Runtime assertion checking and static verification: Collaborative partners

Fonenantsoa Maurica, David R. Cok, Julien Signoles

Runtime Assertion Checking and Static Verification: Collaborative Partners

Fonenantsoa Maurica, David R. Cok, and Julien Signoles

CEA, LIST, Software Safety and Security Laboratory,
PC 174, 91191 Gif-sur-Yvette, France
david.cok@cea.fr

Abstract. Runtime assertion checking aspires to a similar level of sound and complete checking of software as does static deductive verification. Furthermore, for the same source language and specification language, runtime and static checking should implement as closely as possible the same semantics. We describe here the architecture used by two different systems to achieve this goal. We accompany that with descriptions of novel designs and implementations that add new capabilities to runtime assertion checking, bringing it closer to the feature coverage of static verification.

1 Introduction

Automated deductive, static verification of software has been increasing in capability over the past decade.[1] This trend was initially fueled by performance and feature improvements in SMT solvers and has now reached the point that software verifications of industrial software in practical use are being executed [9]. Similarly, runtime assertion checking is improving in capability. Runtime checking requires creative and efficient implementations to be able to execute programs instrumented with runtime checks effectively.

In addition, runtime and static checkers are increasingly part of suites of tools, with the assertions being checked coming from some common specification language. Other related tools might perform tasks like specification inference, white box testing, and abstract interpretation. As parts of tool suites, acting on common programming and specification languages, these tools should all adopt a common semantics for both the programming and specification languages. Ideally they accomplish this using a common software infrastructure.

This paper has two goals. First we describe how two different tool suites — OpenJML [13,14,15,16] for Java and the Java Modeling Language [10] and Frama-C [26] for ANSI C and the ANSI-C specification language [7] — have each architected common infrastructures in order to achieve both common semantics for their respective languages and common software implementations. Secondly, we describe advances in runtime assertion checking that are closing the gap

[1] as has abstract-interpretation-based static analysis, but in this paper we focus on proof-based verification.

between the kinds of assertions that can be checked by runtime checkers and those provable by static deductive program verification.

2 Tool Suites for Specification Languages

In response to the ubiquitous presence of non-robust software, and in particular, software that contains safety or security risks, there is active research and tool development whose goal is to ensure that software does what it is supposed to do. However, defining what a software system is supposed to do is not simple. One aspect, called implicit specifications, is that a program should not violate any of the rules of its underlying programming language, such as executing undefined operations. This is important, since such undefined operations, like out of bounds memory accesses, are an important contributor to security vulnerabilities. However, this aspect alone does not enable checking that a program's actions are correct. For that we need a means to express the functional requirements for the program, using explicit specifications, in a precise enough way that those requirements can be checked against the implementation by (largely) automated tools. That is we need languages to express formal specifications.

A number of such specification languages are in active use, each paired with a programming language:

- Eiffel [29], a programming language, has a built-in specification language;
- JML [10], the Java Modeling Language, expresses specifications for Java programs;
- ACSL [7], the ANSI/ISO-C Specification Language, expresses specifications for ANSI-C programs, and the in-development ACSL++ specification language for C++;
- SPARK [3] expresses specifications for the Ada programming language;
- Spec# [4] expresses specifications for C#, with the following CodeContracts [20] system working for .NET environments;
- Dafny [28] is a specification and programming language purpose-built for verification.

These are all examples of *Behavioral Interface Specification Languages* [24] (BISLs), in that the specification language syntax and semantics are closely aligned with the associated programming language, with modifications to accommodate logical specification and reasoning. Alternative examples are the Z specification language [37] and the B-method [1] whose designs are more mathematical and programming-language independent. The rationale for BISLs is that the similarity to programming languages makes learning easier. All of these languages use similar designs and follow the pioneering work of Larch [22].

Each of these specification/programming languages has associated tool suites. In this paper we will focus on two such tool suites: the OpenJML [13,14,15,16] tool for Java and JML, and the Frama-C [26] platform for C and ACSL. We will also limit discussion to two applications: automated static deductive verification (DV) and runtime assertion checking (RAC).

DV follows the following paradigm: the intent of the software under study (the 'target software') is expressed in machine-readable specifications, with the implicit specifications being generated by tools based on the programming language semantics; the specifications and the target software are together translated into a logical form; a logical proof tool then determines, if possible, whether the logical representations of the specifications and the implementation are consistent. If so, then the implementation is considered *verified*, that is, to be consistent with the specifications; if not, then either the implementation or the specifications (or both) have some fault to be found and corrected. Automation is critically important for the technique to become widespread and for efficiency in application, though some elements of the proof in some tools are delegated to interactive provers. Also, human interaction takes the form of writing and debugging the specifications so that they are amenable to machine proof. Specifications for programs that are affected by the external system or physical environment must include models of those aspects as well. For example, software for cyber-physical systems will include models of the physics of the physical world, including the possibility of inaccuracies or outright failures in sensors and effectors.

