Focused Certification of an Industrial Compilation and Static Verification Toolchain *

Zhi Zhang[1], Robby[1], John Hatcliff[1], Yannick Moy[2], Pierre Courtieu[3]

[1] Kansas State University
{zhangzhi,robby,hatcliff}@ksu.edu
[2] AdaCore
moy@adacore.com
[3] Conservatoire National des Arts et Métiers
pierre.courtieu@cnam.fr

Abstract. SPARK 2014 is a subset of the Ada 2012 programming language that is supported by the GNAT compilation toolchain and multiple open source static analysis and verification tools. These tools can be used to verify that a SPARK 2014 program does not raise language-defined run-time exceptions and that it complies with formal specifications expressed as subprogram contracts. The results of analyses at source code level are valid for the final executable only if it can be shown that compilation/verification tools comply with a common deterministic programming language semantics.
In this paper, we present: (a) a mechanized formal semantics for a large subset of SPARK 2014, (b) an architecture for creating certified/certifying analysis and verification tools for SPARK, and (c) tools and mechanized proofs that instantiate that architecture to demonstrate that SPARK-relevant Ada run-time checks inserted by the GNAT compiler are correct; this includes mechanized proofs of correctness for abstract interpretation-based static analyses that are used to certify correctness of GNAT run-time check optimizations.
A by-product of this work is a substantial amount of open source infrastructure that others in academia and industry can use to develop mechanized semantics, and mechanically verified correctness proofs for analyzers/verifiers for realistic programming languages.

1 Introduction

SPARK is a subset of the Ada programming language targeted at safety- and security-critical applications. It builds on the strengths of Ada for creating highly reliable and long-lived software. SPARK restrictions ensure that the behavior of a SPARK program is unambiguously defined and simple enough that formal verification tools can automatically check the conformance of a program to its software-contract-based specification. The SPARK language and toolset for formal verification have been applied over many years to on-board aircraft systems, control systems, cryptographic systems, and rail systems [1, 15]. The latest version – SPARK 2014 [12], builds on the new specification features added in Ada 2012 [2]. One consequence of the new specification foundation is that SPARK contracts are no longer phrased in Ada comments understood only by the SPARK tools, but the formal specifications are phrased in Ada 2012 metadata constructs that can be understood by a much wider class of tools (including the GNAT

* This material is based upon work supported by the US Air Force Office of Scientific Research (AFOSR) under contract FA9550-09-1-0138.

compiler) and they have an execution semantics. The definition of the SPARK 2014 language subset is motivated by the simplicity and feasibility of formal analysis and the need for an unambiguous semantics.

Static analysis tools are available that provide flow analysis, symbolic execution and proof of SPARK programs. The industrial tool GNATprove[4] co-developed by Altran and AdaCore performs flow analysis to check correct access to data in the program (correct access to global variables as specified in data and information flow contracts and correct access to initialized data) and uses deductive methods to demonstrate that the program is free from run-time errors and that the specified contracts are correctly implemented. The academic tool Bakar Kiasan [3] developed by Kansas State University allows executing symbolically a SPARK program with or without contracts, to detect possible run-time errors and contract violations, and in some cases also prove that no such errors can occur.

Motivations: A major reason for using SPARK for developing critical software is the ability to prove statically that no language specified run-time errors, such as arithmetic overflow, buffer overflow and division-by-zero, can occur.[5] Besides the additional confidence in the software that this result brings, it can be used in some certification domains to lower the verification effort in some other areas like testing. For example, the most recent version DO-178C of the avionics certification standard allow using both tests or proofs as acceptable verification methods [13]. It is also commonly used as an argument to justify the suppression of run-time checks in the final executable, typically for increasing execution speed.

The absence of run-time errors can be guaranteed only relative to the correctness of the compiler and analyzers used. Although correctness is not proved for tools used in practice on typical industrial projects, there is a special process known as tool qualification in safety-critical industry which aims at giving sufficient confidence that the tools behave correctly [9].

