# A Static Analyzer for Large Safety-Critical Software

Bruno Blanchet [*§] Patrick Cousot [§] Radhia Cousot [*¶] Jérôme Feret [§]
Laurent Mauborgne [§] Antoine Miné [§] David Monniaux [*§] Xavier Rival [§]

## ABSTRACT

We show that abstract interpretation-based static program analysis can be made efficient and precise enough to formally verify a class of properties for a family of large programs with few or no false alarms. This is achieved by refinement of a general purpose static analyzer and later adaptation to particular programs of the family by the end-user through parameterization. This is applied to the proof of soundness of data manipulation operations at the machine level for periodic synchronous safety critical embedded software.

The main novelties are the design principle of static analyzers by refinement and adaptation through parameterization (Sect. 3 and 8), the symbolic manipulation of expressions to improve the precision of abstract transfer functions (Sect. 7.3), the octagon (Sect. 7.2.2), ellipsoid (Sect. 7.2.3), and boolean relations (Sect. 7.2.4) abstract domains, all with sound handling of rounding errors in floating point computations, widening strategies (with thresholds: Sect. 8.1.2, delayed: Sect. 8.1.3) and the automatic determination of the parameters (parameterized packing, Sect. 8.2).

### Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification—*formal methods, validation, assertion checkers*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification, assertions, invariants*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Denotational semantics, Program analysis*.

### General Terms

Algorithms, Design, Experimentation, Theory, Verification.

## 1. INTRODUCTION

Critical software systems (as found in industrial plants, automotive, and aerospace applications) should never fail. Ensuring that such software does not fail is usually done by testing, which is expensive for complex systems with high reliability requirements, and anyway fails to prove the impossibility of failure. Formal methods, such as model-checking, theorem proving, and static analysis, can help.

The definition of "failure" itself is difficult, in particular in the absence of a formal specification. In this paper, we chose to focus on a particular aspect found in all specifications for critical software, that is, ensuring that the critical software never executes an instruction with "undefined" or "fatal error" behavior, such as out-of-bounds accesses to arrays or improper arithmetic operations (such as overflows or division by zero). Such conditions ensure that the program is written according to its intended semantics, for example the critical system will never abort its execution. These correctness conditions are automatically extractable from the source code, thus avoiding the need for a costly formal specification. Our goal is to prove automatically that the software never executes such erroneous instructions or, at least, to give a very small list of program points that may possibly behave in undesirable ways.

In this paper, we describe our implementation and experimental studies of static analysis by abstract interpretation over a family of critical software systems, and we discuss the main technical choices and possible improvements.

## 2. REQUIREMENTS

When dealing with undecidable questions on program execution, the verification problem must reconcile *correctness* (which excludes non exhaustive methods such as simulation or test), *automation* (which excludes model-checking with manual production of a program model and deductive methods where provers must be manually assisted), *precision* (which excludes general analyzers which would produce too many false alarms), *scalability* (for software of a few hundred thousand lines), and *efficiency* (with minimal space and time requirements allowing for rapid verification during the software production process which excludes a costly iterative refinement process).

Industrialized general-purpose static analyzers satisfy all criteria but precision and efficiency. Traditionally, static analysis is made efficient by allowing correct but somewhat imprecise answers to undecidable questions. In many usage contexts, imprecision is acceptable provided all answers are sound and the imprecision rate remains low (typically 5 to 15%). This is the case for program optimization (such as static elimination of run-time array bound checks), program transformation (such as partial evaluation), etc.

In the context of program verification, where human in-

[*] CNRS (Centre National de la Recherche Scientifique)
[§] École normale supérieure. *First-name.Last-name*@ens.fr
[¶] École polytechnique. *First-name.Last-name*@polytechnique.fr

teraction must be reduced to a strict minimum, false alarms are undesirable. A 5% rate of false alarms on a program of a few hundred thousand lines would require several person-years effort to manually prove that no error is possible. Fortunately, the abstract interpretation theory shows that for any finite class of programs, it is possible to achieve full precision and great efficiency [5] by discovering an appropriate abstract domain. The challenge is to show that this theoretical result can be made practical by considering infinite but specific classes of programs and properties to get efficient analyzers producing few or no false alarms. A first experience on smaller programs of a few thousand lines was quite encouraging [3] and the purpose of this paper is to report on a real-life application showing that the approach does scale up.

## 3. DESIGN PRINCIPLE

The problem is to find an abstract domain that yields an efficient and precise static analyzer for the given family of programs. Our approach is in two phases, an *initial design* phase by specialists in charge of designing a parameterizable analyzer followed by an *adaptation* phase by end-users in charge of adapting the analyzer for (existing and future) programs in the considered family by an appropriate choice of the parameters of the abstract domain and the iteration strategy.

### 3.1 Initial Design by Refinement

Starting from an existing analyzer [3], the initial design phase is an iterative manual refinement of the analyzer. We have chosen to start from a program in the considered family that has been running for 10 years without any run-time error, so that all alarms are, in principle, due to the imprecision of the analysis. So the analyzer can be iteratively refined for this example until all alarms are eliminated.

Each refinement step starts with a static analysis of the program, which yields false alarms. Then a manual backward inspection of the program starting from sample false alarms leads to the understanding of the origin of the imprecision of the analysis. There can be two different reasons for the lack of precision:

• Some local invariants are expressible in the current version of the abstract domain but were missed either:
– because some *abstract transfer function* (see Sect. 6.2) was too coarse, in which case it must be rewritten closer to the best abstraction of the concrete transfer function [7] (see Sect. 7.3),
– or because a *widening* (see Sect. 6.3) was too coarse, in which case the iteration strategy must be refined (see Sect. 8.1);

• Some local invariants are necessary in the correctness proof but are not expressible in the current version of the abstract domain. To express these local invariants, a new abstract domain has to be designed by specialists and incorporated in the analyzer as an approximation of the reduced product [7] of this new component with the other already existing domains (see Sect. 8.2).

When this new refinement of the analyzer has been implemented, it is tested on typical examples and then on the full program to verify that some false alarms have been eliminated. In general the same cause of imprecision appears several times in the program; furthermore, one single cause of imprecision at some program point later often leads to many false alarms in the code reachable from that program point, so a single refinement typically eliminates a few dozen if not hundreds of false alarms.

This process is to be repeated until there is no or very few false alarms left.

### 3.2 Adaptation by Parameterization

The analyzer can then be used by casual end-users in charge of proving programs in the family. The necessary adaptation of the analyzer to a particular program in the family is by appropriate choice of some parameters. An example provided in the preliminary experience [3] was the *widening with thresholds*. Another example is relational domains (such as octagons [21]) which cannot be applied to all global variables simultaneously because the corresponding analysis would be too expensive; it is possible to have the user supply for each program point groups of variables on which the relational analysis should be independently applied.

In practice we have discovered that the parameterization can be largely automated (and indeed it is fully automated for octagons as explained in Sect. 8). In this way the effort to manually adapt the analyzer for a particular program in the family is reduced to a minimum.

### 3.3 Analysis of the Alarms

We implemented and used a *slicer* [22] to help in the alarm inspection process. If the slicing criterion is an alarm point, the extracted slice contains the computations that led to the alarm. However the classical data and control dependence-based backward slicing turned out to yield prohibitively large slices. Therefore we designed more restrictive ways to close the program dependences involved in the computation of a slice (following closely what we used to do manually): we restrict the data dependences to those defined by some variables: in practice we are not interested in the computation of the variables for which the analyzer already provides a value close to end-user specifications, and we can consider only the variables we lack information about (integer or floating point variables that may contain large values or boolean variables that may take any value according to the invariant).