Runtime assertion checking[2] also takes explicit and (possibly tool-generated) implicit specifications as input. The specifications are converted into boolean assertions that are then compiled into the target software as instrumentation. The target software is then run as usual, perhaps on a suite of dynamic test cases. If any instrumented assertion is found to be false during these executions, the RAC platform will alert the user to the assertion violation. If no such alert is given, then the target software meets the specifications for the given set of test inputs. Software that interacts with the environment is a particular challenge for RAC (and dynamic testing in general), because it is difficult to arrange for all the unusual situations and error conditions that the environment might display.

DV can prove correct behavior for any input, if the required proof is not beyond the capability of the underlying tools. RAC always succeeds but only checks assertions that are executable and only for those inputs tested. In each case, "correctness" is measured by conformance to the specifications, which themselves must be reviewed for fidelity to the system's actual requirements. Both tools are useful together, and all the more so when the target software is safety- or security-critical.

3 Software Architecture

With similar goals and similar reliance on a common specification language, it is good design that static and runtime checking would use a common architecture. In the following subsections we describe how that is achieved by two different systems.

[2] Here we are distinguishing runtime assertion checking from runtime verification [6], which typically deals with temporal properties, e.g. LTL properties.

3.1 OpenJML for Java and JML

The Java Modeling Language is a specification language for Java programs. An example of JML is given in Fig. 1. Syntactically, JML specifications are written as structured Java comments (beginning with `//@` or `/*@`). A method's specification is expressed as a sequence of *clauses*. The **requires** clause is a pre-condition, stating what must be true at the time a method is called, and then equivalently, what may be assumed when checking the implementation of a method. The **assignable** clause denotes a frame-condition, which must list any memory locations that are possibly modified by a method. An **ensures** clause is a post-condition, stating conditions the method implementation must guarantee and thus what may be assumed by callers after the method executes. A **signals** clause is a post-condition that must hold when the method exits by throwing an exception. **Invariant** clauses are part of a class's specification and state data structure consistency properties that must be maintained by all methods. Tools such as OpenJML [13,14,15,16] and KeY [2] are able to read and check the consistency of the Java implementation and the associated specifications.

OpenJML is a tool built on the OpenJDK [42] Java compiler. The architecture of OpenJDK and OpenJML is shown in Fig. 2. As is common for compilers, OpenJDK has multiple phases: the input source code is scanned, parsed, names resolved, and type-checked, producing a forest of Abstract Syntax Trees (ASTs) representing the program. This AST is then subject to various optimizations and transformations and then emitted as Java byte code. The OpenJDK compiler phases are Java classes that are readily extended by JML versions, which scan, parse, resolve, and type-check the text contained in the JML annotations along with the Java code. The JML annotations are converted into assumptions and assertions that are inserted into the AST. This translation step embodies the semantics of the JML specifications. Furthermore, the AST serves as an intermediate representation and the focal point for other tools and program analyses. It can be pretty-printed, subjected to other programmer-initiated transformations, and the like. In this case, the modified ASTs can be sent to the (unmodified) code-generation phase to produce output byte-code with embedded JML assertions. Or the ASTs can be sent to the Java and JML logical encoding phase, which produces an SMT-LIB [5] equivalent of the AST (embodying the semantics of Java). SMT solvers can then determine whether all assertions in the SMT-LIB encoding are valid. A central point of this architecture is that the JML semantics are embodied in the JML-enhanced AST, which is used by both runtime and static checking. Thus by design the two modes of checking rely on

```
1 //@ requires i != Integer.MIN_VALUE;
2 //@ assignable \nothing;
3 //@ ensures \result >= 0 && (\result == i || \result == -i);
4 //@ signals (Exception e) false;
5 int abs(int i);
```

Fig. 1. Example JML specification of an absolute-value method.

a common semantics and a common implementation of AST transformations and optimizations. The Java semantics is still implemented separately, since for runtime checking the Java semantics is embodied in the code generator and for static checking it is embodied in the logical encoding.

