Runtime Abstract Interpretation for Numerical Accuracy and Robustness

Franck Védrine[1], Maxime Jacquemin[1], Nikolai Kosmatov[1,2][0000−0003−1557−2813], and Julien Signoles[1]

[1]Université Paris-Saclay, CEA, List, Palaiseau, France
firstname.lastname@cea.fr
[2]Thales Research & Technology, Palaiseau, France
nikolaikosmatov@gmail.com

Abstract. Verification of numerical accuracy properties in modern software remains an important and challenging task. One of its difficulties is related to unstable tests, where the execution can take different branches for real and floating-point numbers. This paper presents a new verification technique for numerical properties, named Runtime Abstract Interpretation (RAI), that, given an annotated source code, embeds into it an abstract analyzer in order to analyze the program behavior at runtime. RAI is a hybrid technique combining abstract interpretation and runtime verification that aims at being sound as the former while taking benefit from the concrete run to gain greater precision from the latter when necessary. It solves the problem of unstable tests by surrounding an unstable test by two carefully defined program points, forming a so-called split-merge section, for which it separately analyzes different executions and merges the computed domains at the end of the section. Our implementation of this technique in a toolchain called FLDBox relies on two basic tools, FLDCompiler, that performs a source-to-source transformation of the given program and defines the split-merge sections, and an instrumentation library FLDLib that provides necessary primitives to explore relevant (partial) executions of each section and propagate accuracy properties. Initial experiments show that the proposed technique can efficiently and soundly analyze numerical accuracy for industrial programs on thin numerical scenarios.

1 Introduction

Verification of numerical accuracy properties of critical software is an important and complex task. In programs with floating-point operations, the results of computations are approximated with respect to ideal computations on real numbers [30]. An accumulation of rounding errors can result in costly or even disastrous bugs[1][2][3]. Therefore, verifying that such behaviors do not happen, and so that *accuracy properties* do hold, is of the utmost importance. It remains a challenging research problem [28] for both dynamic and static analysis.

Abstract interpretation [8] and runtime verification [18] are two well-established program analysis techniques for verifying program properties. The former is a

[1] http://www-users.math.umn.edu/~arnold/disasters/patriot.html
[2] https://en.wikipedia.org/wiki/Vancouver_Stock_Exchange
[3] http://www-users.math.umn.edu/~arnold/disasters/sleipner.html

static technique that soundly over-approximates the program behaviors in order to verify at compile time that all of them satisfy some property of interest $\mathcal{P}$, while the latter is a dynamic technique that monitors a concrete execution in order to check that this execution satisfies $\mathcal{P}$ at runtime. Both techniques have many successful applications [5,33], but suffer from intrinsic limitations: abstract interpretation may be too slow and imprecise to be tractable, while runtime verification cannot soundly reason about all possible executions and may have a hard time dealing with properties that rely on non-executable models (e.g. real numbers) or several execution traces.

This paper presents a new verification technique for verifying numerical accuracy properties, named *Runtime Abstract Interpretation* (*RAI*), as a hybrid verification technique combining abstract interpretation and runtime verification. Similar to [12] and modern symbolic execution tools [6], the main idea of RAI is to turn a given program into an abstract interpreter for that program, following—in the simplest case—the same control-flow structure. It replaces (i) concrete values by abstract values in an abstract domain and (ii) concrete floating-point operations and comparisons by abstract transformers and predicates. By embedding an abstract interpretation engine into a runtime program execution, it aims at being sound as the former while taking benefit from the concrete run to retrieve the precision of the latter (even if the execution context is unknown at compile time, e.g. in the presence of numerical inputs from an external database). It can also take into account uncertainty of program inputs (e.g. coming from sensors), providing guarantees on their *robustness* [22].

The main difficulty of numerical property verification consists in handling unstable tests in a sound way. Indeed, an *unstable test* happens for instance when the guard of a conditional statement depends on a floating-point expression and can be evaluated to a boolean value different from the one relying on the real values. For example, if we have `x`$\in [0.9, 1.1]$ (e.g. due to input uncertainty or rounding errors) before the statement `if(x<1.0){...}else{...}`, the theoretical execution for the exact (real) value can follow the then branch, while the machine (floating-point) values can lead to the else branch. In such a case, the program execution flow diverges from the theoretical one in real numbers. For a sound analysis of the program, both branches should be considered and a possible imprecision of variables in the rest of the program should be computed comparing different control flows. Some tools [21,14,39] can soundly support unstable tests, but do not scale to large industrial code with >10,000 LOC.

RAI solves this issue by surrounding an unstable test by two carefully defined program points, *split* and *merge*, delimiting a so-called *split-merge section*, for which it separately analyzes different executions and soundly merges the computed abstract values at the end of the section. To make the technique efficient, the (partial) executions of the section are enumerated and separately analyzed only within the section itself, without repeating each time a common execution prefix and suffix before and after the section, thanks to storing and retrieving the context at the split point. A split-merge section is defined as the smallest part of the program that suits the analysis goals, while the lists of variables to save and to merge are carefully minimized. To further reduce repeated execution segments, split-merge sections can be nested: the section defined for some unstable test can be a strict subset of that for another test.

We have implemented FLDBox, a prototype RAI toolchain for verifying numerical accuracy and robustness properties on C code. Numerical properties can be specified using a set of dedicated primitives, or more generally, as annotations in the ACSL specification language [2], which are then translated into instrumented C code using these primitives by the (existing) runtime assertion checker E-ACSL [37] recently extended for their support [25]. The main steps of FLDBox rely on two new tools, FLDCompiler, that defines the split-merge sections, and an instrumentation library FLDLib[4], that provides necessary primitives to explore partial executions of a section and propagate accuracy properties. Each component can be used separately, or can be easily replaced. For instance, it is possible to replace FLDLib by Cadna [23] to obtain accuracy verification by stochastic propagation instead of conservative propagation. We have evaluated FLDBox on several small-size numerical C programs, and on two industrial case studies of synchronous reactive systems of several dozens of thousands of lines of code. The results show that the proposed technique can efficiently and soundly analyze numerical accuracy for industrial programs on *thin numerical scenarios* (where each input is replaced by a small interval of values around it).

Summary of Contributions:

- a *new hybrid verification technique*, named Runtime Abstract Interpretation, for *verifying numerical accuracy and robustness properties*, that embeds an abstract interpreter into the code and relies on split-merge sections;
- a *modular prototype implementation* of RAI, called FLDBox, based on two main components: FLDCompiler and FLDLib;
- an *empirical evaluation* of the whole FLDBox toolchain on representative programs, including industrial case studies (artifact available at [42]).

2 Motivating Numerical Example

Floating-point operations approximate ideal computations on real numbers [30] and, therefore, can introduce rounding errors. Accuracy properties express that these errors stay in acceptable bounds. Robustness of the system means that a small perturbation of the inputs (e.g. due to possible sensor imprecision [22]) will cause only small perturbations on its outputs.

Consider for instance the C function of Fig. 1. It implements an interpolation table `tbl` composed of `n` measures for linear approximation of a continuous function on a point $\texttt{in} \in [0, n-1]$. Such tables are quite common in numerical analysis. We are interested in two properties:

accuracy: the round-off error of the result (`out`) increases the imprecision of the input (`in`) by at most twice the biggest difference between two consecutive measures of the table;

robustness: the previous property is satisfied not only for every concrete input value `in`, but also near it, in $[\texttt{in} - \varepsilon, \texttt{in} + \varepsilon]$, for a given small $\varepsilon > 0$.

```
1 double interpolate(double *tbl, int n, double in) {
2   double out;
3   int idx = (int) in; // truncation to an integer
4   if (idx < 0 || idx >= n-1) // out-of-bound values
5     out = (idx < 0) ? tbl[0] : tbl[n-1];
6   else // computation from the two closest integer values
7     out = tbl[idx] + (in - idx) * (tbl[idx+1] - tbl[idx]);
8   return out;
9 }
```

Fig. 1: Motivating example: an interpolation table.

The first property will be (more precisely) expressed by the assertion of Fig. 4, as we will explain in Sec. 3. Both properties are verified for `in` $\in [0, n-1]$, but fail for values around -1. Indeed, for two close values -1 and $-1+\varepsilon$ of `in` (with a small $\varepsilon > 0$), `idx` is equal to -1 and 0 respectively. Therefore the result `out` is equal to `tbl[0]` and `tbl[0]`$+(-1+\varepsilon)\times($`tbl[1]`$-$`tbl[0]`$) \approx 2\times$`tbl[0]`$-$`tbl[1]` respectively: that is an obvious discontinuity. Any tool checking this property should raise an alarm if (and, optimally, only if) such an input is encountered.