In the future we plan to design more adapted forms of slicing: an *abstract slice* would contain the computations that lead to an alarm point and a meaningful fragment of the invariant so as to make the alarm diagnosis easier.

## 4. THE CONSIDERED FAMILY OF PROGRAMS

The considered programs in the family are automatically generated using a proprietary tool from a high-level specification familiar to control engineers, such as systems of differential equations or synchronous operator networks (block diagrams), which is equivalent to the use of synchronous languages (like LUSTRE [15]). Such synchronous data-flow specifications are quite common in real-world safety-critical control systems ranging from letter sorting machine control to safety control and monitoring systems for nuclear plants and “fly-by-wire” systems. Periodic synchronous programming perfectly matches the need for the real-time integration of differential equations by forward, fixed step numerical methods. Periodic synchronous programs have the form:

**declare** volatile input, state and output variables;

```
initialize state variables;
loop {indefinitely}
      – read volatile input variables,
      – compute output and state variables,
      – write to volatile output variables;
  wait for next clock tick;
end loop
```

Our analysis proves that no exception can be raised and that all data manipulation operations are sound. The bounded execution time of the loop body can also be checked by static analysis [11] to prove that the real-time clock interrupt does occur at idle time.

We operate on the C language source code of those systems, ranging from a few thousand lines to 132,000 lines of C source code (75 kLOC after preprocessing and simplification as in Sect. 5.1). We take into account all machine-dependent aspects of the semantics of C (as described in [3]) as well as the periodic synchronous programming aspects (for the **wait**). We use additional specifications to describe the material environment with which the software interacts (essentially ranges of values for a few hardware registers containing volatile input variables and a maximal execution time to limit the possible number of iterations in the external loop[1]).

The source codes we were interested in use only a reduced subset of the C programming language, both in the automatically generated glue code and the handwritten pieces. As it is often the case with critical systems, there is no dynamic memory allocation and the use of pointers is restricted to call-by-reference. On the other hand, an important characteristic of those programs is that the number of global and `static`[2] variables is roughly linear in the length of the code. Moreover the analysis must take values of all variables into account and the abstraction cannot ignore any part of the program without generating false alarms. It was therefore a grand challenge to design an analysis that is precise and does scale up.

## 5. THE STRUCTURE OF THE ANALYZER

The analyzer is implemented in Objective Caml [18]. It operates in two phases: the preprocessing and parsing phase followed by the analysis phase.

### 5.1 The Preprocessing Phase

The source code is first pre-processed using a standard C preprocessor, then parsed using a C99-compatible parser. Optionally, a simple linker allows programs consisting of several source files to be processed.

The program is then type-checked and compiled to an intermediate representation, a simplified version of the abstract syntax tree with all types explicit and variables given unique identifiers. Unsupported constructs are rejected at this point with an error message.

Syntactically constant expressions are evaluated and replaced by their value. Unused global variables are then deleted. This phase is important since the analyzed programs use large arrays representing hardware features with constant subscripts; those arrays are thus optimized away.

Finally the preprocessing phase includes preparatory work for trace partitioning (see Sect. 8.1.4) and parameterized packing (as described in Sect. 8.2).

### 5.2 The Analysis Phase

The analysis phase computes the reachable states in the considered abstract domain. This abstraction is formalized by a concretization function $\gamma$ [6, 7, 8]. The computation of the abstraction of the reachable states called *abstract execution* is performed compositionally, by induction on the abstract syntax, and driven by the *iterator* described in next Sect. 6. When finished, the iterator emits warnings to alarm on potential errors.

## 6. THE ITERATOR

Before starting the abstract execution, the abstract interpreter creates the global and `static` variables of the program (the stack-allocated variables are created and destroyed on-the-fly). The iterator is then run.

### 6.1 General Structure

The abstract execution starts at a user-supplied entry point for the program, such as the `main` function. Each program construct is then interpreted by the iterator according to the semantics of C as well as some information about the target environment (some orders of evaluation left unspecified by the C norm, the sizes of the arithmetic types, etc., see [3]).

The iterator transforms the C instructions into directives for the abstract domain that represents the memory state of the program (see Sect. 7.1), that is, the static and stack-allocated variables.

The iterator operates in two modes: the *iteration mode* and the *checking mode*. The iteration mode is used to generate invariants; no warning is displayed when some possible errors are detected. When in *checking mode*, the iterator issues a warning for each operator application that may give an error on the concrete level (that is to say, the program is interrupted, such as when dividing by zero, or the computed result does not obey the end-user specification for this operator, such as when integers wrap-around due to an overflow). In all cases, the analysis goes on with an abstract over-approximation of the concrete results that do not result in the immediate termination of the program.

Tracing facilities with various degrees of detail are also available. For example the loop invariants which are generated by the analyzer can be saved for examination.

### 6.2 Primitives

Whether in iteration or checking mode, the iterator starts with an abstract environment $E^\sharp$ at the beginning of a statement $S$ in the program and outputs an abstract environment $[\![S]\!]^\sharp(E^\sharp)$ abstracting $[\![S]\!](\gamma(E^\sharp))$ where $[\![S]\!]$ is the collecting semantics of $S$ [6], as follows:

- *Tests*: let us consider a conditional

$$S \quad = \quad \textbf{if}\ (c)\ \{\ S_1\ \}\ \textbf{else}\ \{\ S_2\ \}$$

(an absent **else** branch is considered as an empty execution sequence). The condition $c$ can be assumed to have no side effect and no function call, which can be handled by first performing program transformations. The iterator

[1]Most physical systems cannot run for ever and some event counters in their control programs are bounded because of this physical limitation.

[2]In the C programming language, a `static` variable has limited lexical scope yet is persistent with program lifetime. Semantically, it is the same as a global variable with a fresh name.

computes:

$$[\![S]\!]^\sharp(E^\sharp) \quad = \quad [\![S_1]\!]^\sharp(\mathit{guard}^\sharp(E^\sharp, c)) \sqcup^\sharp [\![S_2]\!]^\sharp(\mathit{guard}^\sharp(E^\sharp, \neg c))$$