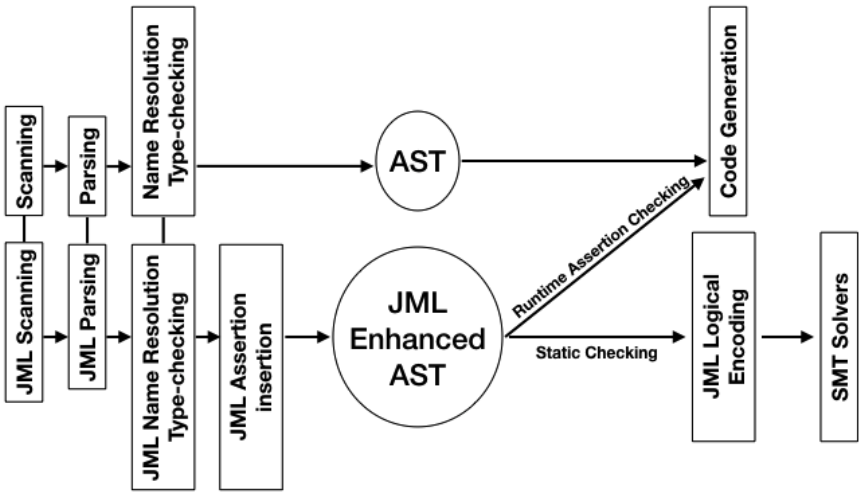

Fig. 2. Architecture of OpenJDK and OpenJML

3.2 Frama-C for C Code and (E-)ACSL Specifications

Frama-C [26] is an open source platform that provides a collection of interoperable sound software analysis tools for C source code (more precisely, ISO C99 code). The analyzed C source code may be annotated with formal specifications written in the ACSL specification language [7]. ACSL formal annotations may also be generated by Frama-C analyzers in order to be verified by others. ACSL shares many features with JML for Java, but relies on the C syntax, as shown in Fig. 3, which is a translation of the JML example of Fig. 1 to ACSL. Note that there is no **signals** clause in ACSL since there is no exception mechanism in C.

The Frama-C platform is based on a common kernel that provides a uniform setting and common services to analyzers seen as plug-ins, as depicted by its architecture shown in Fig. 4. The most important service shared by the Frama-C analyzers is a common normalized typed AST for ACSL-annotated C code. It ensures that every Frama-C tool has the very same abstract view of the analyzed code and the same pieces of information about implementation-defined behaviors (e.g., the size of C types and endianness are provided by a kernel parameter controlling machine-dependent information: they are shared by every analyzer).

```
#include <limits.h>

//@ requires i != INT_MIN;
//@ assigns \nothing;
//@ ensures \result >= 0 && (\result == i || \result == -i);
int abs(int i);
```

Fig. 3. Example ACSL specification of an absolute-value function.

However, the Frama-C kernel only provides a small amount of semantic information about the code: the Frama-C plug-ins must analyze the AST in a consistent and sound way with respect to the ISO C99 standard and the ACSL reference manual. Only a few kernel checks prevent introducing inconsistency and unsoundness. Among them, the kernel checks (1) that most AST internal invariants are preserved by program transformation and (2) that the validity statuses of properties are consistently emitted by Frama-C analyzers (that is, no analyzer A indicates that property p is valid, while analyzer B indicates that property p is invalid) [17].

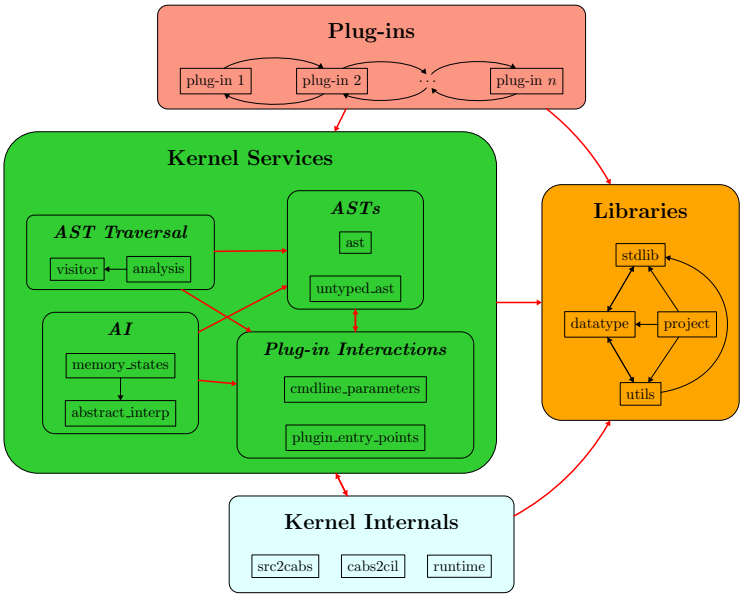

Fig. 4. Frama-C Software Architecture

Among many others, Frama-C plug-ins include three principal plugins: the abstract-interpretation based value analysis Eva [8], the deductive verification tool Wp, and the runtime assertion checker E-ACSL [36].

Eva computes a sound over-approximation of the set of values for each program memory location at each program point. It also checks each potential undefined behavior and raises an alarm for any of them as soon as it might happen. These alarms are also expressed by ACSL annotations: if proven valid (resp. invalid) (possibly, by another Frama-C analyzer), it ensures that the undefined behavior never happens (resp. does happen in at least one concrete execution). In addition, to checking for undefined behaviors, Eva also tries to evaluate ACSL annotations, but it usually succeeds for the simplest ones only.

Indeed, verifying statically that the analyzed code satisfies its ACSL annotations is the goal of Wp, by means of deductive verification. For that purpose, Wp generates proof obligations to be proven by external provers (e.g., Alt-Ergo, Z3, CVC4) or proof assistants (e.g., Coq, PVS) through Why3 [21].