Numerical analysis of a complex computation-intensive industrial application (typically, >10,000 lines of code) for the whole set of possible inputs is not feasible in the majority of cases. A suitable numerical property can be complex to define (and even in this example, the property above should be slightly corrected to become true, as we explain in Sec. 3). Expressing such properties for a large interval of values (like the interval `in` $\in [0, n-1]$ in our example) is not always possible (e.g. for more complex properties or functions) or not sufficient to ensure the desired precision (e.g. on irregularly-spaced interpolation data when the table entries become greater on some sub-intervals while a more precise estimate is required for other sub-intervals, or in the presence of singularities). A more precise estimate can often be found on smaller intervals (as we will illustrate on Fig. 6 in Sec. 4.2).

In practice, industrial engineers often seek to ensure accuracy and robustness properties by considering a rich test suite and by replacing in each test case the concrete value of each input variable by an interval around this concrete value, thus creating a *thin numerical scenario* from the test case. This approach allows engineers to check accuracy and robustness on such thin scenarios, better understand the numerical properties of the program, and possibly prepare their later proof if it is required. The purpose of the present work is to provide a practical and sound technique for this goal.

Dynamic analysis tools cannot soundly assess robustness and accuracy for an interval of values because they do not reason on intervals and can only check properties for a specific execution with given concrete inputs *(Issue 1)*, and because of unstable tests, like at lines 3–4: the branch taken at runtime for machine values may be different from the theoretical execution with real numbers. The imprecision of computation of `in` (prior to the call to this function) could lead to executing, say, the positive branch at runtime while the negative branch should be executed in real numbers *(Issue 2a)*.

[4] The source code of FLDLib is available at `https://github.com/fvedrine/fldlib`.

Abstract interpreters may have a hard time dealing with (possibly, nested) unstable tests [22,39] *(Issue 2b)*. They also hardly keep precise relationships between variables, e.g. between `idx` and `in` after the truncation from `double` to `int` at line 3. That usually leads to imprecise analysis results *(Issue 3)*. In addition, a practical abstract interpreter usually requires to stub input-output (I/O) functions such as communications with the environment in order to model possible behaviors outside the analysis scope *(Issue 4)*. In our example, the interpolation table values can be read during system initialization from a file by another function, like we often observed in industrial code.

Last but not least, the user needs to express the accuracy properties in a formal way and the analysis tools need to understand them. For that purpose, a formal specification language for numerical properties is required *(Issue 5)*.

In this paper, we propose a new hybrid verification technique for verifying accuracy and robustness properties, named Runtime Abstract Interpretation (RAI), embedding an abstract interpretation engine into the code, where:

- a dedicated extension of a formal specification language solves Issue 5 (Sec.4.1);
- relying on concrete runs solves Issue 4, with two possibilities: either by taking the concrete values from the environment (when these values are known to be fixed) or by defining value and error intervals for them (when not fixed);
- Issue 3 is solved since the relations between variables are implicitly kept by the execution flow, while the RAI toolchain automatically replaces the concrete floating-point values and operations by their abstract counterparts that soundly take into account round-off errors (Sec. 4.2);
- representing concrete values by abstract ones solves Issue 1;
- analyzing possible executions solves Issues 2a and 2b (Sec. 4.3).

3 Overview of Runtime Abstract Interpretation

Figure 2 describes the whole process of RAI. **Bold** font shows the main steps and elements (detailed in Sec. 4) that we have designed from scratch or extended from earlier work. We illustrate these steps for the function `abs` of Fig. 5a.

A key element of our RAI toolchain FLDBox is FLDLib, the Abstract Analysis Library (presented in Sec. 4.2). It implements (in C++) the required primitives of the analyzer (e.g. abstract domain types, transfer functions, join operators of abstract domains, split and merge instructions). Its implementation is eventually linked to the user code to produce a Self-Analyzing Executable Code, but only its API is required at compile time to allow calls to its primitives.

Our RAI toolchain takes as inputs a C source code with formal annotations in the ACSL specification language [2] that express numerical properties to be verified in the code. The first step consists in encoding the annotations as additional source code in order to evaluate them at runtime. It produces an instrumented code, that we call here Self-evaluating Code. This step is performed by the pre-existing runtime assertion checker of the Frama-C verification platform [24], namely the E-ACSL tool [37,16], that we have extended to support the target numerical properties (cf. Sec. 4.1). Alternatively, the user can manually instrument the code with property checking instructions using primitives provided by FLDLib.

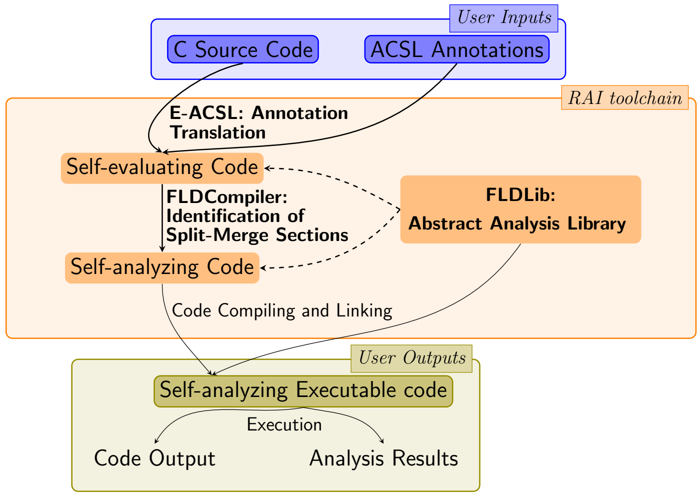

Fig. 2: Principle of Runtime Abstract Interpretation.

For example, the assertion on lines 27–29 of Fig. 5a (stating that the absolute error x_e of `x` at that point is between the given bounds) will be translated by E-ACSL into C code using the corresponding primitive (`accuracy_assert_ferr`) of FLDLib. For short, we will give a pseudo-code translation on line 29 of Fig. 5b.

The second step of RAI is performed by FLDCompiler that embeds an abstract analyzer into the code by extending the behavior of all numerical operations. It leads to Self-analyzing code (in C++) able to analyze the target annotations in addition to the normal code behavior. For that purpose, the `double` and `float` types are overloaded and become abstract domains represented by struct types. So, a variable `float x` becomes a tuple of abstract values $x = (x_{\rm r}, x_{\rm f}, x_{\rm e}, x_{\rm rel})$ whose elements represent the ideal (real) domain $x_{\rm r}$, the machine (floating-point) domain $x_{\rm f}$, the absolute error domain $x_{\rm e}$, and the relative error domain $x_{\rm rel}$. Numerical comparisons and operations are overloaded to soundly propagate these domains (cf. Sec. 4.2). To handle unstable tests, FLDCompiler defines split-merge sections allowing the analyzer to run some execution segments several times when it is necessary to relate machine and real values of diverging executions (cf. Sec. 4.3).

For the example of Fig. 5a, FLDCompiler inserts split and merge instructions on lines 8 and 23 in order to surround the unstable test on line 14 and allow the analyzer to re-execute the code between them when necessary. Let $b^{\rm r}, b^{\rm f}$ denote the branches (i.e. the truth values of `b`) executed, resp., for a real and a machine value of x. Basically, RAI *partitions* the domain of values of x into four subsets such that $(b^{\rm r}, b^{\rm f}) = (0,0),\ (0,1),\ (1,0)$ or $(1,1)$. The corresponding execution paths within the limits of the section are analyzed separately for each subset, and the results are soundly merged at the end of the section. For example, the subset $(b^{\rm r}, b^{\rm f}) = (1,0)$ is here defined by $x_{\rm r} < 0, x_{\rm f} \geq 0$. For this subset the section will be executed twice: once forcing the true branch $b = 1$ to compute the expected real domain, and once forcing the false branch $b = 0$ to compute the resulting machine domain, both being needed to soundly merge the results and compute errors. If another unstable test is met inside the section, the tool (dynamically) partitions the current subset into smaller subsets to explore relevant execution flows for the domains of values that do lead to these flows. Broadly inspired by dynamic symbolic execution [6,7] (but more complex in our case due to the need of soundly merging/re-slitting subexecutions to make the approach efficient), this exploration is the most technical part of the contribution. Its main ideas will be presented below in Sec. 4.3 using Fig. 5b, the source code being available online.