where the abstract domain implements:
– $\sqcup^\sharp$ an approximation of the union $\cup$ of sets of environments;
– $\mathit{guard}^\sharp(E^\sharp, c)$ as an approximation of $\gamma(E^\sharp) \cap [\![c]\!]$ where $[\![c]\!]$ is the collecting semantics of the condition $c$, that is, the set of environments satisfying $c$. In practice, the abstract domain only implements $\mathit{guard}^\sharp$ for atomic conditions and compound ones are handled by structural induction. Each arithmetic abstract domain is supposed to implement for each comparison operator $op$ a function $op^\sharp$ such that $op^\sharp(a_1^\sharp, a_2^\sharp)$ is either $\bot$, in which case there is no $a_1 \in \gamma(a_1^\sharp)$ and $a_2 \in \gamma(a_2^\sharp)$ such that $a_1\ op\ a_2$, or $(b_1^\sharp, b_2^\sharp)$ such that for all $a_1 \in \gamma(a_1^\sharp)$ and $a_2 \in \gamma(a_2^\sharp)$ such that $a_1\ op\ a_2$, $a_1 \in \gamma(b_1^\sharp)$ and $a_2 \in \gamma(b_2^\sharp)$.
Given an atomic condition of the form $\mathbf{e}_1\ op\ \mathbf{e}_2$ and an abstract environment $E^\sharp$, we consider several cases:
∗ if $\mathbf{e}_1$ is a variable $v_1$ and $\mathbf{e}_2$ is another variable $v_2$, then if $op^\sharp(\mathbf{e}_1, \mathbf{e}_2) = \bot$ we return $\bot$, if it is $(a_1^\sharp, a_2^\sharp)$ we return $E^\sharp[v_1 \mapsto a_1^\sharp, v_2 \mapsto a_2^\sharp]$.
∗ if $\mathbf{e}_1$ (resp. $\mathbf{e}_2$) is a variable $v_1$, then if $op^\sharp(\mathbf{e}_1, \mathbf{e}_2) = \bot$ we return $\bot$, if it is $(a_1^\sharp, a_2^\sharp)$ we return $E^\sharp[v_1 \mapsto a_1^\sharp]$ (resp. *mutatis mutandis*);
∗ otherwise: if $op^\sharp(\mathbf{e}_1, \mathbf{e}_2) = \bot$ we return $\bot$, otherwise $E^\sharp$;
This implements a limited form of backward analysis, which can be made more precise using symbolic rearrangements, as with octagons (see Sect. 7.2.2).

• *Loops* are by far the most delicate construct to analyze. The analysis of the loop:

$$\textbf{while } (c)\ \{\ \mathit{body}\ \}$$

is done in two phases:
– Let us denote by $E_0^\sharp$ the environment before the loop. First, a loop invariant is computed for the head of the loop. It is an upper approximation (see Sect. 6.3) of the least invariant of $F$ where $F(E) = \gamma(E_0^\sharp) \cup [\![\mathit{body}]\!](E \cap [\![c]\!])$. This fixpoint computation is done in iteration mode and requires a widening (see Sect. 6.3).
– If in checking mode, we run in checking mode the loop body starting from the loop invariant.

• *Sequences* $i_1; i_2$: first $i_1$ is analyzed, then $i_2$, so that:

$$[\![i_1; i_2]\!]^\sharp(E^\sharp) = [\![i_2]\!]^\sharp \circ [\![i_1]\!]^\sharp(E^\sharp) \ .$$

• *Function calls* are analyzed by abstract execution of the function body in the context of the point of call, creating temporary variables for the parameters and the return value. Since the considered programs do not use recursion, this gives a context-sensitive polyvariant analysis equivalent to inlining.

• *Assignments* are passed to the abstract domain.

• *Return statement*: We implemented the `return` statement by carrying over an abstract environment representing the accumulated return values (and environments, if the function has side effects).

## 6.3 Least Fixpoint Approximation with Widening

The analysis of loops involves the iterative computation of an invariant $E^\sharp$ that is such that $F^\sharp(E^\sharp) \sqsubseteq E^\sharp$ where $F^\sharp$ is an abstraction of the concrete monotonic transfer function $F$ of the test and loop body. In abstract domains with infinite height, this is done by *widening iterations* computing a finite sequence $E_0^\sharp = \bot, \ldots, E_{n+1}^\sharp = E_n^\sharp \nabla F^\sharp(E_n^\sharp), \ldots, E_N^\sharp$ of successive abstract elements, until finding an invariant $E_N^\sharp$. The *widening operator* $\nabla$ should be sound (that is the concretization of $x \nabla y$ should overapproximate the concretizations of $x$ and $y$) and ensure the termination in finite time [6, 8] (see an example in Sect. 8.1.2).

In general, this invariant is not the strongest one expressible in the abstract domain. This invariant is then made more and more precise by *narrowing iterations*: $E_N^\sharp, \ldots, E_{n+1}^\sharp = E_n^\sharp \Delta F^\sharp(E_n^\sharp)$ where the *narrowing operator* $\Delta$ is sound (the concretization of $x \Delta y$ is an upper approximation of the greatest lower bound of $x$ and $y$) and ensures termination [6, 8].

## 7. THE ABSTRACT DOMAINS

The elements of an abstract domain abstract concrete predicates, that is, properties or sets of program states. The operations of an abstract domain are transfer functions abstracting predicate transformers corresponding to all basic operations in the program [6]. The analyzer is fully parametric in the abstract domain (this is implemented using an Objective Caml functor). Presently the analyzer uses the *memory abstract domain* of Sect. 7.1, which abstracts sets of program data states containing data structures such as simple variables, arrays and records. This abstract domain is itself parametric in the arithmetic abstract domains (Sect. 7.2) abstracting properties of sets of (tuples of) booleans, integer or floating-point values. Finally, the precision of the abstract transfer functions can be significantly improved thanks to symbolic manipulations of the program expressions preserving their abstract semantics (Sect. 7.3).

### 7.1 The Memory Abstract Domain

When a C program is executed, all data structures (simple variables, arrays, records, etc) are mapped to a collection of memory cells containing concrete values. The *memory abstract domain* is an abstraction of sets of such concrete memory states. Its elements, called *abstract environments*, map variables to abstract cells. The arithmetic abstract domains operate on the abstract value of one cell for non-relational ones (Sect. 7.2.1) and on several abstract cells for relational ones (Sect. 7.2.2, 7.2.3 and 7.2.4). An abstract value in a abstract cell is therefore the reduction of the abstract values provided by each different basic abstract domain (that is an approximation of their reduced product [7]).

#### 7.1.1 Abstract Environments

An abstract environment is a collection of abstract cells, which can be of the following four types:

• An *atomic cell* represents a variable of a simple type (enumeration, integer, or float) by an element of the arithmetic abstract domain. Enumeration types, including the booleans, are considered to be integers.

• An *expanded array cell* represents a program array using one cell for each element of the array. Formally the abstraction is component-wise, $\alpha(\emptyset) = \bot$ (where $\bot$ represents non-accessibility of dead code) and:

$$\alpha(\{(v_1^i, \ldots, v_n^i) \mid i \in \Delta\}) \quad = \quad (\bigcup_{i\in\Delta} v_1^i, \ldots, \bigcup_{i\in\Delta} v_n^i) \ .$$

Each element of the array is thus abstracted separately.

• A *shrunk array cell* represents a program array using a single cell. Formally the abstraction is $\alpha(\emptyset) = \bot$ and:

$$\alpha(\{(v_1^i, \ldots, v_n^i) \mid i \in \Delta\}) \quad = \quad \bigcup_{j=1}^n \bigcup_{i \in \Delta} v_j^i \ .$$

All elements of the array are thus "shrunk" together. We use this representation for large arrays where all that matters is the range of the stored data.

• A *record cell* represents a program record (`struct`) using one cell for each field of the record.

### 7.1.2 Fast implementation of abstract environments

A naive implementation of abstract environments may use an array. We experimented with in-place and functional arrays and found this approach very slow. The main reason is that least upper bound and widening operations are expensive, because they operate in time linear in the number of abstract cells; since both the number of global variables (whence of abstract cells) and the number of tests are linear in the length of the code, this yields a quadratic time behavior.

A simple yet interesting remark is that in most cases, least upper bound and widening operations are applied between abstract environments that are identical on almost all abstract cells: branches of tests will modify only a limited subset of the abstract cells; iterations will stabilize to a fixpoint. It is therefore desirable that those operations should have a complexity proportional to the number of *differing* cells between both abstract environments. We chose to implement abstract environments using functional maps implemented as balanced binary trees, with short-cut evaluation when computing the least upper bound or widening of identical subtrees [3, §6.2].