Compliance of the code to ACSL annotations may also be checked at runtime through the E-ACSL tool, which converts each annotation to an equivalent C expression. However, even if close to JML, the ACSL specification language was designed with deductive verification in mind. Thus, some of its constructs cannot be checked in finite time at runtime, typically (unbounded) quantifications over all mathematical integers or reals. Consequently, the E-ACSL tool only deals with a (large) subset of the ACSL specification language, named the E-ACSL specification language [19,35].[3] It is worth noting that, while being consistent, the semantics of the ACSL and E-ACSL specification languages differ. The former is a (total) mathematical semantics such that the predicate `1/0 == 1/0` is well defined and trivially holds by reflexivity of equality. However, this semantics is not suitable at runtime because terms such as `1/0` cannot be evaluated. To solve this issue, the E-ACSL specification language follows Chalin's strong validity principle [12]: such terms and the predicates that use them are undefined. Consequently, the E-ACSL tool reports a runtime error when trying to evaluate them. JML also uses a semantics requiring well-defined specification expressions for both static and runtime checking.

4 Recent Improvements in Runtime Assertion Checking

In some arenas, RAC has been the poor cousin to DV. One of the reasons for this opinion is that static verification, when successful, can prove the absence of errors and property violations for all permitted inputs, whereas RAC only demonstrates this for the test inputs with which the program is run. On the other hand, DV attempts can fail because of lack of capability in the logical solvers used to determine validity of assertions.[4] RAC can always run a program, albeit

[3] Similarly, runtime checking of JML does not encode unbounded or even very large ranges of quantification, though the language subset supported by RAC in OpenJML is not as precisely defined as E-ACSL is for Frama-C.

[4] This is not just a theoretical concern. The state-of-the-art in SMT technology is rapidly evolving, but still does not efficiently handle all the concepts natural to software. Proofs using bit-vector operations on 64-bit numbers and floating-point operations can routinely take tens of minutes if they complete at all; quantified ex-

with some performance degradation caused by assertion checking instrumented into the program. A second problem with RAC, however, is that not all of the assertions contained in a specification are executable. Although DV may encounter assertions that are not provable, the set of such assertions is smaller than the current limitations of RAC.

Accordingly it is a research area for RAC to find well-performing algorithms to check what have been to date non-executable assertions. The following subsections describe some in-progress advances in this area. Most of these advances are too recent to have a full assessment of their performance and general applicability; such studies are planned for future publications. Here we will describe the overall motivation, present the current state of practice and outline the in-progress advances. This discussion focuses on the work on Frama-C using E-ACSL for C programs, though most considerations apply also to RAC in OpenJML.

4.1 Memory-Related Properties

As already explained, E-ACSL for C is close to JML for Java. However, since the C programming language contains low-level constructs to access to the program memory, in particular pointers, E-ACSL must be able to express memory-related properties. Consequently, it contains a set of built-in logic functions and predicates that are absent from JML (and from the other behavioral specification languages designed for high-level programming languages).

One such construct is `\valid(p)`, which means that pointer `p` can safely be dereferenced in order to access the pointed-to value. Another construct is `\initialized(&x)`, stating that variable `x` has been initialized. Checking these kinds of properties at runtime is usually the role of dynamic memory analyzers such as AddressSanitizer [33] or MemCheck [34] that rely on efficient implementations of memory shadowing. Such tools are able to access in constant time the necessary information (such as its validity or its initialization status) about a particular memory address to detect memory violations at runtime.

However, not only is E-ACSL able to express properties about some particular address, but it may also refer to allocated memory blocks. For instance, `\block_length(p)` is the size (in bytes) of the memory block containing `p`, `\base_address(p)` is the first address of the memory block containing `p`, and `\offset(p)` is the byte-offset from this first address to `p`. Unfortunately, traditional memory shadowing techniques are not able to express such block-level properties. Consequently, the E-ACSL tool relies on a custom shadow memory model [41] with a compact representation of block-level properties to support these operations.

Evaluations over standard benchmarks demonstrated that this memory model is able to express more properties than classical models [40], while being as effi-

pressions require heuristic algorithms to decide when to instantiate the expressions; recursion is not natural to ground solvers such as SMT tools; dynamic allocations and heterogeneous casts between integers and pointers (in C) require low-level memory models that make proof intractable.

cient as MemCheck (but still slower than AddressSanitizer) and consuming *less* memory than these tools [41].

Note that in DV, memory models are based on abstractions: such memory models allow proof to be done automatically but can only express properties about high-level data structures such as arrays, records and objects, while low-level models may express additional operations (such as dynamic allocations and heterogeneous casts between integers and pointers), but usually require the help of a proof assistant (e.g. Coq) to prove the expected program properties. RAC can deal with any programming language construct, but cannot accurately check finest properties that may be expressed by DV memory models.

4.2 Efficient Integer Computations

Computations over integers are ubiquitous in software. Most programming languages provide various precisions of integers for different needs. Specifications, however, are most naturally expressed in unbounded, mathematical integers ($\mathbb{Z}$). Indeed, Chalin's research [11] showed that specifications using mathematical integers not subject to over- or under-flow were the expected semantics and best understood by readers of specifications. Accordingly E-ACSL and JML allow specifications to be written using unbounded integers.