The third step of RAI is "compile & link" using a standard C++ compiler. It embeds the abstract analysis primitives' code into the final executable. Its execution performs the analysis, evaluates the annotations and produces the code output as if executed in a normal way, without RAI. If an annotation fails, the failure can be reported and, if desired, the execution can be aborted.

4 The RAI Technique in More Detail

4.1 Primitives to Express Numerical Properties

We rely on (a rich, executable subset of) the ACSL specification language [2,35] to express accuracy properties on C programs. It is a powerful language, well supported by the Frama-C [24] platform. Among others, it comes with a runtime assertion checker, named E-ACSL [37], that converts the formal annotations into C code to check them at runtime.

Specification. ACSL annotations are logical properties enclosed in special comments `/*@...*/`. They include pre-/postconditions and assertions that may be written before any C instruction. They can contain logical functions, predicates and comparison operators over terms. All constants and numerical operators are over mathematical numbers (integers in $\mathbb{Z}$, or rationals in $\mathbb{Q}$, depending on the context). C integers and floating-point values are implicitly coerced to their mathematical counterparts.

To express numerical properties, we have extended ACSL with a rich set of numerical built-ins presented in Fig. 3, in which $\mathbb{F}$ denotes either type `float` (if `f`) or `double` (if `d`). These primitives have their C counterparts supported by the FLDLib library. The two built-ins starting with `accuracy_enlarge` enlarge the intervals of values and the absolute errors to the two pairs of bounds provided as arguments. The `accuracy_assert` built-ins check whether the absolute or (if `rel` is indicated) the relative error is included within the given bounds. The `accuracy_get_[rel]err` built-ins return the lower and upper bounds of the absolute or relative error, while the `accuracy_get_real/impl` built-ins return the bounds of the real-number or implementation domain. The last built-ins print the FLDLib representation $(x_{\mathrm{r}}, x_{\mathrm{f}}, x_{\mathrm{e}}, x_{\mathrm{rel}})$ of a floating-point variable x. Thanks to these built-ins, numerical properties can be easily expressed in ACSL.

A simple ACSL assertion, stating that the absolute error is in the provided bounds, is given on lines 27–29 of Fig. 5a. As another example, the accuracy property stated in Sec. 2 for the program of Fig. 1 can be expressed—more precisely—by the assertion of Fig. 4. Here, the logic function `max_distance`

Built-in name	: type
accuracy_get_[f,d][rel]err	: $\mathbb{F} \to \mathbb{Q}^2$
accuracy_get_[f,d]real	: $\mathbb{F} \to \mathbb{Q}^2$
accuracy_get_[f,d]impl	: $\mathbb{F} \to \mathbb{Q}^2$
accuracy_enlarge_[f,d]val_err	: $\mathbb{F} \times \mathbb{Q}^4 \to$ `bool`
accuracy_assert_[f,d][rel]err	: $\mathbb{F} \times \mathbb{Q}^2 \to$ `bool`
[f,d]print	: $\mathbb{F} \to$ `bool`

Fig. 3: Numerical built-ins extending ACSL. The first three lines are logic functions, while the others are predicates. Their counterparts exist in FLDLib.

```
1 /*@ assert
2     \let (err_min, err_max) = accuracy_get_derr(in);  // primitive
3     \let cst = max_distance(tbl, n);             // logic function
4     \let (val_min, val_max) = accuracy_get_dimpl(out); // primitive
5     \let bound = max(-val_min, val_max);         // logic function
6       accuracy_assert_derr(out,
7       -2.0 * cst * max(-err_min, err_max) - 1e-16 * bound,
8       +2.0 * cst * max(-err_min, err_max) + 1e-16 * bound); */
```

Fig. 4: ACSL assertion expressing—more precisely—the accuracy property of Sec. 2 for the function of Fig. 1.

computes the maximal distance between two successive elements of `y`, that is, $\max_{i=0,\ldots,n-2} |y[i+1]-y[i]|$. Lines 4–5 compute the upper `bound` for $|$`out`$|$, which is used in the last terms on lines 7–8, added to take into account a small round-off error from the addition operation on line 7 in Fig. 1. This correction illustrates the difficulty to define correct error bounds for machine computation. Robustness follows from this assertion: a small input error leads to a small output error.

Encoding for Runtime Checking. We have extended the E-ACSL tool in two ways to support numerical properties. First, the numerical built-ins of Fig. 3 are directly compiled into their FLDLib counterparts. Second, since the ACSL specification language relies on mathematical integers and rational numbers, the generated code cannot soundly use standard C operators over integral or floating-point types. Instead, E-ACSL generates special code relying on GMP library[5] to soundly represent mathematical integers and rationals. This translation has been optimized to rely on the machine representation as much as possible, when the values fit it, and generate GMP code only when necessary. This second extension was presented in [25] and is outside of the main scope of this paper.

4.2 Propagating Abstract Values at Runtime

As the design of RAI is very technical, the following presentation focuses on the key design ideas illustrated by Fig. 5 that provides a (simplified pseudo-code) version of the resulting Self-analyzing Code for function `abs`. The reader can refer to the open-source code of FLDLib for more detail.

FLDLib is an open-source instrumentation library that infers accuracy properties over C or C++ code. It implements numerical abstract domains inspired

[5] `https://gmplib.org/`

```
float abs(float x) {




// Here FLDCompiler
// will insert:
// split(x);





  int b = (x < 0);
  if (b) {
    x = -x;
  }



// Here FLDCompiler
// will insert:
// merge(x);


// Will be translated to C by E-ACSL:
/*@ assert
    accuracy_assert_ferr(x,
    -1e-5, 1e-5);*/
  return x;
}
```

```
float abs(float_fld x){//x = (x_r, x_f, x_e) =(real,float,error)
  int b^r, b^f, b^exec;
  float_fld x^save, x^merged, x^tmp;
  x^save = x; // store init. domains at split-merge section entry
  x^merged = (⊥,⊥,⊥); // set merged domains to empty
  // fix branches taken for real and machine values:
  for(b^r,b^f ∈ {0,1}){
    x^tmp = (⊥,⊥,⊥); // store empty domains in x^tmp
    for(b^exec ∈ {b^r,b^f}){ // fix the branch b^exec to follow now
      x = x^save; // start each execution from initial domains
      // reduce domains to execute the chosen branches b^r,b^f:
      if(b^r) Assume(x_r < 0) else Assume(x_r ≥ 0);
      if(b^f) Assume(x_f < 0) else Assume(x_f ≥ 0);
      int b = b^exec; // ensure we follow the chosen branch
      if(b) { // deduce new domains after num. operations:
        x = ComputeUnitOp(−, x); // propagates x = -x;
      }
      // if real/machine executions diverge, i.e. b^r ≠ b^f:
      if(b^r != b^f){ // then merge them separately
        if(b^exec == b^r) x_r^tmp = Join_r(x_r^tmp, x_r);
        if(b^exec == b^f) x_f^tmp = Join_f(x_f^tmp, x_f);
        x_e^tmp = ComputeErr(x_r^tmp, x_f^tmp);
        x = x^tmp;
      }
    } // end of enumeration of subcases for b^exec ∈ {b^r,b^f}
    x^merged = Join(x^merged, x); // merge output variables
  } // end of enumeration of possibles cases for b^r,b^f ∈ {0,1}
  x = x^merged; // set resulting merged domains
  assert (−10^−5 ≤ x_e ≤ 10^−5); // translated ACSL assert
  return x;
}
```

Fig. 5: (a) Function abs with an assertion and a split-merge section to be inserted by FLDCompiler, and (b) the resulting (simplified) Self-analyzing Code for RAI. For simplicity, we omit here the relative error x_{rel} in $x = (x_{\mathrm{r}}, x_{\mathrm{f}}, x_{\mathrm{e}}, x_{\mathrm{rel}})$.

by those implemented in the close-source tool Fluctuat [21]. Since these domains themselves are not a key contribution of this paper, we present them briefly.

FLDLib only deals with detecting numerical errors and computing domains of numerical variables. Discrete values (pointers included) are only enumerated. In particular, it has no pointer analysis. Therefore, it is better used on *thin scenarios* that encompass concrete test cases in small intervals. In such scenarios, pointers have only one or two possible value(s). This way, RAI scales to large numerical codes or pieces of code inside bigger developments (>$10,000$ lines of code).