On a 10,000-line example we tried [3], the execution time was divided by seven, and we are confident that the execution times would have been prohibitive for the longer examples.

An additional benefit is that this implementation technique both promotes and benefits from memory sharing between different abstract values. This contributes to the rather light memory consumption of our analyzer.

### 7.1.3 Operations on Abstract Environments

Operations on a C data structure are translated into operations on cells of the current abstract environments. Most translations are straightforward.

– *Assignments:* An assignment is of the form *lval* := *rval*, where:

$$\begin{array}{llll} \textit{lval} ::= & \textit{variable} & \textit{rval} ::= & \textit{lval} \\ & \mid \textit{lval}[\textit{rval}] & & \mid \mathrm{op1}(\textit{op}, \textit{rval}) \\ & \mid \textit{lval.field} & & \mid \mathrm{op2}(\textit{op}, \textit{rval}, \textit{rval}) \end{array}$$

The memory abstract domain converts this assignment into an assignment for the arithmetic abstract domain of the form:

$$\begin{array}{lll} \textit{cell set} :=_{\textit{must/may}} \textit{rval}' & \textit{rval}' ::= & \textit{cell set} \\ & & \mid \mathrm{op1}(\textit{op}, \textit{rval}') \\ & & \mid \mathrm{op2}(\textit{op}, \textit{rval}', \textit{rval}') \end{array}$$

The abstract value of an expression *rval* is computed bottom-up by applying abstract counterparts of unary $X^\sharp \to X^\sharp$ and binary $X^\sharp \times X^\sharp \to X^\sharp$ C arithmetic, logical, and bitwise operators *op*.
Because the array subscript indices may not be fully known, the exact alias set for the *lval*s cannot be computed, this results in index sets and *may* aliases. Shrunk arrays also result in *may* aliases.

– *Guard:* The translation of concrete to abstract guards is not detailed since similar to the above case of assignments.

– *Least upper bound, widening:* Performed cell-wise between abstract environments.

## 7.2 Arithmetic Abstract Domains

The non-relational arithmetic abstract domains abstract sets of numbers while the relational domains abstract sets of tuples of numbers. The basic abstract domains we started with [3] are the intervals and the clock abstract domain abstracting time. They had to be significantly refined using octagons (Sect. 7.2.2), ellipsoids (Sect. 7.2.3) and boolean relations (Sect. 7.2.4).

### 7.2.1 Basic Abstract Domains

• *The Interval Abstract Domain* The first, and simplest, implemented domain is the domain of intervals, for both integer and floating-point values [6]. Special care has to be taken in the case of floating-point values to always perform rounding in the right direction and to handle special IEEE [17] values such as infinities and *NaN*s (Not a Number).

• *The Clocked Abstract Domain* A simple analysis using the intervals gives a large number of false warnings. A great number of those warnings originate from possible overflows in counters triggered by external events. Such errors cannot happen in practice, because those events are counted at most once per clock cycle, and the number of clock cycles in a single execution is bounded by the maximal continuous operating time of the system.
We therefore designed a parametric abstract domain. (In our case, the parameter is the interval domain [3].) Let $X^\sharp$ be an abstract domain for a single scalar variable. The elements of the clocked domain consist in triples in ${X^\sharp}^3$. A triple $(v^\sharp, v^\sharp_-, v^\sharp_+)$ represents the set of values $x$ such that $x \in \gamma(v^\sharp)$, $x - \textit{clock} \in \gamma(v^\sharp_-)$ and $x + \textit{clock} \in \gamma(v^\sharp_+)$, where *clock* is a special, hidden variable incremented each time the analyzed program waits for the next clock signal.

### 7.2.2 The Octagon Abstract Domain

Consider the following program fragment:

`R:=X−Z; L:=X; if (R>V) { L:=Z+V; }`

At then end of this fragment, we have $\mathtt{L} \leq \mathtt{X}$. In order to prove this, the analyzer must discover that, when the **then** branch is taken, we have $\mathtt{R} = \mathtt{X} - \mathtt{Z}$ and $\mathtt{R} > \mathtt{V}$, and deduce from this that $\mathtt{Z} + \mathtt{V} < \mathtt{X}$. This is possible only with a *relational domain* able to capture simple linear inequalities between variables.

Several such domains have been proposed, such as the widespread polyhedron domain [10]. In our prototype, we chose a relational abstract domain developed recently: the *octagon abstract domain* [21, 20], which is less precise but faster than the polyhedron domain: it can represent sets of constraints of the form $\pm x \pm y \leq c$, and it features a cubic time and a quadratic space cost (w.r.t. the number of variables), instead of exponential for polyhedra. Even with this reduced cost, the huge number of live variables prevents us from representing sets of concrete environments as one big abstract state (as it was done for polyhedra in [10]). Therefore we group variables into small packs and use one octagon for each pack. The set of packs is a parameter of the analysis which can be determined automatically (see Sect. 8.2.1).

Another reason for choosing octagons is the lack of support for floating-point arithmetics in the polyhedra domain.

Designing relational domains for floating-point variables is indeed a difficult task, not much studied until recently [19]. On one hand, the abstract domain must be sound with respect to the concrete floating-point semantics (handling rounding, *NaN*s, etc.); on the other hand it should use floating-point numbers internally to manipulate abstract data so that it is reasonably efficient. Because invariant manipulations in relational domains rely on some properties of the real field not true for floating-points (such as $x+y \leq c$ and $z-y \leq d$ implies $x+z \leq c+d$), it is natural to consider that abstract values represent subsets of $\mathbb{R}^N$ (in the relational invariant $x+y \leq c$, the addition $+$ is considered in $\mathbb{R}$, without rounding, overflow, etc.). Our solution separates the problem in two. First, we design a sound abstract domain for variables in the real field (our prototype uses [20]). This is much easier for octagons than for polyhedra, as most computations are simple (addition, multiplication and division by 2). Then, each floating-point expression is transformed into a sound approximate real expression taking rounding, overflow, etc. into account (we use the linear forms described in Sect. 7.3) and evaluated by the abstract domain.

Coming back to our example, it may seem that octagons are not expressive enough to find the correct invariant as $\mathtt{Z} + \mathtt{V} < \mathtt{X}$ is not representable in an octagon. However, our assignment transfer function is smart enough to extract from the environment the interval $[c, d]$ where $\mathtt{V}$ ranges (with $d \leq \max \mathtt{R}$) and synthesize the invariant $c \leq \mathtt{L} - \mathtt{Z} \leq d$, which is sufficient to prove that subsequent operations on $\mathtt{L}$ will not overflow. Thus, there was no need for this family of programs to use a more expressive and costly relational domain.

We believe that this approach is general and can be adapted to many programs by:

- adapting the determination of packs (see Sect. 8.2.1),
- choosing the right relational abstract domains *on real numbers*,
- refining the abstract transfer functions.