It is possible, using a dedicated arithmetic library (e.g., GMPZ for C code), to perform all numeric calculations using unbounded precision. However, this is not at all efficient compared to using machine integers. For example there is no need to rely on unbounded precision when handling the term `1 + 2`: addition over the C type `int` is sufficient. Similarly, if `c` is a C `char` variable, the `int` expression `c+1` will not overflow. The idea, introduced through a dedicated type system in [25] and implemented in E-ACSL, consists in tracking the range of possible values for each E-ACSL arithmetic term to determine what precision is needed to perform the computation. In practice, almost all integer arithmetic operations are computed with machine bounded integers thanks to this type system. It is worth noting that Adacore has adapted this idea to Spark2014.

This type system has recently been extended to logic functions and predicates in E-ACSL. For instance, the user can now define, say, the `sum` function `integer sum(integer a, integer b) = a + b`. If the only call to `sum` is done through `sum(1, 2)` then the prototype of the corresponding generated C function is `long __gen_e_acsl_sum(int a, int b)` (on a standard 64-bit architecture) and so only relies on bounded machine integers. We present in the following a few points on that work that are worth mentioning.

Recursive functions Recursive functions need special attention. For example, consider the function `f` presented in Fig. 5. We need to generate multiple prototypes for `f`. Indeed if it is called with an argument that fits into `int` (resp. `long`, `mpz`) then we will generate the C function of prototype `int __gen_e_acsl_f1(int n)` (resp. `int __gen_e_acsl_f2(long n)`, `int __gen_e_acsl_f3(mpz n)`). The return type of the different prototypes is `int` since `f` always returns 1 for any possible value of its argument.

```
1 /*@ logic integer f(integer n) =
2     n < 0 ? 1 : f(n - 1) * f(n - 2) / f(n - 3); */
3 int __gen_e_acsl_f1(int __gen_e_acsl_n);
4 int __gen_e_acsl_f2(long __gen_e_acsl_n);
5 int __gen_e_acsl_f3(mpz __gen_e_acsl_n);
```

Fig. 5. A logic function f defined in ACSL and declarations of three corresponding C functions depending on the value of n. The most precise interval within which f ranges is $I = [1; 1]$. Thus f will be typed into `int` in the best scenario.

Now, *automatically* computing the most precise range of E-ACSL's recursive functions *in the general case* is extremely hard, if not impossible, considering the expressiveness of the language. For example, even for the very restricted subset of E-ACSL where we only consider linear expressions, the problem is already NP. Fortunately, our concern is not to perform the utmost precise interval inference, but rather to perform an interval inference that is only precise enough so that we do not call bignum libraries.

To achieve that, we consider an over-approximation of the problem, that we express in a system of interval equations. For example, the system we build for the above presented `f` is $X_1 = [1; 1] \cup X_2 \cdot X_2 / X_2 \bigwedge X_2 = [1; 1] \cup X_3 \cdot X_3 / X_3 \bigwedge X_3 = [1; 1] \cup X_3 \cdot X_3 / X_3$ where X_1 (resp. X_2, X_3) denotes the interval over which ranges `__gen_e_acsl_f1` (resp. `__gen_e_acsl_f2`, `__gen_e_acsl_f3`). Our current way of solving those systems is such that we can infer that `f` ranges over $[-10^3; 10^3]$. Though $[-10^3; 10^3]$ is much wider than the optimal range $[1; 1]$, it still lies within `int`: E-ACSL is able to determine that `int` is sufficient as return type of f.

Termination We point out that the interval inference process is independent of the function's termination. Indeed we always obtain an interval, $[-\infty; +\infty]$ in the worst case, in a finite amount of time whether the function terminates or not. This raises the following question: how should we treat non-terminating recursive functions? The user will most likely not appreciate having his analysis stuck in some infinite recursion. Unfortunately, it is impossible to check termination of E-ACSL logic functions in the general case. This is, once again, due to its expressiveness.

We could perform (incomplete) termination analysis before translating and emit a warning every time termination is not guaranteed. However there is a solution that completely ensures that no non-terminating function is defined. This solution requires the user to provide a ranking function for each recursive function definition. This can be achieved by syntactically forcing each E-ACSL recursive function definition to have an attached **decreases** clause. Such clauses already exist for specifying termination of (recursive) C functions.

A `decreases` clause takes some quantity Q that is *supposed* to strictly decrease at *each* function call until reaching a minimum value, thus guaranteeing termination. During RAC, if Q is indeed decreasing then the function runs normally. Otherwise the execution is aborted with the indication that Q failed to

decrease: the user needs to provide another ranking function candidate and/or check whether the function is actually non-terminating. Support for `decreases` clauses is a future work for E-ACSL and is complicated by the possibility of defining arbitrarily large groups of mutually recursive functions.