Domains. FLDLib domains combine intervals and *zonotopes* [20]. Zonotopes allow to maintain linear relationships between program variables V that share the same perturbations (noise symbols) by mapping V to affine forms. Sharing noise symbols between variables helps at keeping precise information since it means that the source of uncertainty is the same. We do not detail the zonotope domain here for lack of space, but Fig. 6 illustrates the benefits of combining zonotopes and intervals, in particular with a domain subdivision. For instance, if $x \in [0,1]$, an interval is more precise than a zonotope for representing x^2 (providing an interval $x^2 \in [0,1]$ instead of $[-0.25, 1]$, cf. the projection of abstractions onto the

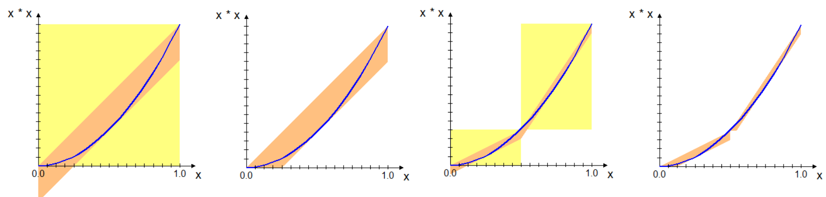

Fig. 6: Function x^2 abstracted (a) with intervals (yellow) and affine forms (orange) shown separately, and (b) the resulting intersection. The same abstractions with a subdivision, (c) shown separately, and (d) the resulting intersection.

$x \times x$ axis in Fig. 6a), but less precise for representing $x - x^2$ ($[-1, 1]$ instead of $[0, 0.25]$, cf. the distance from the diagonal in Fig. 6a). The intersection of both abstractions provides more precise results (Fig. 6b). A subdivision of the input interval into two sub-intervals significantly improves the results (Fig. 6c,d)—the orange area of Fig. 6d is much less than in Fig. 6b. As mentioned in Sec.2, using thin scenarios helps to keep precise relationships between variables.

Type Redefinition and Operation Overloading. A key principle of FLDLib consists in redefining `double` and `float` types and overloading all related operations. The `float` type becomes a structure that is called in this paper `float_fld` (cf. line 1 in Fig. 5a,b). A variable `float x;` becomes a variable `float_fld` x`;` that, mathematically speaking, contains a tuple of abstract values $(x_{\mathrm{r}}, x_{\mathrm{f}}, x_{\mathrm{e}}, x_{\mathrm{rel}})$ whose elements represent the real domain x_{r} as a zonotope, the floating-point domain x_{f} as an interval, the absolute error domain x_{e} as a zonotope, and the relative error domain x_{rel} as an interval. For simplicity, we omit the relative error computation in our examples.

Like Cadna [23] (for an execution with concrete values), FLDLib uses C++ operator overloading to propagate these domains over the program execution (with abstract values). All arithmetic operations and comparisons, as well as casts from floating-point to integral types are redefined as abstract transformers.

For instance, the unary operation assignment `x = -x;` can be replaced in the resulting Self-analyzing Code as a primitive x = $\mathit{ComputeUnitOp}(-, x)$; (cf. line 16 in Fig. 5a,b) that computes the resulting abstract values of the components of x after the operation. Similarly, a binary operation `x = x + y;` is replaced by a primitive x = $\mathit{ComputeBinOp}(+, x, y)$;. Such abstract operations (transfer functions) are well-known and we do not detail them here.

In addition to abstract versions of all numerical operations, FLDLib provides other useful primitives for constraint propagation. In the (simplified) examples of this paper, we also use a primitive Assume(`<cond>`) to assume a condition (and propagate it to all relevant domains), a primitive $\mathit{Join}(x', x'')$ to merge (join) the domains coming from different execution paths, its variants $\mathit{Join}_{\mathrm{r}}(x'_{\mathrm{r}}, x''_{\mathrm{r}})$ and $\mathit{Join}_{\mathrm{f}}(x'_{\mathrm{f}}, x''_{\mathrm{f}})$ to merge the domains for real or machine numbers only, and $\mathit{ComputeErr}(x_{\mathrm{r}}, x_{\mathrm{f}})$ to compute a new error (e.g. after such a separate merge).

Operator overloading is particularly convenient in our context since it limits necessary source-to-source transformations. We also have promising initial experiments on Ada programs that support operator overloading through the libadalang library[6]. A similar approach could be applied to C programs with no operator overloading capabilities, where such a transformation can be automatically done e.g. by the Clang compiler.

4.3 Covering All Executions for Unstable Tests

Unstable Tests by Example. The key difficulty of our method is related to unstable tests. For instance, for the conditional at line 15 in Fig. 5a, if the domains and precision of `x` ensure that both the real number and the machine number satisfy `x<0` and thus execute the same branch ($\mathtt{b} = 1$), the Self-analyzing Code needs to execute only this branch and perform the analysis (thanks to the overloaded operations) along this path to obtain a sound result. In general, the evaluation of the condition for real numbers (denoted b^{r}) can lead to the true or false branch (we write $b^{\mathrm{r}} = 1$ or 0, resp.), while the condition for machine numbers (denoted b^{f}) does not necessarily lead to the same branch. Therefore, the Self-analyzing Code has to consider four cases: $(b^{\mathrm{r}}, b^{\mathrm{f}}) \in \{0, 1\}^2$ (cf. line 7 in Fig. 5b) which create a partition of the set of possible values. It analyzes each case separately (saving and restoring initial values, cf. lines 4, 10 in Fig. 5b) and finally merges the results of all cases (cf. lines 5, 26 in Fig. 5b). For each case, the domains are reduced to fit the assumption of the case (cf. lines 12–13 in Fig. 5b) before a new execution starts. The domains of the four cases are indeed different: even if, say, b^{f} is the same, different assumptions on b^{r} lead to different domains.

We denote by b^{exec} the branch(es) to be executed in each case. For each of the two cases with $b^{\mathrm{r}} = b^{\mathrm{f}}$ (where real and machine numbers activate the same branch), it is sufficient to execute only that branch, that is, $b^{\mathrm{exec}} = b^{\mathrm{r}} = b^{\mathrm{f}}$, since its execution by assumption (and thanks to the overloaded operations) computes both the new real values and the new machine values. However, in each of the two diverging cases (with $b^{\mathrm{r}} \neq b^{\mathrm{f}}$), we need to execute the real value flow (taking $b^{\mathrm{exec}} = b^{\mathrm{r}}$) to evaluate the new real values, and the machine value flow (taking $b^{\mathrm{exec}} = b^{\mathrm{f}}$) to evaluate the new machine values (cf. lines 9, 14–15 in Fig. 5b). Both subcases are then merged accordingly: real values from the real value branch, machine values from the machine value branch (cf. line 8, 19–24 in Fig. 5b) before being merged as a complete case (cf. line 28 in Fig. 5b). Incomplete data written on lines 22–23 after the first subcase are ignored and overwritten by the second subcase. So, the machine domains coming from the execution for real values ($b^{\mathrm{exec}} = b^{\mathrm{r}}$) and the real domains coming from the execution for machine values ($b^{\mathrm{exec}} = b^{\mathrm{f}}$) are indeed ignored. Overall, line 15 is executed 6 times.

Assume we have $|x_{\mathrm{e}}| = |x_{\mathrm{f}} - x_{\mathrm{r}}| \leq 10^{-5}$ for the input value. Then the assertion on line 29 will be satisfied. For instance, for the unstable case $b_{\mathrm{r}} = 1$, $b_{\mathrm{f}} = 0$, the *Assume*'s on lines 12–13 reduce domains to $-x_{\mathrm{r}}$, $x_{\mathrm{f}} \in [0, 10^{-5}]$. After executing both subcases, i.e. after lines 20–22 in the second iteration of the internal loop, the RAI computes $x_{\mathrm{r}}^{\mathrm{tmp}}, x_{\mathrm{f}}^{\mathrm{tmp}} \in [0, 10^{-5}]$, hence $x_{\mathrm{e}}^{\mathrm{tmp}} \in [-10^{-5}, 10^{-5}]$. The constraint $x_{\mathrm{e}} \in [-10^{-5}, 10^{-5}]$ being respected in all cases, it remains respected after the merge on line 26. Notice that the execution of the Self-analyzing Code

[6] `https://github.com/AdaCore/libadalang`

after the merge point continues as a unique execution (unless a subsequent split-merge section splits it again). In this way, RAI reruns the execution segments only when it is necessary for a sound analysis of the program.