A critical element for the qualification of both the GNAT compiler (the most widely used Ada compiler) and the GNATprove analyzer, both developed by AdaCore, is that they correctly interpret the semantics of SPARK with respect to the placement of run-time checks. The compiler works by producing first a semantically analyzed abstract syntax tree (AST) of the program, decorated with flags that indicate positions in the AST where run-time checks should be inserted. This AST is then expanded into a lower level representation with explicit run-time checking code. The input of GNATprove is based on the same AST used for compilation (using the same compilation *front-end*), decorated with the same flags that, in this case, indicate where absence of a particular run-time error should be proved. Because the compiler and the analyzer share the code that inserts decorations in the AST, this code is much less likely to miss checks, and some effort has been invested in optimizing out useless checks.

Hence, the compiler and analyzers all share the AST produced by the front-end, with decorations indicating where run-time checks should be inserted. However, we have discovered various situations where decorations were missing, which ultimately led to a correction of the GNAT front-end. The last such occasion was the implemen-

[4] `http://www.adacore.com/sparkpro`

[5] *Language-specified* run-time errors that are relevant for all programs in the language can be contrasted with *application-specific* run-time errors that correspond to violations of a program's application-specific requirements. Our work addresses the former notion.

tation of a Tetris game in SPARK for a demo at a customer gathering[6]: after proving that the program was free of run-time errors, the first test on the actual board stopped unexpectedly due to a range check failing during execution. There was indeed a possible check failure in the code (later corrected) on a new attribute recently introduced in SPARK, which was not detected during proof because the corresponding decoration was not set by GNAT.

Thus, it is of critical importance to be able to guarantee that all check decorations are set on the AST produced by the front-end, as defined in SPARK 2014 language reference semantics. That is, instead of assuming that GNAT correctly decorates AST with the required run-time checks, this paper presents our work to ensure that is indeed the case.

Contributions: To address the issues described above, and to enable a long-term research program investigating the use of mechanized semantics and proofs for SPARK 2014[7], we have developed multiple proof infrastructure components in the Coq proof assistant. We have created *certified*[8] SPARK run-time check generators with a small trust-base footprint that can be used to substantiate the correctness of the industrial SPARK 2014 toolchain. Our contributions include:

- The formalization of the language (dynamic/evaluation) semantics for a core subset of the SPARK 2014 language using the Coq proof assistant [20] (Section 2.1). The core language subset includes scalar subtypes and derived types, array types, record types, procedure calls, and locally defined subprograms; a large class of programs can be desugared to this core subset, thus, enabling evaluations on realistic SPARK systems to some extent. The formal semantics specification represents our trust-base (along with Coq, which itself has been highly-regarded as a proof system that has a smaller trust-base compared to others); the specification is trustable because, for example, it has been manually inspected by leading experts in SPARK/Ada both in industry and academia. Hence, it can be considered as *the* reference SPARK 2014 formal semantics.
- An implementation of a certified run-time check generator for the core language (Section 2.2); that is, the implementation is proved to be *consistent* with the reference semantics with respect to the class of errors that can arise in the language subset (such as overflow checks, range checks, array index checks and division by zero checks). The consistency guarantees that if language-defined run-time checks generated by the certified implementation do not fail, a program cannot "go wrong" according to the SPARK formal semantics. The generated checks by the implementation represents the baseline as the most conservative run-time check set (i.e., a larger set is unnecessary and could even be problematic).
- An implementation of a certified run-time check optimizer (Section 2.3). The optimization is needed because the GNAT frontend employs various optimizations to reduce the set of run-time checks that it generates for run-time efficiency sake. The certified optimizer uses an abstract interpretation-based [7] interval analysis; it

[6] `http://blog.adacore.com/tetris-in-spark-on-arm-cortex-m4`

[7] By "mechanized", we mean the construction of formal definitions (of semantics, translations, analysis, etc.) and formal proofs of associated properties in a proof assistant that enables correctness to be checked automatically by the proof assistant.