4.3 On Support for Real Numbers

RAC is especially well-suited for verifying programs that manipulate Floating-Point (FP) numbers. FP computations are affected by tricky rounding behaviors. For example the Java equality `0.2f == 0.19999999925494194030761718 75f` counter-intuitively evaluates to `true`. These behaviors render verification extremely hard. The father of modern FP computation himself, Kahan, called out for "desperately needed remedies for the undebuggability of large floating-point computations in science and engineering" in 2011[5]. Still, industrial programs that deal with FP numbers, including those from critical industries, perform very complex numeric computations.

For the time being, DV still needs maturation before being able to handle large FP computations. This is mainly due to the fact that SMT solvers do not scale for large FP formulas. Until then, we propose RAC to come to the rescue. For verifying numeric programs written in C, we propose to write the specifications in standard mathematics and let E-ACSL, coupled with a few other tools, check them at runtime. By standard mathematics we mean real semantics for which operations are error-free. One advantage of our approach is that it can be used by users that are not familiar with FP computations, which is the case for "95% of folks out there" as jokingly said by the father of Java, Gosling[6], as long as they remember their high school mathematics.

The role of E-ACSL is to generate the sequence of calls to the real-arithmetic functions for all the real operations found in the specifications. In particular, special care needs to be taken to make sure that support for reals is well integrated with the interval inference and the type system for integers discussed in the previous section. Moreover, we can minimize calls to specialized libraries for real arithmetic in the same way as is accomplished for integers, as illustrated in Figure 6. However, such optimizations are not yet implemented. Plus, we have only implemented support for rationals for the time being. Indeed, we only use FP numbers and the operations $+ - */$ in our current test cases. Thus, though we can have complex use cases within such a setting, such as inversions by LU decomposition, computations can still be done with rational numbers.

Superset of $\mathbb{Q}$. Now what if we go beyond rational numbers? As a motivating example, positioning systems use trigonometric functions ubiquitously, say for computing distances and angles in polar coordinates[7]. First, we point out that any sound and fully automatic runtime checking of numeric properties expressed in a specification language as rich as E-ACSL is doomed to be incomplete

[5] `https://people.eecs.berkeley.edu/~wkahan/Boulder.pdf`
[6] `https://people.eecs.berkeley.edu/~wkahan/JAVAhurt.pdf`, p. 4
[7] For example at NASA, some containment algorithms are verified at runtime [38].

```
1 //@ assert \let real r = (float)(0.1f + 0.2f); ...
2 __e_acsl_real __gen_e_acsl_1 = __e_acsl_real_of(0.1f);
3 __e_acsl_real __gen_e_acsl_2 = __e_acsl_real_of(0.2f);
4 __e_acsl_real __gen_e_acsl_add = __e_acsl_real_add(__gen_e_acsl_1,
5                                                    __gen_e_acsl_2);
6 float __gen_e_acsl_r = __e_acsl_real_to_float(__gen_e_acsl_add);
7 ...
```

Fig. 6. An unoptimized translation. However, by taking advantage of the fact that correct rounding is guaranteed for FP addition, as required by IEEE-754, we could simply generate `float __gen_e_acsl_r = 0.1f + 0.2f;`

in the general case. This is because of the undecidability of equality between computable numbers [39][8]. Since we want neither to sacrifice soundness nor to resort to non-automatic solutions, there are only two remaining options:

1. restrict the supported constructs in a way such that decidability is preserved. Unfortunately, the obtained restriction would be too limited to be of interest. Indeed Richardson's theorem [32] prevents us from going beyond the set $\mathbb{Q} \cup \{\pi\}$ and the operations $+, *, -, /, sin$.
2. be incomplete, that is allow the emission of I Don't Know at runtime.

In practice, libraries for exact real arithmetic already exist [23][9]. In particular, we could use iRRAM [30], which provides support for algebraic functions such as square root, transcendentals such as exponential and logarithm, and an extensive set of trigonometric functions. Unfortunately iRRAM may not terminate when comparing two reals that are equal. The simplest solution to that would be to stop iRRAM when it takes too much time, in which case we should return I Don't Know. We expect that simple stopping criterion to give relatively satisfying performance since comparison between two equal reals does not happen often. However this is yet to be supported by experimental evidence which we leave as future work.

5 Related work

There are large bodies of related work on both deductive verification and runtime assertion checking, which we will not enumerate here. Our principal concern is systems that seek to integrate these tools with others in a common architecture and with well-defined specification semantics. Some of the systems we have already referenced are in this category.

[8] This theoretical limitation says that there is no *terminating* algorithm that can decide, statically through DV or at runtime through RAC, the equality relation between any pair of numbers that can be computed by Turing machines.

[9] see also sections "Software Using MPFR" and "Other Related Free Software" on MPFR's webpage, `https://www.mpfr.org/`

- The Eiffel language [29] pioneered specifications integrated with programs. Its first intent was RAC, with specifications limited to being executable, and later has been adding proof capability.
- Ada has a well-supported commercial companion tool, SPARK, which also supports both DV and RAC using an integrated architecture. Like ACSL, many contracts are executable, but non-executable assertions can be written.
- Spec# [4] was built as an extension of C# with non-null types as well as contracts, enforced by Eiffel-like run-time checks and by a static program verifier. En route to market, it became CodeContracts [20], a simplified and language-agnostic specification language for .NET with a tool to insert run-time checks and with a static analyzer to check certain properties statically.