Split-Merge Sections. As illustrated by Fig. 5, in order to be sound, RAI encloses each unstable test b within a loop that executes its body several times to analyze all possible cases of evaluation of b for real and machine numbers. FLDBox provides two directives to delimit those loops: `split` marks the start of a block of code B that must be run multiple times to analyze all possible executions, while `merge` marks the point of convergence where all memory states after the executions of B must be joined into a unique state. Such a block B enclosed between these directives is called a *split-merge section*. Such sections can include several branches and be nested (for instance, for nested conditional statements). The `split-merge` directives are provided by FLDLib and inserted into the generated code by FLDCompiler.

In the general case, `split` is parameterized by the variables that must be restored before a new execution in order to ensure that the initial memory state is the same at each loop iteration (i.e. each execution of the section runs from the same state), while `merge` is parameterized by the variables to be joined after different executions. A simple example of a split-merge section is shown in Fig. 5a, where the `split` and `merge` directives become, resp., lines 2–13 and 18–28 in Fig. 5b. They are parameterized by `x` since `x` must be restored before a new execution (it may have been overwritten by a previous one at line 16) and `x` is the only section's output to be merged (cf. lines 4, 10, 26 in Fig. 5b).

For the example of Fig 1, FLDCompiler inserts a `split` directive with no argument (since `in` is never overwritten) before the cast at line 3, while a `merge` directive parameterized by `out` is inserted before line 8. Indeed, a cast from a floating-point value to an integer is a form of unstable test since the real value can be casted to a different integer than the floating-point one. The `merge` directive cannot be placed earlier because `out` would not be computed yet.

Annotation Criteria. FLDCompiler is a source-to-source program transformation that automatically annotates a program with the needed `split` and `merge` directives together with their parameters. For the sake of performance and precision, a generated split-merge section should be minimal (as small as possible), `split` should only restore what is needed, and `merge` should only join variables that are modified by the section and used afterward. Positioning the split-merge sections is done by a greedy algorithm that expands them through the code until three criteria, presented below, are satisfied. These criteria are illustrated on the example of Fig. 7 that contains the unstable test `if(2 * x + 3 < 0)`.

Criterion 1 *A `split` must* strictly dominate *its associated `merge`. Conversely, a `merge` must* strictly post-dominate *its associated `split`.*

Dominance and post-dominance relations [32] used in this criterion state that all paths that go through `split` must go through its associated `merge` and, conversely, all paths that go through `merge` must have gone through its

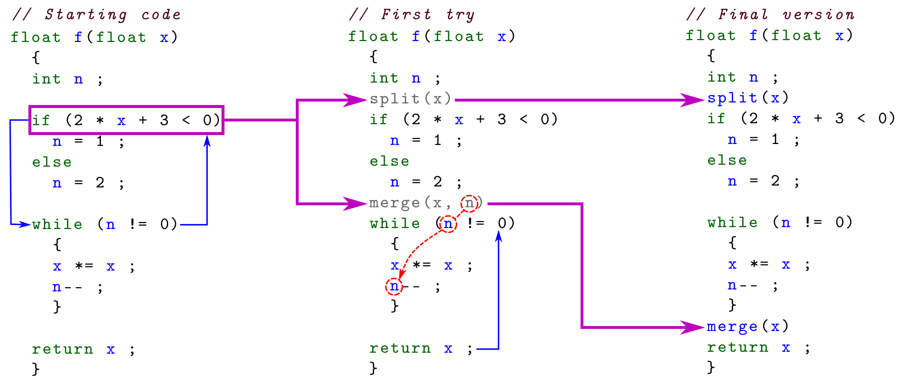

Fig. 7: (a) A code and (b),(c) transformation steps performed by FLDCompiler.

$$\textit{save-list}(p) = \{x \mid \exists(s_1, s_2), (x, s_1) \in \textit{maydef}(p) \wedge (x, s_2) \in \textit{mayref}(p)\},$$
$$\textit{merge-list}(p) = \{x \mid \exists(s_1, s_2), (x, s_1) \in \textit{maydef}(p) \wedge (s_1, s_2, x) \in \textit{datadep}(\mathcal{F}(p)) \wedge s_2 \notin p\},$$
where $\mathcal{F}(p)$ is the body of the function containing p.

Fig. 8: Computation of *save-list* and *merge-list*.

associated `split`. This criterion ensures that the memory allocations performed by `split` are eventually freed by `merge`. The other way round, the memory freed by `merge` must have been initially allocated by `split`. In our example, the `if` statement is *post-dominated* by the `while`, which is *dominated* by the `if`. Therefore, a `split` (resp. `merge`) directive is added before the `if` (resp. `while`).

Criterion 2 *A split-merge section must start and end in the same block.*

A split-merge section is enclosed in a loop that starts in the part generated by `split` and ends in the part generated by `merge`. The criterion must be satisfied to produce a syntactically valid C code, as in Fig. 5a and Fig. 7b,c.

Criterion 3 *Non floating-point variables must be kept unchanged in every memory state generated by a `split` and joined by its associated `merge`.*

This criterion is mandatory because the FLDLib library has no abstraction for non floating-point variables: merging them would lead to an error. For example, Fig. 7b presents a first positioning attempt for the split-merge section that actually violates Criterion 3. Indeed, because the value of the integer variable `n` is modified in the `if` and is needed after the `merge`, its values must be joined. To fix this, `merge` is delayed as shown in Fig. 7c. This criterion enables to prove the robustness propety in our motivating example in Fig. 1 whereas linear domains usually fail at keeping enough relationships between the `idx` variable and the input `in`.

In some cases, e.g. when an integer variable depending on the result of an unstable test is part of the outputs of the function, the split-merge section cannot be closed inside the function. In such cases (met only in one industrial example for $<$ 10% of unstable tests), the user may need to move the section to the caller(s) to respect this criterion. The user can indeed adjust the split-merge directives manually, e.g. making one section instead of two consecutive sections. This can sometimes increase precision, since domain merging is done later on the path and fewer times, at the cost of increasing the number of paths to analyze and analysis time. A similar observation is true for nested sections: without a nested section inside another one, the analysis can be more precise (with less merges) but can take longer (since more and longer path segments are replayed).

Arguments of `split` and `merge`. As said previously, `split` and `merge` take parameters that specify, resp., the variables to restore before a new execution of the section, and the ones to be eventually merged after it. To minimize the analysis cost, only necessary parameters should be generated. For example, if a variable is never modified, restoring its value is useless. These parameters for `split` and `merge` are respectively computed by a *save-list* and a *merge-list* whose computation is explained below. They are based on a dedicated data dependency analysis inspired from [26]. More precisely, for each statement p, this analysis gathers four sets, informally defined as follows:

mustdef(p) : a set of variables necessarily modified in p (that is, all executions modify them). For instance, variable `n` of Fig. 7 is in the *mustdef* set of `if`.

maydef(p) : a set of pairs (x, s) where s is a sub-statement of p that may modify the variable x. In Fig. 7, the *maydef* set of the `while` loop contains (`x`, `x+=x`). However, `x` does not belong to the *mustdef* set of `while` because, if `n` $= 0$, then `x` is left unchanged. An important difference in our approach w.r.t. [26] is that the floating-point variables read *in the branching conditions* in p are also considered in *maydef*(p) since the analysis of a branch b adds the constraints $Assume(b_r)$ and $Assume(b_f)$ (cf. lines 12–13 in Fig. 5b) that are propagated to these variables and may thus modify their domains.

mayref(p) : a set of pairs (x, s) where s is a statement of p that may read the variable x. In Fig. 7, `x` belongs to the *mayref* set of `if` because it is read by its condition. For sequence of statements S, this set does not contain variables that are read after being assigned in S. For instance, `x` (paired to any statement) does not belong to *mayref* of sequence $S \equiv$ (`x = 2; y = x + 3;`).

datadep(p) : a set of tuples (s_1, s_2, x) in which s_1 writes a variable x that is later read by s_2 (without intermediate writings). Its computation uses the three previous sets. For the example of sequence S above, variable `x` is modified by `x = 2;` and then read by `y = x + 3`, so (`x = 2`, `y = x + 3`, `x`) $\in$ *datadep*(S).

The *save-list* and *merge-list* of a split-merge section are computed as shown in Fig. 8. A variable x is added to the *save-list* of a section p if there is a statement inside p that may modify x and another statement that may read x. Said another way, if a new execution may depend on the value of a variable that could have been modified in another execution, then we need to restore it before a new execution. Dually, a variable x is added to the *merge-list* of a section p if there is a statement in p that may modify x and there is another statement outside the section that may read that modified value afterwards.