### 7.2.3 The Ellipsoid Abstract Domain

In our examples, we have to analyze code of the form:
**if** $(\mathtt{B})\{\ \mathtt{Y}{:=}i;\ \mathtt{X}{:=}j;\ \}$ **else** $\{\ \mathtt{X}'{:=}a\mathtt{X} - b\mathtt{Y} + t;\ \mathtt{Y}{:=}\mathtt{X};\ \mathtt{X}{:=}\mathtt{X}';\ \}$
where $a$ and $b$ are floating-point constants, $i$, $j$ and $t$ are floating-point expressions, $\mathtt{B}$ is a boolean expression, and $\mathtt{X}$, $\mathtt{X}'$, and $\mathtt{Y}$ are variables. We assume we can compute bounds to the expression $t$ by the previously described analyses, say $|t| \leq t_M$. The first branch is a reinitialization step, the second branch consists in an affine transformation $\Phi$. Since this code is repeated inside loops, the analysis has to find an invariant preserved by this code. The previously described analyses fail to find such an invariant, and so yield the imprecise result that $\mathtt{X}$ and $\mathtt{Y}$ may take any value.

To find an interval that contains the values of $\mathtt{X}$ and $\mathtt{Y}$, we have designed a new abstract domain based on ellipsoids, that can capture the required invariant. More precisely, we can show that:

**Proposition 1** *If* $0 < b < 1$, $a^2 - 4b < 0$, *and* $k \geq \left(\frac{t_M}{1-\sqrt{b}}\right)^2$, *then the constraint* $\mathtt{X}^2 - a\mathtt{X}\mathtt{Y} + b\mathtt{Y}^2 \leq k$ *is preserved by the affine transformation* $\Phi$.

The proof of this proposition follows by algebraic manipulations using standard linear algebra techniques. In our examples, the conditions on $a$ and $b$ required in Prop. 1 are satisfied. We still have to propagate the invariant in the program, and to take into account rounding errors that occur in floating-point computations (and are not modeled in the above proposition).

Having fixed two floating-point numbers $a$ and $b$ such that $0 < b < 1$ and $a^2 - 4b < 0$, we present a domain $\varepsilon_{a,b}$, for describing ellipsoidal constraints. An element in $\varepsilon_{a,b}$ is a function $r$ which maps a pair of variables $(\mathtt{X}, \mathtt{Y})$ to a floating-point number $r(\mathtt{X}, \mathtt{Y})$, the element $r$ means that for all variables $\mathtt{X}$ and $\mathtt{Y}$, $\mathtt{X}^2 - a\mathtt{X}\mathtt{Y} + b\mathtt{Y}^2 \leq r(\mathtt{X}, \mathtt{Y})$.

We briefly describe some primitives and transfer functions of our domain:

- *Assignments.* Let $r \in \varepsilon_{a,b}$ be the abstract element describing some constraints before a statement $\mathtt{X} := e$, our goal is to compute the abstract element $r'$ describing the constraints satisfied after this statement:

1. in case $e$ is a variable $\mathtt{Y}$, each constraint containing $\mathtt{Y}$ gives a constraint for $\mathtt{X}$: we take $r'$ such that $r'(U, V) = r(\sigma U, \sigma V)$ where $\sigma$ is the substitution which substitutes the variable $\mathtt{Y}$ for the variable $\mathtt{X}$;
2. in case $e$ is an expression of the form $a\mathtt{Y} + b\mathtt{Z} + t$, we first remove any constraint containing $\mathtt{X}$, then we add a new constraint for $\mathtt{X}$ and $\mathtt{Y}$. So we take:
$$r' = r[(\mathtt{X}, \mathtt{Y}) \mapsto \delta(r(\mathtt{Y}, \mathtt{Z})), (\mathtt{X}, \_)|(\_, \mathtt{X}) \mapsto +\infty] .$$
We have used the function $\delta$ defined as follows:
$$\delta(k) = \left( \left( \sqrt{b} + \left( 4f \frac{|a|\sqrt{b} + b}{\sqrt{4b - a^2}} \right) \right) \sqrt{k} + (1+f)t_M \right)^2$$
where $f$ is the greatest relative error of a float with respect to a real, $f = 2^{-13}$ and $t \in [-t_M, t_M]$. Indeed, we can show that, if $\mathtt{Y}^2 - a\mathtt{Y}\mathtt{Z} + b\mathtt{Z}^2 \leq k$ and $\mathtt{X} = a\mathtt{Y} - b\mathtt{Z} + t$, then in exact real arithmetic $\mathtt{X}^2 - a\mathtt{X}\mathtt{Y} + b\mathtt{Y}^2 \leq (\sqrt{bk} + t_M)^2$, and taking into account rounding errors, we get the above formula for $\delta(k)$;
3. otherwise, we remove all constraints containing $\mathtt{X}$ by taking the top element $r' = r[(\mathtt{X}, \_)|(\_, \mathtt{X}) \mapsto +\infty]$.

- *Union* and *widening* are computed component-wise. The widening uses thresholds as described in Sect. 8.1.2.

The abstract domain $\varepsilon_{a,b}$ cannot compute accurate results by itself, mainly because of inaccurate assignments. So we use an approximate reduced product with the interval constraints. A refinement step consists in substituting in the function $r$ the image of a couple $(\mathtt{X}, \mathtt{Y})$ by the smallest element among $r(\mathtt{X}, \mathtt{Y})$ and the floating-point number $k$ such that $k$ is the least upper bound to the evaluation of the expression $\mathtt{X}^2 - a\mathtt{X}\mathtt{Y} + b\mathtt{Y}^2$ in the floating-point numbers when considering the computed interval constraints. These refinement steps are performed:

- before computing the union between two abstract elements $r_1$ and $r_2$, we reduce each constraint $r_i(\mathtt{X}, \mathtt{Y})$ such that $r_i(\mathtt{X}, \mathtt{Y}) = +\infty$ and $r_{2-i}(\mathtt{X}, \mathtt{Y}) \neq +\infty$ (where $i \in \{1; 2\}$);
- before computing the widening between two abstract elements $r_1$ and $r_2$, we reduce each constraint $r_2(\mathtt{X}, \mathtt{Y})$ such that $r_2(\mathtt{X}, \mathtt{Y}) = +\infty$ and $r_1(\mathtt{X}, \mathtt{Y}) \neq +\infty$;
- before an assignment of the form $\mathtt{X}' := a\mathtt{X} - b\mathtt{Y} + t$, we refine the constraints $r(\mathtt{X}, \mathtt{Y})$.

These refinements steps are especially useful in handling a reinitialization iteration.

Another refinement consists in taking into account the equality relations between variables that are satisfied before an assignment of the form $\texttt{X}' := a\texttt{X} - b\texttt{Y} + t$ to synthesize constraints not only between $\texttt{X}'$ and $\texttt{X}$, but also between $\texttt{X}'$ and any variable equal to $\texttt{X}$.

Ellipsoidal constraints are then used to reduce the intervals of variables: after each assignment $A$ of the form $\texttt{X}' := a\texttt{X} - b\texttt{Y} + t$, we use the fact that $|\texttt{X}'| \leq 2\sqrt{b}\sqrt{\frac{r'(\texttt{X}',\texttt{X})}{4b-a^2}}$, where $r'$ is the abstract element describing ellipsoidal constraints just after the assignment $A$.

### *7.2.4 The Boolean Relation Abstract Domain*

Apart from numerical variables, the code uses also a great deal of boolean values, and no classical numerical domain deals precisely enough with booleans. In particular, booleans can be used in the control flow and we need to relate the value of the booleans to some numerical variables. Here is an example:

$$\texttt{B}:= (\texttt{X}{=}0);\ \textbf{if}\ (\neg\ \texttt{B})\ \{\ \texttt{Y}:=1/\texttt{X};\ \}$$

We found also more complex examples where a numerical variable could depend on whether a boolean value had changed or not. In order to deal precisely with those examples, we implemented a simple relational domain consisting in a decision tree with leaf an arithmetic abstract domain[3]. The decision trees are reduced by ordering boolean variables (as in [4]) and by performing some opportunistic sharing of subtrees.

