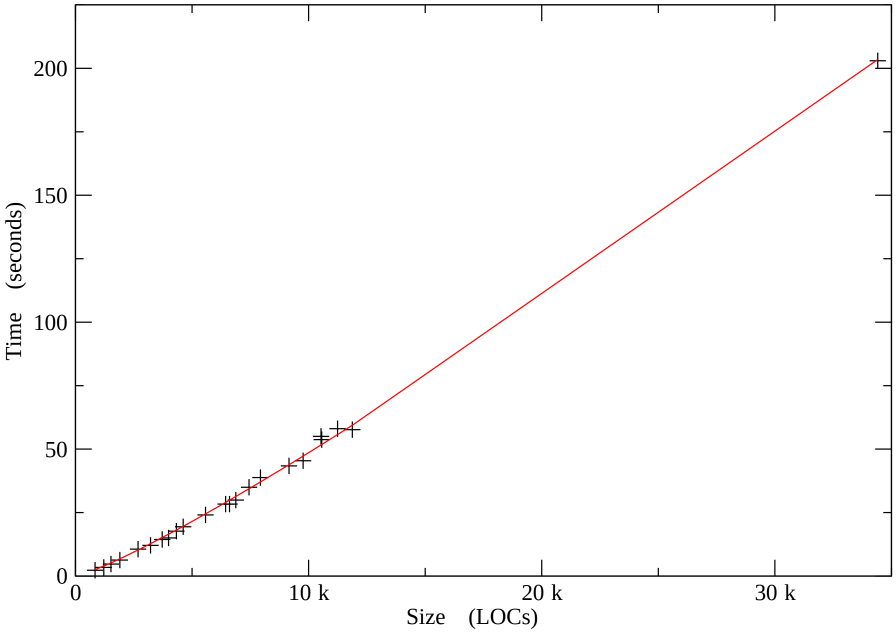

Fig. 4. Execution time as a function of the subject program size.

specific refinements which must be applied for any program in the considered family.

The project is now going on with real-life much larger programs (over $250,000$ LOCs) . The resource usage estimates of Figures 1 and 5 were confirmed. Not surprisingly, false alarms showed up since the floating-point numerical computations in these programs are much involved than in the reported first experimentation. A new refinement cycle is therefore restarted to design appropriate abstract domains which are definitely necessary to reach the zero alarm objective at a low analysis cost.

References

1. J.-R. Abrial. On B. In D. Bert, editor, *Proc. 2nd Int. B Conf. , B'98: Recent Advances in the Development and Use of the B Method*, Montpellier, FR, LNCS 1393, pages 1–8. Springer-Verlag, 22–24 Apr. 1998.
2. American National Standards Institute, Inc. IEEE standard for binary floating-point arithmetic. Technical Report 754-1985, ANSI/IEEE, 1985. `http://grouper.ieee.org/groups/754/`.
3. P. R. Bevington and D. K. Robinson. *Data Reduction and Error Analysis for the Physical Sciences.* McGraw-Hill, 1992.
4. P. Cousot. The Marktoberdorf'98 generic abstract interpreter. `http://www.di.ens.fr/~cousot/Marktoberdorf98.shtml`, Nov. 1998.

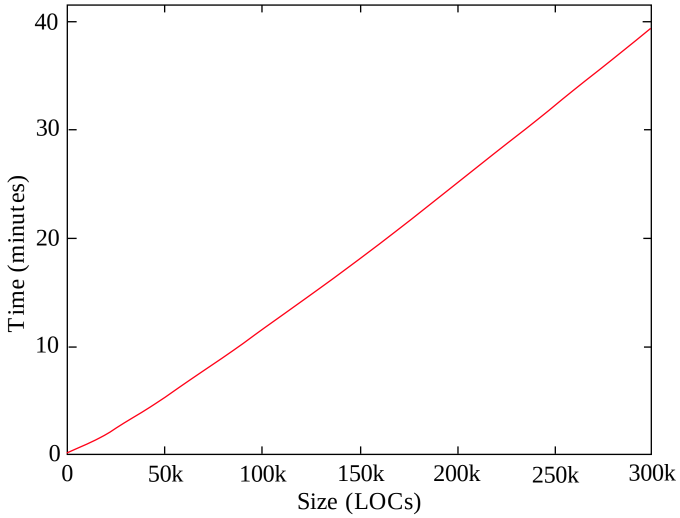

Fig. 5. Extrapolated execution time as a function of the subject program size ($y = 2.2134 \times 10^{-11} + 5.16024 \times 10^{-8} \times x + 0.00015309 \times x \times (\ln x)^{1.55729}$).

5. P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 138–156. Springer-Verlag, 2000.
6. P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B.Y. Choueiry and T. Walsh, editors, *Proc. 4th Int. Symp. SARA '2000*, Horseshoe Bay, TX, US, LNAI 1864, pages 1–25. Springer-Verlag, 26–29 Jul. 2000.
7. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd Int. Symp. on Programming*, pages 106–130. Dunod, 1976.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.
10. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, Mar. 1991.
11. M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In G. Levi, editor, *Proc. 5th Int. Symp. SAS '98*, Pisa, IT, 14–16 Sep. 1998, LNCS 1503, pages 200–214. Springer-Verlag, 1998.
12. G.J. Holzmann. Software analysis and model checking. In E. Brinksma and K.G. Larsen, editors, *Proc. 14th Int. Conf. CAV '2002*, Copenhagen, DK, LNCS 2404, pages 1–16. Springer-Verlag, 27–31 Jul. 2002.
13. N. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Int. Series in Computer Science. Prentice-Hall, June 1993.
14. JTC 1/SC 22. Programming languages — C. Technical report, ISO/IEC 9899:1999, 16 Dec. 1999.

15. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, documentation and user's manual (release 3.04). Technical report, INRIA, Rocquencourt, FR, 10 Dec. 2001. `http://caml.inria.fr/ocaml/`.
16. A. Miné. A new numerical abstract domain based on difference-bound matrices. In 0. Danvy and A. Filinski, editors, *Proc. 2^{nd} Symp. PADO '2001*, Århus, DK, 21–23 May 2001, LNCS 2053, pages 155–172. Springer-Verlag, 2001. `http://www.di.ens.fr/~mine/publi/article-mine-padoII.pdf`.
17. S. Owre, N. Shankar, and D.W.J. Stringer-Calvert. PVS: An experience report. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *PROC Applied Formal Methods - FM-Trends'98, International Workshop on Current Trends in Applied Formal Method*, Boppard, DE, LNCS 1641, pages 338–345. Springer-Verlag, 7–9 Oct. 1999.