FLDCompiler is implemented as a Frama-C plug-in [36] and relies on its kernel to pretty-print the generated code. It visits the whole source code and generates the split-merge sections based on the declared type of variables. The basic version has no notion of alias, so if a pointer iterates on the cells of a floating-point array, it does not add them to the *save-list* and the *merge-list*, which may produce unsound results. To soundly solve this problem, FLDCompiler relies on Eva [4], the value analysis plugin of Frama-C, in order to know all possible targets of pointers to be added to the *save-list* and the *merge-list*. It may add unnecessary variables since Eva's analysis by abstract interpretation is conservative. Finally, FLDCompiler issues a warning if it tries to add to the lists something that is dynamically allocated and thus that does not exist at compile-time.

Path Exploration within Split-Merge Sections. The example of Fig. 5 illustrated the key ideas of the exploration This simplified approach would not be directly suitable though for nested conditions, loops or nested split-merge sections.

The actual implementation is much more technical (and cannot be presented here for lack of space): it performs a depth-first exploration of path segments inside each section, dynamically discovers new branches and records (dynamically allocated) execution contexts in a worklist of executions to be explored. Nested split-merge sections are treated by storing a section context in a stack. Since the abstract values of outcoming variables are merged at the end of the path segment of the inner section, they can be used to continue the considered execution for the outer section in a transparent way. Thanks to this approach, the directives `split`(*id*, *save-list*) and `merge`(*id*, *merge-list*), (which, in practice, have a unique identifier *id* for each section) are defined as macros. The interested reader may find all implementation details in the open-source code of FLDLib.

5 Experimental Results

Our RAI toolchain FLDBox has been evaluated [42] on (i) variations of the motivating example with different sizes of the table, (ii) a benchmark of small-size C examples, and (iii) on two large industrial case studies. They were run on an Intel Core i7 CPU, 2.60GHz with 32Gb RAM (on an artifact virtual machine, execution time can depend on the provided resources and be longer).

Motivating Example with Different Table Sizes. We first consider a version of the motivating example of Fig. 1 that loads the measures of the interpolation table from a file and calls `interpolate` with a large scenario `in` $\in [0, n-1]$. This is a very frequent code pattern in industrial code. It uses an external I/O library that is compiled with standard options and is not instrumented with our custom floating-point domains. We compare time (see Fig. 9) and precision of the tools supporting unstable tests (Fluctuat, Rosa and Precisa) and FLDBox for different sizes of the table. Rosa and Precisa do not manage such examples that generate a combinatorial explosion: with 2 elements Rosa takes 9s, with 3 elements it takes 111s and more than 20 min for 4 elements; Precisa takes 9.1s for 8 elements, 37s for 9 elements, 131s for 10 elements. Since FLDBox accepts dynamic values, the Self-analyzing Executable is compiled only once and can be used with different files, unlike Fluctuat that parses the interpolation table in the source code.

Table size	10	20	100	200	400	1000	2000
FLDBox	0.01s	0.02s	0.14s	0.47s	1.85s	11.6s	69s
Fluctuat	0.05s	0.09s	0.16s	0.28s	7.00s	92.0s	838s
Precisa	131s	TO	TO	TO	TO	TO	TO
Rosa	TO	TO	TO	TO	TO	TO	TO

Fig. 9: Analysis time for the motivating example. Timeout (TO) is set to 20 min.

FLDBox reports an accuracy error on the result of 8×10^{-6}, while Fluctuat reports a maximal accurracy error of 0.89. Hence RAI shows that the interpolate function is robust, whereas Fluctuat cannot show it, at least, without additional subdivision annotations from the user that can be tricky to find.

Benchmarks. We use benchmarks from [22,13] with unstable tests and present in [10]. They contain several small-size C examples in several categories (cf. Fig. 10). *Simple examples* show basic computations that focus on accuracy properties. *Unstable branches* are robustness tests for unstable branch handling. *Interpolation tables* contain various ways to compute an interpolation table. They also focus on testing robustness of unstable branches. *Maths* models functions of `math.h` for error estimation. *Miscellaneous* contain other examples. File `filter.c` is a second order linear filter that focuses on accuracy. File `patriot.c` is a historical example that contains a sum of 0.1 whose error shifts over time. File `complex_LU.c` finds a vector X such that $M(X) = (Y)$ for a square matrix M with a Lower/Upper decomposition. File `complex_intersect.c` shows iterative computations. File `scanf.c` shows how to manage external library functions not related to floating-point operations. The variable whose precision is analyzed is given after the file name.

Results. Each example has been annotated with ACSL assertions modeling the expected properties to use our toolchain. All of them have also been run with a timeout of 20 min in Fluctuat [21], Precisa [39] and Rosa [13]. Figure 10 presents the accuracy and time (either on top of the whole category for very small values, or per example otherwise). `ko` identifies a case where the tool failed to treat the example. `n/t` means "not translated" into PVS for Precisa or into Scala for Rosa due to the difficulty or impossibility to give an equivalent encoding of the C version. The best accuracy for a particular example is written in bold. Therefore, the table clearly shows that **FLDBox has almost always the best accuracy.**

The results of FpDebug were also recorded to show an under-approximation of the precision, where "unstable" means that FpDebug detects an unstable test and exits. They show that the results of our RAI toolchain, while being obtained using over-approximations, are not very far from the results returned by FpDebug and providing an under-approximation. Hence, **on the considered examples, FLDBox remains reasonably precise.**

Since FLDLib uses the same reasoning as Fluctuat except for constraint management, many results are merely the same. However, Fluctuat has only a limited support for unstable branches. Rosa manages them well but chains of `if`'s lead to a combinatorial explosion. Rosa approximates the errors on constant values but it is the most precise tool on non-linear computations. Precisa was used without

Target file/variable	**FLDBox**	Fluctuat	Rosa	Precisa	FpDebug
Simple examples:	< 0.01s	< 0.01s	< 0.6s	< 0.2s	
absorption.c/z	**1e-8**	**1e-8**	5.96e-8	5.96e-8	1e-8
associativity.c/u	**6.67e-16**	1.55e-15	1.55e-15	4.21e-15	-2.22e-16
division.c/z2	**1.805e-16**	5.55e-16	5.55e-16	5.55e-16	-1.57e-17
exp.c/y	**4.47e-13**	5.61e-13	n/t	4.45e-12	ko
polynome.c/t	1.066e-14	9.21e-15	**7.33e-15**	1.80e-14	-2.41e-16
relative.c/z	**2.33e-12**	**2.33e-12**	**2.33e-12**	6.59e-12	1.82e-13
triangle.c/A	**2.59e-13**	**2.59e-13**	1.58e-12	2.58e-8	-5.6e-21
Unstable branches:	< 0.01s	< 0.01s	see below	< 0.2s	
comp_abs.c/z	**4.44e-16**	2 (false al.)	3.73e-9/0.3s	4 (false al.)	-2.85e-8
comp_cont.c/y	**5.03e-5**	9.03e-5	7.0e-5/0.2s	3 (false al.)	-2.25e-8
comp_cont_nested.c/w	**1.67e-18**	**1.67e-18**	4.52e-16/3e4s	n/t	-1.0e-18
comp_cont_mult.c/res	**3.30e-5**	105 (false al.)	3.41e-5/0.4s	192 (false al.)	unstable
comp_disc_nested.c/z	**0.1** (true al.)	**0.1**	0.3/1.6s	n/t	ko
comp_disc.c/z	1.0	1.0	**0.5** (true al.)/0.2s	ko	ko
comp_model_err.c/S	**0.023** (true al.)	3.82e-1	**0.024**/2.2s	ko	ko
smartRoot.c/VAR	**1.52e-15**	0.27	1.61e-15/25s	ko	1.38e-17
cav10.c/VAR	102	101	**2.9**/1.4s	101	-3.3e-17
squareRoot3.c/VAR	**1.25e-11**	0.43	2.75e-9/4.5s	2.71 (false al.)	7.27e-17
squareRoot3Inv.c/VAR	**1.25e-9**	0.43	3.93e-9/4.5s	2.71	7.27e-17
Interpolation tables:	< 0.1s	< 0.1s	see below	see below	
inter_cond.c/res	**1.33e-5**	105 (false al.)	192/0.5s	191/0.02s	4.77e-7
inter_loop.c/result	**1.45e-6**	4.17e-6	ko	33/0.05s	-4.60e-7
inter_tbl_cast.c/out	**4e-6**	77.1 (false al.)	time out	time out	-1.04e-15
inter_tbl_loop.c/res	**4e-8**	time out	n/t	n/t	-1.04e-15
motiv_example.c/out1	**1.19e-7**	77.1 (false al.)	time out	time out	-1.04e-8
motiv_example.c/out2	**4** (true al.)	95.1	time out	time out	ko
Maths:	< 0.2s	< 0.1s	see below	see below	
sin_model_error.c/res	**2.57e-16**	**2.57e-16**	n/t	n/t	8.79e-18
sqrt_unroll.c/t.v	**7.11e-15**	7.82e-14	n/t	n/t	-4.81e-15
sqrt_fixpoint.c/Output	**3.15e-15**	1.39e-14	n/t	n/t	3.51e-16
Miscellaneous:	see below	see below	see below	see below	
filter.c/S	**1.65e-14**/0.13s	**1.65e-14**/1s	time out	time out	1.44e-16
NBody.c/VAR	**1.13e-6**/4.4s	time out	time out	n/t	1.91e-4
patriot.c/t	**1.91e-4**/0.05s	**1.91e-4**/0.8s	time out	time out	7.14e-15
complex_LU.c/det	**7.15e-15**/0.01s	n/t	n/t	n/t	n/t
complex_intersect.c/x	0.53/0.27s	**0.2**/0.6s	n/t	n/t	n/t
scanf.c/res	**4.57e-7**/0.04s	n/t	n/t	n/t	n/t