The only problem with this approach is that the size of decision trees can be exponential in the number of boolean variables, and the code contains thousands of global ones. So we extracted a set of variable packs, and related the variables in the packs only, as explained in Sect. 8.2.2.

## 7.3 Symbolic Manipulation of Expressions

One problem of non-relational abstract domains is that they perform poorly when the variables in an expression are not independent. Consider, for instance, the simple assignment $\texttt{X} := \texttt{X} - 0.2 * \texttt{X}$ performed in the interval domain in the environment $\texttt{X} \in [0, 1]$. Bottom-up evaluation will give $\texttt{X} - 0.2 * \texttt{X} \Rightarrow [0, 1] - 0.2 * [0, 1] \Rightarrow [0, 1] - [0, 0.2] \Rightarrow [-0.2, 1]$. However, because the same $\texttt{X}$ is used on both sides of the $-$ operator, the precise result should have been $[0, 0.8]$.

In order to solve this problem, we perform some simple algebraic simplifications on expressions before feeding them to the abstract domain. Our approach is to *linearize* each expression $\mathbf{e}$, that is to say, transform it into a linear form on $V$ with interval coefficients: $[\![\mathbf{e}]\!] = \sum_{i=1}^{N}[\alpha_i, \beta_i]v_i + [\alpha, \beta]$. $[\![\mathbf{e}]\!]$ is computed by recurrence on the structure of $\mathbf{e}$. Linear operators on linear forms (addition, subtraction, multiplication and division by a constant interval) are straightforward. For instance, $[\![\texttt{X}{-}0.2{*}\texttt{X}]\!] = 0.8{*}\texttt{X}$, which will be evaluated to $[0, 0.8]$ in the interval domain. Non-linear operators (multiplication of two linear forms, division by a linear form, non-arithmetic operators) are dealt by evaluating one or both linear form argument into an interval.

Although the above symbolic manipulation is correct in the real field, it does not match the semantics of C expressions for two reasons:

- Floating-point computations incur rounding;
- Errors (division by zero, overflow, etc.) may occur.

Thankfully, the systems we consider conform to the IEEE 754 norm [17] that describes rounding very well. Thus, it is easy to modify the recursive construction of linear forms from expressions to add the error contribution for each operator. It can be an *absolute* error interval, or a *relative* error expressed as a linear form. We chose relative error which are more easily implemented and turned out to be precise enough.

To address the second problem, we first evaluate the expression in the abstract interval domain and proceed with the linearization to refine the result only if no possible arithmetic error was reported. We are then guaranteed that the simplified linear form has the same semantics as the initial expression.

[3]The arithmetic abstract domain is generic. In practice, the interval domain was sufficient.

## 8. ADAPTATION VIA PARAMETERIZATION

In order to adapt the analyzer to a particular program of the considered family, it may be necessary to provide information to help the analysis. A classical idea is to have users provide assertions (which can be proved to be invariants and therefore ultimately suppressed). Another idea is to use parameterized abstract domains in the static program analyzer. Then the static analysis can be adapted to a particular program by an appropriate choice of the parameters. We provide several examples in this section. Moreover we show how the analyzer itself can be used in order to help or even automatize the appropriate choice of these parameters.

## 8.1 Parameterized Iteration Strategies

### *8.1.1 Loop Unrolling*

In many cases, the analysis of loops is made more precise by treating the first iteration of the loop separately from the following ones; this is simply a semantic *loop unrolling* transformation: a *while* loop may be expanded as follows:

$$\textbf{if}\ (\textit{condition})\ \{\ \textit{body};\ \textbf{while}\ (\textit{condition})\ \{\ \textit{body}\ \}\ \}$$

The above transformation can be iterated $n$ times, where the concerned loops and the unrolling factor $n$ are user-defined parameters. In general, the larger the $n$, the more precise the analysis, and the longer the analysis time.

### *8.1.2 Widening with Thresholds*

The *widening with threshold* $\nabla_T$ for the interval analysis of Sect. 7.2.1 is parameterized by a *threshold set* $T$ that is a finite set of numbers containing $-\infty$ and $+\infty$ and defined such that:

$$[a,b]\ \nabla_T\ [a',b'] = [\textit{if}\ a' < a\ \textit{then}\ \max\{\ell \in T \mid \ell \leq a'\}\ \textit{else}\ a,\\ \textit{if}\ b' > b\ \textit{then}\ \min\{h \in T \mid h \geq b'\}\ \textit{else}\ b]$$

In order to illustrate the benefits of this parameterization (see others in [3]), let $x_0$ be the initial value of a variable $\texttt{X}$ subject to assignments of the form $\texttt{X} := \alpha_i * \texttt{X} + \beta_i$, $i \in \Delta$ in the main loop, where the $\alpha_i$, $\beta_i$, $i \in \Delta$ are floating point constants such that $0 \leq \alpha_i < 1$. Let be any $M$ such that $M \geq \max\{|x_0|, \frac{|\beta_i|}{1-\alpha_i}, i \in \Delta\}$. We have $M \geq |x_0|$ and $M \geq \alpha_i M + |\beta_i|$ and so all possible sequences $x^0 = x_0$, $x^{n+1} = \alpha_i x^n + \beta_i$, $i \in \Delta$ of values of variable $\texttt{X}$ are bounded since $\forall n \geq 0 : |x^n| \leq M$. Discovering $M$ may be difficult in particular if the constants $\alpha_i$, $\beta_i$, $i \in \Delta$ depend on complex boolean conditions.

As long as the set $T$ of thresholds contains some number greater or equal to the minimum $M$, the interval analysis of $\texttt{X}$ with thresholds $T$ will prove that the value of $\texttt{X}$ is bounded at run-time since some element of $T$ will be an admissible

$M$. In practice we have chosen $T$ as an exponential series. Which particular one is unimportant since it must only contain numbers which are large enough to capture stability and are small enough to capture hardware defined bounds.

#### 8.1.3 Delayed Widening

When widening the previous iterate by the result of the transfer function on that iterate at each step as in Sect. 6.3, some values which can become stable after two steps of widening may not stabilize. Consider the example:

$$\texttt{X} := \texttt{Y} + \gamma;\ \texttt{Y} := \alpha * \texttt{X} + \delta$$

This should be equivalent to $\texttt{Y} := \alpha * \texttt{Y} + \beta$ (with $\beta = \delta + \alpha\gamma$), and so a widening with thresholds should find a stable interval. But if we perform a widening with thresholds at each step, each time we widen `Y`, `X` is increased to a value surpassing the threshold for `Y`, and so `X` is widened to the next stage, which in turn increases `Y` further and the next widening stage increases the value of `Y`. This eventually results in top abstract values for `X` and `Y`.

In practice, we first do $N_0$ iterations with unions, then we do widenings unless a variable which was not stable becomes stable (this is the case of `Y` here when the threshold is big enough as described in Sect. 8.1.2). We add a fairness condition to ensure termination even in the pathological cases where a variable becomes stable every few iterations.

#### 8.1.4 Trace Partitioning

In the abstract execution of the program, when a test is met, both branches are executed and then the abstract environments computed by each branch are merged. As described in [3] we can get a more precise analysis by delaying this merging.