Dafny [28], mentioned above, is designed to ensure that all assertions are statically verified as part of compilation, so no assertions are compiled into the code. It has mechanisms for adding detailed proof steps if necessary to verify difficult assertions.

Regarding the verification of numeric properties, as discussed in §4 for the particular case of RAC, the authors of [18] present a sound way of compiling specifications written in real semantics into programs that can be executed on machines with finite amount of memory (Scala programs). From a conceptual point of view, the main difference with our work is that they require a tolerance as well as the target precision to be explicitly stated. In contrast to that, we want our compiled programs to be 100% accurate, no less. Moreover, we let our tool decide the precision (the types) within which the different computations must be done.

6 Combining DV and RAC

DV and RAC tools working against a common target software and its specifications can be usefully used together. One helpful workflow is the following. When the target software is partially written, not yet provable, but executable and has some specifications, RAC can be run to check that the specifications are valid for the set of unit test cases. It takes more effort to actually prove (including to debug) the software and specifications with DV, so RAC is used to perform quick initial checks of the specifications. Once RAC has shown the software and specifications to be largely correct, then DV can be used more efficiently to verify that the combination of software and specifications is indeed consistent for all possible program inputs.

Similarly, suppose DV is not able to prove a set of specifications and produces a counterexample, which is a set of program inputs that DV cannot prove to obey the specifications. Then the engineer (or tools) can create an executable version of the counterexample to be run with RAC; RAC will then pinpoint which assertions are failing, identifying the incorrect software or the misconceptions in the specifications [31].

Furthermore, when combining DV and testing with RAC, it may be hard to know whether all the pieces of code and all the specifications are covered

by the verification campaign. Here the combination of RAC+runtime coverage measurements and DV+ tools that check specification coverage can be enhanced by new tools [27] that provide unified coverage criteria for both runtime testing and static proof. The cited paper also provides means to avoid verifying the same functions redundantly with both DV and RAC.

7 Future Work

While the architectures we described are fairly well implemented, the principal areas for improvement are these:

- Improvements in both the usability of specification languages and the clarity of their semantics
- Reducing the gap between what can be checked at runtime and what can be checked deductively. The biggest challenges for RAC are checking memory properties, complex properties over real numbers and checking axiomatically-stated specifications. The challenges for DV are specification in the presence of abstraction and refinement, information hiding, and effective performance of solvers with quantified expressions.
- Reducing the time- and memory-overhead of runtime checking, extending the areas discussed in this paper.

One of the gaps to be filled for both static and runtime checking is modeling the semantics of concurrency. Concurrency is a challenge for runtime checking because the runtime assertion instrumentation can change the timing of portions of the program and so change what races or deadlocks might occur. Even worse, concurrent accesses to E-ACSL's shadow memory model may lead to incorrectness. Concurrency is a challenge for static verification because of its complexity: one must model and check all possible interleavings of concurrent threads, along with the appropriate memory model. Neither ACSL nor JML currently models concurrency. However, the closed-source prototype plug-in Mthread at CEA relies on Eva to automatically detect unsafe concurrent accesses to shared variables. Another Frama-C plug-in[10], developed by Adelard on a quite old version of Frama-C, shares the same goal.

This paper has discussed two tools: Frama-C for C/ACSL and OpenJML for Java/JML. At present these are independent tools, sharing common history, specification language concepts and implementation techniques, but no common software or intermediate representations. It is, of course, possible for this situation to be different. Indeed, Frama-C is already evolving to support C++. One can envision a Frama-X framework whose internal representation of software and specifications is general enough to accommodate multiple modern programming languages. Each programming language would have a front-end and a specialized specification language as similar as possible to specification languages for other supported programming languages. All the languages would then use a common back-end that created the logical encoding of the software+specifications and managed the proof environment.

[10] `https://bitbucket.org/adelard/simple-concurrency`

8 Conclusion

Practical, sound runtime assertion checking depends on two characteristics. First, the translation of programming language source code and specification language assertions must be correct, according to a well-understood semantics. This correctness is best achieved when the RAC tool shares an intermediate form with other tools also needing correct semantic translations. Here we have presented static deductive verification as one such tool, but a few other possibilities are model checkers, test generation, and specification inference. A shared infrastructure reduces the implementation work and increases the semantic conformance among tools.

The second needed characteristic is practicality: the RAC tool must be able to check as many kinds of assertions as possible and do so as efficiently as possible. In this paper we presented advances in three areas — memory properties and integer and real computations — demonstrating that implementations of RAC are continuously improving and that RAC is a viable and useful element in a suite of program analysis tools, at times in ways that cannot yet be achieved by deductive verification.