Fig. 10: Tool comparison over small-size C examples.

the SMT optimization with FPRock. It is left as future work to evaluate if it can scale better with it. Nevertheless FLDBox aims at providing **guaranteed accuracy analysis with unstable branches on real-life C code** containing loops and thousands of lines of code, while Precisa (as Rosa) is more concerned with robustness proofs of smaller algorithms. Finally, unlike the other two sound tools (Fluctuat, Precisa), **Rosa and FLDBox did not report any false alarms on these examples,** whereas Rosa has timed out on some.

Industrial Case Studies. We also experimented FLDBox on two (non public) industrial case studies (synchronous reactive systems of several dozens of thousands of lines of code) on thin scenarios coming from existing tests with relative error, resp, 10^{-6} and 10^{-16}. The first one was automatically generated in C, whereas the second one was manually written in C++. Thus only Fluctuat and our tool were used on the first, and only our tool on the second. The first one contains computations that represent physical models, with many components like interpolation tables, but also linear filters, threshold functions. The second one contains solving algorithms coming from the C++ template library for linear algebra eigen[7], which is very convenient for our instrumentation mechanism as all the floating-point code is inlined.

Results. On the first case study, FLDCompiler added about 50 `split-merge` sections whose nested depth was up to 5. Even if we only used its syntactic version (that is not based on the Eva plug-in of Frama-C, resulting in a loss of precision), the results were very useful. Our tool exercised all interesting split-merge sections by performing the simulation of 80,000 loop cycles in <24h! It took only 2s to analyze one loop cycle with FLDLib (while Fluctuat took 1h, so did not scale). All these sections have been proved to be continuous. More precisely, when the output absolute value was > 0.1, the relative error was $< 10^{-2}$, otherwise the absolute error was $< 10^{-3}$, that was acceptable for that case study.

The second case study with eigen demonstrated the need to extend FLD-Compiler to provide better results on some linear algebra algorithms and some discontinuous unstable branches. For example, the determinant computation is a continuous formula but often internally uses a LU (Lower/Upper) matrix decomposition that contains many unstable branches due to the choice of the best pivoting number. In this case, we have manually defined 25 split-merge sections (it took only about 3 hours) whose depth was up to 4. FLDBox was able to successfully analyze between 10 and 20 cycles and validate the robustness of the unstable tests. The relative error was proved to be $< 10^{-10}$ for the first 7 cycles, and then progressively increased, e.g., to $< 10^{-4}$ for the 15^{th} cycle.

FLDBox scales better than Fluctuat on these case studies for the reasons mentioned in Sec. 2 and since it does not care about pointers. Nevertheless, its scalability is directly related to the trade-off between precision and analysis time: if the number of noise symbols in zonotopes is not bounded, the analysis may be quadratic. To address this issue, FLDLib offers an option to set a bound (typically, $\sim$15) for the number of noise symbols introduced in an affine form.

On the first industrial C code, FLDBox succeeds in keeping a reasonable error for a thin scenario and thus avoiding excessive over-approximations. On the second industrial C++ code, the guaranteed numerical error delivered by FLDLib increases at every loop cycle, so that, for 20 cycles, false alarms appear from the accumulation of overapproximations because more and more unstable branches are detected. In this case, FLDLib helped to identify and better understand the tricky numerical parts of a big code.

All in all, these industrial use cases demonstrate that **FLDBox scales on thin scenarios** up to several dozens of thousands of lines of code. At worst, a few `split-merge` directives have to be manually adjusted and FLDLib provides a helpful support for this task. It is also worth noting that FLDLib can be replaced by Cadna to obtain a stochastic analysis that scales better, even if the results are non-necessarily sound but close to the expected ones. We also experimented the exact part of FLDLib (without domains) that works like FpDebug, but at source code level, and obtained the same under-approximated results as FpDebug.

[7] https://gitlab.com/libeigen/eigen

6 Related Work

Many techniques and tools [3,34,19,23,17,39,21,20,38,15,13,14,41,40,9,11,27,1] have been developed for analysis of numerical properties in the last fifteen years. They can be roughly classified in two categories: *testing* and *static analysis* tools.

Among testing tools, FpDebug [3] and Herbgrind [34] are based on Valgrind [31] and detect accuracy property failures with few false alarms. FpDebug relies on MPFR[8] to associate a highly-precise value to each floating-point value of the tested program; its results are under-approximations. Herbgrind uses symbolic execution to detect sudden important accuracy loss. Both tools scale up on bigger programs. However, unlike FLDBox, they cannot guarantee the absence of failures even on thin scenarios. Verrou [19], Cadna [23], and Verificarlo [17] aim at reporting possible instances of errors with stochastic arithmetic. The core idea consists in randomly (with a selected probability) changing the rounding mode used for each floating-point operation during the program execution. For each execution, the obtained floating-point values differ, and with enough executions, an accuracy estimation can be made with a good confidence. Like FLDBox, those tools do not avoid false alarms because of the stochastic process, but their results are rather realistic and robust. However, unlike RAI, they cannot guarantee the absence of errors.

Among static analysis tools, Fluctuat [21], Gappa [15], Rosa [13,14] and Daisy [11] use a data-flow approach with interval or zonotope abstract domains. Precisa [39], FPTaylor [38] and real2Float [27] use optimization-based approaches. Gappa, Daisy, FPTaylor, real2Float, and Precisa allow formal verification in a theorem prover by generating proof scripts or certificates. Among all these tools, only Fluctuat, Rosa and Precisa have support for unstable tests.

These last tools have different design choices and trade-offs between scalability and tightness of over-approximations. Fluctuat [21] favor some scalability with forward propagation of domains. Fluctuat scales reasonably well for programs of a few thousand lines of code. Precisa uses interval arithmetic combined with branch-and-bound optimization and symbolic error computations; Rosa uses external SMT solver like Z3 [29], while Fluctuat relies on the zonotope abstract domain [20] to represent values and errors. Compared to Rosa, Precisa and Fluctuat, FLDBox scales better and can handle I/O and memory manipulations without stubs.

FPTaylor [38] favors tightness: it handles bounding errors as an optimization problem that is soundly solved by first-order Taylor approximations of arithmetic expressions. FPTaylor generally provides tighter approximations than our toolchain. However, unlike FLDBox, it cannot analyze large programs and handles neither loops, nor I/O operations, nor unstable tests. Finally, Gappa [15] presents a third possible trade-off. Indeed, Gappa is intended to help verifying and formally proving properties on numerical programs. It is based on interval arithmetic and rewriting rules for floating-point rounding errors expressions.

Rosa [13,14] and PVS-based tools [41,40] generate suitable optimized types for given accuracy and manage unstable tests using constraint solvers. Rosa

[8] `https://www.mpfr.org`

optimizes the format of the floating-point variables given a required accuracy whereas [41] generates programs with contracts to check the stability of tests. Salsa [9] improves the accuracy of programs but it does not treat unstable tests.

RAI combines abstract interpretation [8] and runtime verification [18]. The idea of computing abstract domains at runtime (but without handling unstable tests) was proposed e.g. in [12]. Modern symbolic execution tools [6,7] also combine static and dynamic analyses by replacing concrete values by symbolic ones and exploring execution paths. But they do not need to merge/re-split/re-merge several executions to treat unstable tests, and soundly define relevant points, which constitutes the key difficulty of RAI.