This means that

$$\textbf{if}\ (c)\ \{\ S_1\ \}\ \textbf{else}\ \{\ S_2\ \}\ \textit{rest}$$

is analyzed as if it were

$$\textbf{if}\ (c)\ \{\ S_1;\ \textit{rest}\ \}\ \textbf{else}\ \{\ S_2;\ \textit{rest}\ \}\ .$$

A similar technique holds for the unrolled iterations of loops.

As this process is quite costly, the analyzer performs this *trace partitioning* in a few end-user selected functions, and the traces are merged at the return point of the function. Informally, in our case, those functions that need partitioning are those who iterate simultaneously over arrays `a[]` and `b[]` such that `a[`$i$`]` and `b[`$i$`]` are linked by an important numerical constraint which does not hold in general for `a[`$i$`]` and `b[`$j$`]` where $i \neq j$. This solution was simpler than adding complex invariants to the abstract domain.

### 8.2 Parameterized Abstract Domains

Recall that our relational domains (octagons of Sect. 7.2.2, and decision trees of Sect. 7.2.4) operate on small packs of variables for efficiency reasons. This packing is determined syntactically before the analysis. The packing strategy is a parameter of the analysis; it gives a trade-off between accuracy (more, bigger packs) and speed (fewer, smaller packs). The strategy must also be adapted to the family of programs to be analyzed.

#### 8.2.1 Parameterized Packing for Octagons

We determine a set of packs of variables and use one octagon for each pack. Packs are determined once and for all, before the analysis starts, by examining variables that interact in linear assignments within small syntactic blocks (curly-brace delimited blocks). One variable may appear in several packs and we could do some information propagation (i.e. *reduction* [7]) between octagons at analysis time, using common variables as pivots; however, this precision gain was not needed in our experiments. There is a great number of packs, but each pack is small; it is our guess that our packing strategy constructs, for our program family, a linear number of constant-sized octagons, effectively resulting in a cost linear in the size of the program. Moreover, the octagon packs are efficiently manipulated using functional maps, as explained in Sect. 7.1.2, to achieve sub-linear time costs *via* sharing of unmodified octagons.

Our current strategy is to create one pack for each syntactic block in the source code and put in the pack all variables that appear in a linear assignment or test within the associated block, ignoring what happens in sub-blocks of the block. For example, on a program of 75 kLOC, 2,600 octagons were detected, each containing four variables on average. Larger packs (resulting in increased cost and precision) could be created by considering variables appearing in one or more levels of nested blocks; however, we found that, in our program family, it does not improve precision.

Our analyzer outputs, as part of the result, whether each octagon actually improved the precision of the analysis. It is then possible to re-run the analysis using only packs that were proven useful, thus greatly reducing the cost of the analysis. (In our 75 kLOC example, only 400 out of the 2,600 original octagons were in fact useful.) Even when the program or the analysis parameters are modified, it is perfectly safe to use an list of useful packs output by a previous analysis. We experimented successfully with the following method: generate at night an up-to-date list of good octagons by a full, lengthy analysis and work the following day using this list to cut analysis costs.

#### 8.2.2 Parameterized Packing for Boolean Relations

In order to determine useful packs for the boolean relations of Sect. 7.2.4, we used the following strategy: each time a numerical variable assignment depends on a boolean, or a boolean assignment depends on a numerical variable, we put both variables in a tentative pack. If, later, we find a place where the numerical variable is inside a branch depending on the boolean, we mark the pack as confirmed. In order to deal with complex boolean dependences, if we find an assignment `b:=`*expr* where *expr* is a boolean expression, we add `b` to all packs containing a variable in *expr*. In the end, we just keep the confirmed packs.

At first, we restrained the boolean expressions used to extend the packs to simple boolean variables (we just considered `b:=b'`) and the packs contained at most four boolean variables and dozens of false alarms were removed. But we discovered that more false alarms could be removed if we extended those assignments to more general expressions. The problem was that packs could then contain up to 36 boolean variables, which gave very bad performance. So we added a parameter to restrict arbitrarily the number of boolean variables in a pack. With that strategy (which limits the number of boolean variables in a pack to three) we got an efficient and precise analysis of boolean behavior.

## 9. EXPERIMENTAL RESULTS

The main program we are interested in is 132,000 lines of C with macros (75 kLOC after preprocessing and simplifi-

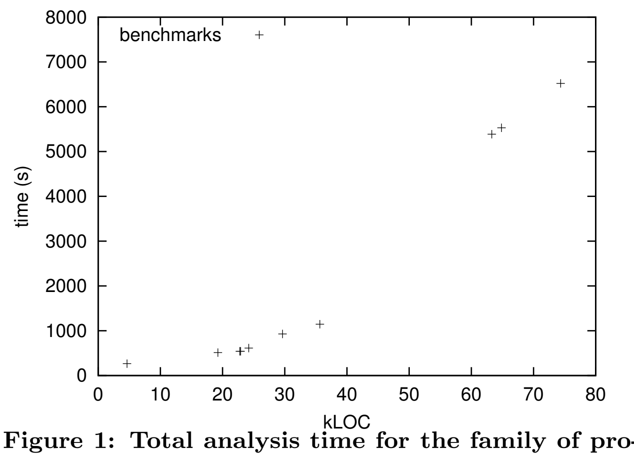

**Figure 1: Total analysis time for the family of programs**

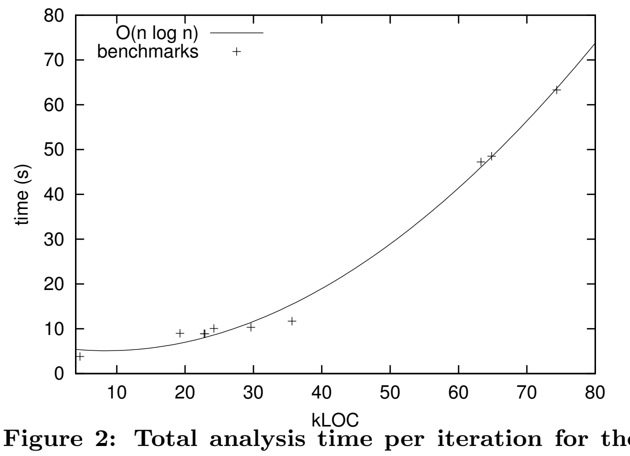

**Figure 2: Total analysis time per iteration for the family of programs**

cation as in Sect. 5.1) and has about 10,000 global/static variables (over 21,000 after array expansion as in Sect. 7.1). We had 1,200 false alarms with the existing analyzer [3] we started with. The refinements of the analyzer described in this paper reduces the number of alarms to 25, only 5 on more recent versions of the analyzed program. Fig. 1 and 2 respectively give the total analysis time and the time per fixpoint iteration of the main loop for a family of related programs.

The memory consumption of the analyzer is reasonable (450 Mb for the full-sized program). Several parameters, for instance the size of the octagon packs (8.2.1), allow for a space-precision trade-off.

The strategy outlined at the end of 8.2.1 of reusing results from preceding analysis to reduce the number of octagons reduces, on the largest example code, memory consumption from 550 Mb to 150 Mb and time from 1 h 40 min to 45 min.

## 10. RELATED WORK

Let us discuss briefly some other proof methods that could be considered.