Acknowledgements. This work is done in the context of project VESSEDIA, which has received funding from the European Union's 2020 Research and Innovation Program under grant agreement No. 731453.

References

1. Jean-Raymond Abrial, A. Hoare, and Pierre Chapron. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.
3. John Barnes. *Spark: The Proven Approach to High Integrity Software*. Altran Praxis, http://www.altran.co.uk, UK, 2012.
4. Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and Verification: The Spec# Experience. *Communications of the ACM*, 54(6):81–91, June 2011.
5. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.
6. Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. *Introduction to Runtime Verification*, pages 1–33. Springer International Publishing, Cham, 2018.
7. Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language.
8. Sandrine Blazy, David Bühler, and Boris Yakobowski. Structuring Abstract Interpreters through State and Value Abstractions. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'17)*, 2017.

9. Abderrahmane Brahmi, David Delmas, Mohamed Habib Essoussi, Famantanantsoa Randimbivololona, Abdellatif Atki, and Thomas Marie. Formalise to automate: deployment of a safe and cost-efficient process for avionics software. In *Embedded Real-Time Software and Systems (ERTS^2'18)*, January 2018.
10. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.
11. Patrice Chalin. Logical foundations of program assertions: What do practitioners want? In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 383–393, 2005.
12. Patrice Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *International Conference on Software Engineering (ICSE'07)*, pages 23–33, May 2007.
13. David R. Cok. Improved usability and performance of SMT solvers for debugging specifications. *STTT*, 12:467–481, 2010.
14. David R. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In Mihaela Bobaru, Klaus Havelund, GerardJ. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*. 2011.
15. David R. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In *Workshop on Formal Integrated Development Environment (F-IDE 2014)*, volume 149 of *EPTCS*, pages 79–92, 2014/04/06/ 2014. Grenoble, France.
16. David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.
17. Loc Correnson and Julien Signoles. Combining Analyses for C Program Verification. In Marille Stoelinga and Ralf Pinger, editors, *Formal Methods for Industrial Case Studies (FMICS'12)*, volume 7437 of *Lecture Notes in Computer Science*, pages 108–130. Springer, August 2012.
18. Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 235–248, 2014.
19. Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. Common specification language for static and dynamic analysis of C programs. In *Symposium on Applied Computing (SAC'13)*, March 2013.
20. Manuel Fähndrich, Michael Barnett, Daan Leijen, and Francesco Logozzo. Integrating a set of contract checking tools into visual studio. In *TOPI@ICSE*, 2012.
21. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where Programs Meet Provers. In *European Symposium on Programming (ESOP'13)*, March 2013.
22. Stephen J. Garland and John V. Guttag. A guide to lp, the larch prover. Technical Report 82, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991. Order from src-report@src.dec.com.
23. Paul Gowland and David R. Lester. A survey of exact arithmetic implementations. In *Computability and Complexity in Analysis, 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers*, pages 30–47, 2000.

24. John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01, University of Central Florida, School of EECS, Orlando, FL, March 2009.
25. Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. Rester statique pour devenir plus rapide, plus précis et plus mince (French). In *Journes Francophones des Langages Applicatifs, JFLA 2015*, January 2015. In French.
26. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing*, pages 1–37, January 2015.
27. Viet Hoang Le, Loïc Correnson, Julien Signoles, and Virginie Wiels. Verification coverage for combining test and proof. In *International Conference on Tests and Proofs (TAP'18)*, June 2018.
28. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10. Springer-Verlag, 2010.
29. Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, 1988.
30. Norbert Th. Müller. The iRRAM: Exact arithmetic in C++. In *Computability and Complexity in Analysis, 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers*, pages 222–252, 2000.
31. Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. Your Proof Fails? Testing Helps to Find the Reason. In *International Conference on Tests and Proofs (TAP'16)*, July 2016.
32. Daniel Richardson. Some undecidable problems involving elementary functions of a real variable. *Journal of Symbolic Logic*, 33(4):514–520, 1968.
33. Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Annual Technical Conference (ATC'12)*, June 2012.
34. Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-precision. In *Annual Technical Conference (ATC'05)*, April 2005.
35. Julien Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language*. `http://frama-c.com/download/e-acsl/e-acsl.pdf`.
36. Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. 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'17)*, September 2017.
37. J. M. Spivey. *The Z notation: A Reference Manual*. Prentice Hall International (UK) Ltd., 1992.
38. Laura Titolo, Cesar A. Muñoz, Marco A. Feliu, and Mariano M. Moscato. Eliminating Unstable Tests in Floating-Point Programs. *ArXiv e-prints*, August 2018. To appear in the proceedings of LOPSTR'18.
39. Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265, 1937.
40. Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles. Detection of Security Vulnerabilities in C Code using Runtime Verification. In *International Conference on Tests and Proofs (TAP'18)*, June 2018.
41. Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov. Shadow State Encoding for Efficient Monitoring of Block-level Properties. In *International Symposium on Memory Management (ISMM'17)*, pages 47–58, June 2017.
42. OpenJDK: `http://www.openjdk.org`.