Relying on various ideas of previous work (type overloading, abstract domains and transformers, enriching concrete execution with additional symbolic features, program dependency analysis), RAI combines and enriches them in order to support unstable tests, bringing specific technical contributions on how to efficiently and soundly analyze relevant executions segments several times, how to define split-merge sections and find minimal lists of variables to save/merge. To the best of our knowledge, such a combined technique for numerical analysis has never been proposed before. The main benefits of FLDBox lie in its ability to scale up well for thin scenarios while preserving soundness, and in its management of I/O and memory manipulations without the need of stubs.

7 Conclusion and Perspectives

Assessment of numerical accuracy in critical programs is crucial to prevent accumulation of rounding errors that can provoke dangerous bugs. This work has presented an original hybrid verification technique for verification of numerical accuracy and robustness, Runtime Abstract Interpretation (RAI), that combines abstract interpretation and runtime verification and is able to soundly and efficiently handle unstable tests. We implemented FLDBox, a prototype RAI toolchain, and evaluated it on a representative set of numerical C programs and on two industrial case studies. The results show that RAI can efficiently and soundly analyze numerical accuracy for industrial programs on thin numerical scenarios.

An interesting work perspective is to integrate our toolchain into a continuous integration process. For that purpose, it only requires to instrument the unit test files. Any other file (including library files) can remain unchanged. Future work also includes a larger evaluation on real-life programs and an extension of FLDBox to support all features of the C programming language. It is planned to continue the research on these topics in the ANR project Interflop.

Acknowledgement. The authors thank Romain Soulat and Thales Research and Technology for providing case studies and participation in the evaluation. We are also grateful to Jean Gassino, Gregory de la Grange and IRSN for their support and evaluation on additional case studies, as well as to the anonymous reviewers for their valuable comments.

References

1. Bard, J., Becker, H., Darulova, E.: Formally verified roundoff errors using SMT-based certificates and subdivisions. In: Third World Congress Formal Methods – The Next 30 Years (FM 2019). LNCS, vol. 11800, pp. 38–44. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_4
2. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, `http://frama-c.com/acsl.html`
3. Benz, F., Hildebrandt, A., Hack, S.: A dynamic program analysis to find floating-point accuracy problems. In: Conference on Programming Language Design and Implementation (PLDI'12) (2012)
4. Blazy, S., Bühler, D., Yakobowski, B.: Structuring abstract interpreters through state and value abstractions. In: International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'17) (2017)
5. Boulanger, J.: Static Analysis of Software: The Abstract Interpretation (2011)
6. Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: The 33rd International Conference on Software Engineering (ICSE'11). pp. 1066–1071. ACM (2011). https://doi.org/10.1145/1985793.1985995
7. Cadar, C., Sen, K.: Symbolic Execution for Software Testing: Three Decades Later. Communications of the ACM (2013)
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principles of Programming Languages (POPL'77) (1977)
9. Damouche, N., Martel, M.: Salsa: An automatic tool to improve the numerical accuracy of programs. In: Automated Formal Methods, AFM@NFM (2017)
10. Damouche, N., Martel, M., Panchekha, P., Qiu, J., Sanchez-Stern, A., Tatlock, Z.: Toward a standard benchmark format and suite for floating-point analysis. In: NSV'16 (2016)
11. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - framework for analysis and optimization of numerical programs (tool paper). In: The 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18), Held as Part of the European Joint Conferences on Theory and Practice of Software, (ETAPS'18). LNCS, vol. 10805, pp. 270–287. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_15
12. Darulova, E., Kuncak, V.: Trustworthy numerical computation in scala. In: The 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'11), part of SPLASH 2011. pp. 325–344. ACM (2011). https://doi.org/10.1145/2048066.2048094
13. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Symposium on Principles of Programming Languages (POPL) (2014)
14. Darulova, E., Kuncak, V.: Towards a compiler for reals. ACM Trans. Program. Lang. Syst. (2017)
15. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. Transactions on Mathematical Software (2010). https://doi.org/10.1145/1644001.1644003
16. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: The 28th Annual ACM Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT 2013). pp. 1230–1235. ACM (2013). https://doi.org/10.1145/2480362.2480593

17. Denis, C., de Oliveira Castro, P., Petit, E.: Verificarlo: Checking floating point accuracy through monte carlo arithmetic. In: Symposium on Computer Arithmetic (ARITH) (2016). https://doi.org/10.1109/ARITH.2016.31
18. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Engineering Dependable Software Systems. IOS Press (2013)
19. Févotte, F., Lathuilière, B.: Studying the numerical quality of an industrial computing code: A case study on code_aster. In: Numerical Software Verification (NSV) (2017). https://doi.org/10.1007/978-3-319-63501-9_5
20. Ghorbal, K., Goubault, E., Putot, S.: The Zonotope Abstract Domain Taylor1+. In: International Conference on Computer Aided Verification (CAV) (2009)
21. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI) (2011)
22. Goubault, E., Putot, S.: Robustness analysis of finite precision implementations. In: Asian Symposium on Programming Languages and Systems (APLAS) (2013)
23. Jézéquel, F., Chesneaux, J.M.: CADNA: a library for estimating round-off error propagation. Computer Physics Communications (2008)
24. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A Software Analysis Perspective. Formal Aspects of Computing (2015)
25. Kosmatov, N., Maurica, F., Signoles, J.: Efficient runtime assertion checking of properties over mathematical numbers. In: Proceedings of the 20th International Conference on Runtime Verification (RV 2020). LNCS, vol. 12399, pp. 310–322. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_17
26. Léchenet, J., Kosmatov, N., Le Gall, P.: Cut branches before looking for bugs: certifiably sound verification on relaxed slices. Formal Aspects of Computing (2018). https://doi.org/10.1007/s00165-017-0439-x
27. Magron, V., Constantinides, G.A., Donaldson, A.F.: Certified roundoff error bounds using semidefinite programming. ACM Trans. Math. Softw. **43**(4), 34:1–34:31 (2017). https://doi.org/10.1145/3015465
28. Monniaux, D.: The pitfalls of verifying floating-point computations. Transactions on Programming Languages and Systems (TOPLAS) (2008)
29. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: The 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), Held as Part of the Joint European Conferences on Theory and Practice of Software, (ETAPS'08). pp. 337–340
30. Muller, J., Brisebarre, N., de Dinechin, F., Jeannerod, C., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser (2010). https://doi.org/10.1007/978-0-8176-4705-6
31. Nethercote, N., Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Conference on Programming Language Design and Implementation (PLDI) (2007)
32. Prosser, R.T.: Applications of Boolean Matrices to the Analysis of Flow Diagrams. In: Eastern Joint IRE-AIEE-ACM Computer Conference (1959). https://doi.org/10.1145/1460299.1460314
33. Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstić, S., Lourenço, J.M., Nickovic, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A.: A Survey of Challenges for Runtime Verification from Advanced Application Domains (Beyond Software). Formal Methods in System Design (2019)

34. Sanchez-Stern, A., Panchekha, P., Lerner, S., Tatlock, Z.: Finding root causes of floating point error. ACM Sigplan Notice (2018), http://arxiv.org/abs/1705.10416
35. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language, http://frama-c.com/download/e-acsl/e-acsl.pdf
36. Signoles, J., Antignac, T., Correnson, L., Lemerre, M., Prevosto, V.: Frama-C Plug-in Development Guide, http://frama-c.com/download/plugin-developer.pdf
37. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper. In: International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES) (2017)
38. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. Transactions on Programming Languages and Systems (2018). https://doi.org/10.1145/3230733
39. Titolo, L., Feliú, M.A., Moscato, M.M., Muñoz, C.A.: An abstract interpretation framework for the round-off error analysis of floating-point programs. In: Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (2018). https://doi.org/10.1007/978-3-319-73721-8_24
40. Titolo, L., Moscato, M., Muñoz, C.A.: Automatic generation and verification of test-stable floating-point code. arXiv e-prints (2020)
41. Titolo, L., Muñoz, C.A., Feliú, M.A., Moscato, M.M.: Eliminating unstable tests in floating-point programs. In: Logic-Based Program Synthesis and Transformation (LOPTSR) (2018). https://doi.org/10.1007/978-3-030-13838-7_10
42. Védrine, F., Jacquemin, M., Kosmatov, N., Signoles, J.: Companion artifact evaluated by the VMCAI 2021 artifact evaluation committee. Zenodoo (Nov 2020). https://doi.org/10.5281/zenodo.4275521