Automated static proof of software run-time properties is a recurrent research subject since a few decades. Most fully automatic methods, such as *software model-checking* [16], do not proceed directly on the software but on a finite model, with a small enough state space, which is impossible in our case since sharp data properties must be taken into account. Moreover most modeling languages (such as PROMELA for SPIN [16]) concentrate on aspects of software systems to trace logical design errors, which in our case has already been performed at earlier stages of the software development, whereas we concentrate on abstruse machine implementation aspects of the software.

*Proof assistants* face semantic problems. The prover has to take the machine-level semantics into account (e.g., floating-point arithmetic with rounding errors as opposed to real numbers). The meticulous and precise design of abstract transfer functions for all considered abstract domains represents a very important part of our work, which can hardly be automated by lack of formal semantics of programming languages.[4] In addition, the prover needs to operate on the C source code, not on some model written in a prototyping language.

*Predicate abstraction*, which consists in specifying an abstraction by providing the atomic elements of the abstract domain in logical form [14], would certainly have been the best candidate. Moreover most implementations incorporate an automatic refinement process by success and failure [1] whereas we successively refined our abstract domains manually, by experimentation. A number of difficulties seem to be insurmountable to automate this design process in the present state of the art of deductive methods, in addition to the semantic problems shared by proof assistants:

- *State Explosion Problem:* to get an idea of the size of the necessary state space, we have dumped the main loop invariant (a textual file over 4.5 Mb).

  The main loop invariant includes 7,000 boolean interval assertions ($x \in [0, 1]$), 10,500 constant interval assertions ($x \in [a, a]$), 1,900 interval assertions ($x \in [a, b]$), 22,600 clock assertions (see Sect. 7.2.1), 5,000 constant octagonal assertions ($x = a$), 19,100 additive octagonal assertions ($a \leq x + y \leq b$), 19,200 subtractive octagonal assertions ($a \leq x - y \leq b$, see Sect. 7.2.2), 100 boolean relations (see Sect. 7.2.4) and 50 ellipsoidal assertions (see Sect. 7.2.3)[5], all these involving, e.g., over 16,000 floating point constants at most 550 of them appearing in the program text.

  Obviously some of these atomic predicates might be superfluous but on one hand it it is hard to say which ones and on the other hand this does not count all other predicates that may be indispensable at some program point to be locally precise. In order to allow for the reuse of boolean model-checkers, the conjunction of true atomic predicates is usually encoded as a boolean vector over boolean variables associated to each predicate [14] (the disjunctive completion of this abstract domain can also be used to get more precision [1], but this would introduce an extra exponential factor). Model-checking state graphs corresponding to several tenths of thousands of boolean variables (not counting hundreds of thousands of program points) is presently inappropriate.

[4]For example ESC is simply unsound with respect to modular arithmetics [12].

[5]Figures are rounded to the closest hundred. We get more assertions than variables because in the 10,000 global variables arrays are counted once whereas the element-wise abstraction yields assertions on each array element. Boolean assertions are needed since booleans are integers in C.

- *Refinement Problem:* predicate abstraction *per se* uses a finite domain and is therefore provably less powerful than our use of infinite abstract domains (see [9], the intuition is that all inductive assertions have to be provided manually). Therefore predicate abstraction is often accompanied by a refinement process to cope with false alarms [1]. Under specific conditions, this refinement can be proved equivalent to the use of an infinite abstract domain with widening [2]. These specific conditions (essentially that the widening is by constraint elimination) are not satisfied by the widening with thresholds of Sect. 8.1.2 and so all possible intervals for all possible stages and all program variables would have to be manually provided in the list of atomic predicates, and similarly for octagons, which introduces prohibitive human and computational costs for end-users. Formally this refinement is a fixpoint computation [5, 13] at the concrete semantics level, whence introduces new elements in the abstract domain state by state whereas, e.g., when introducing clocks from intervals or ellipsoids from octagons we exactly look for an opposite more synthetic point of view. Therefore the present state of the art on counterexample-based refinement does not cope with the design of adequate abstract domains.

## 11. CONCLUSION

In this experiment, we had to cope with stringent requirements. Industrial constraints prevented us from requiring any change in the production chain of the code. For instance, it was impossible to suggest changes to the library functions that would offer the same functionality but would make the code easier to analyze. Furthermore, the code was mostly automatically generated from a high-level specification that we could not have access to, following rules of separation of design and verification meant to prevent the intrusion of unproved high-level assumptions into the verification assumptions. It was therefore impossible to analyze the high-level specification instead of analyzing the C code.

That the code was automatically generated had contrary effects. On the one hand, the code fit into some narrow subclass of the whole C language. On the other hand, it used some idioms not commonly found in human-generated code that may make the analysis more difficult; for instance, where a human would have written a single test with a boolean connective, the generated code would make one test, store the result into a boolean variable, do something else do the second test and then retrieve the result of the first test. Also, the code maintains a considerable number of state variables, a large number of these with local scope but unlimited lifetime. The interactions between several components are rather complex since the considered program implement complex feedback loops across many interacting components.

Despite those difficulties, we developed an analyzer with a very high precision rate, yet operating with reasonable computational power and time. Our main effort was to discover an appropriate abstraction [6, 5] which we did by manual refinement through experimentation of an existing analyzer [3] and can be later adapted by end-users to particular programs through parameterization (Sect. 7.3 and 8). To achieve this, we had to develop two specialized abstract domains (Sect. 7.2.3 and 7.2.4) and improve an existing domain (Sect. 7.2.2). The question is now whether this intellectual process could have been automated.

## 12. REFERENCES

[1] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. *PLDI. ACM SIGPLAN Not. 36(5) (2001)*, 203–213.

[2] T. Ball, A. Podelski, and S. Rajamani. Relative completeness of abstraction refinement for software model checking. *TACAS (2002)*, LNCS 2280, Springer, 158–172.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones (2002)*, LNCS 2566, Springer, 85–108.

[4] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Trans. Comput. C-35* (1986), 677–691.

[5] P. Cousot. Partial completeness of abstract fixpoint checking. *SARA (2000)*, LNAI 1864, Springer, 1–25.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *4th ACM POPL (1977)*, 238–252.

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. *6th ACM POPL (1979)*, 269–282.

[8] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp. 2*, 4 (1992), 511–547.

[9] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. *PLILP (1992)*, LNCS 631, Springer, 269–295.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. *5th ACM POPL (1978)*, 84–97.

[11] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. *ESOP (2001)*, LNCS 2211, Springer, 469–485.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. *PLDI. ACM SIGPLAN Not. 37(5) (2002)*, 234–245.

[13] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. *SAS (2001)*, LNCS 126, Springer, 356–373.

[14] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. *CAV (1997)*, LNCS 1254, Springer, 72–83.

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

[16] G. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng. 23*, 5 (1997), 279–295.

[17] IEEE Computer Society. IEEE standard for binary floating-point arithmetic, 1985.

[18] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, documentation and user's manual (release 3.06). Tech. rep., INRIA, Rocquencourt, France, 2002. `http://caml.inria.fr/ocaml/`.

[19] M. Martel. Static analysis of the numerical stability of loops. *SAS (2002)*, LNCS 2477, Springer, 133–150.

[20] A. Miné. The octagon abstract domain library. `http://www.di.ens.fr/~mine/oct/`.

[21] A. Miné. The octagon abstract domain. *IEEE AST in WCRE (2001)*, 310–319.

[22] M. Weiser. Program slicing. *5th IEEE ICSE (1981)*, 439–449.