[8] Certified here means that there are formal mathematical artifacts (such as machine-checked proofs) that serve as rigorous evidence that an implementation is consistent with its specification [5].

generates a smaller set of run-time checks compared to the ones produced by the GNAT frontend, while still being consistent.
- An implementation of a conformance checker as a back-end of the GNAT front-end (including, e.g., a SPARK program translator to fully resolved SPARK ASTs in Coq) that automates evaluations of the GNAT frontend against the certified run-time check generators (Section 2.4). This essentially turns the industrial GNAT frontend into a certifying[9] tool with respect to introduction of run-time error check decorations. This increases the confidence in the GNAT compiler back-end that embeds run-time assertion checking when it emits machine code for testing, as well as in the GNATprove verifier that uses the run-time check decorations to determine what verification conditions to generate.
- The evaluation of the GNAT front-end against both the certified run-time check generators (Section 2.4). We evaluated that: (1) the set of run-time check decorations inserted by the GNAT front-end is in fact a subset of the decorations generated by the unoptimized run-time check generator, while (2) it is a superset of the decorations generated by the optimized run-time check generator. In addition to confirming the correctness of the GNAT front-end run-time check decoration generator, the evaluation exposed some subtle differences in the run-time checks generated by the GNAT frontend, as well as exposing further optimizations that can be done by the frontend while still preserving its correctness property.

While our research work includes making an industrial impact on SPARK run-time error checking, the significant investments reflected in our contributions (e.g., over 25,000 lines of Coq proofs and reusable AST, translations, and semantics infrastructure), enable much broader research and engineering. All of the Coq artifacts and associated tools created in this work are publicly available under an EPL open source license[10]. The formalized SPARK semantics and certified run-time check generators can be leveraged to develop certified/certifying program analyzers (e.g., a contract verifier) and translators (e.g., a SPARK to CompCert [19] intermediate representation translator to benefit from CompCert's certified translation toolchain down to machine-code level). In general, our approach of using both unoptimized and optimized certified run-time check generators to turn an untrusted (industrial) implementation into a certifying tool can be adopted to other programming languages/development tools for critical systems that ensure absence of run-time errors.

2 Technical Approach

Figure 1 gives an architectural overview of our approach; the subsequent sub-sections describe each of the components. Due to space constraints, we only highlight some limited language features sufficient to illustrate our approach (see the publicly available artifacts for the complete definitions).

2.1 Core SPARK 2014 Mechanized Semantics

As stated previously, one main component of our approach is a formal language reference semantics of core SPARK 2014 mechanized using Coq [20]; Coq allows for

[9] Certifying here means that the tool generates evidence testifying that it is in fact consistent with its specification for a particular use of the tool.

[10] `http://santoslab.org/pub/TR/SAnToS-TR2016-03-11/`

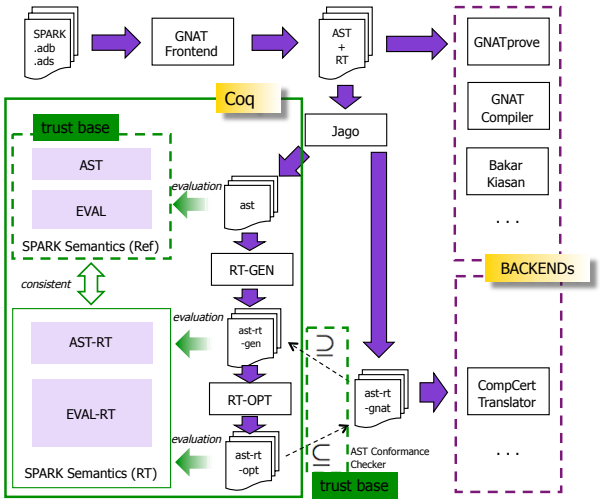

Fig. 1. Architectural Overview

specifying, implementing, and proving programming language related properties. We chose Coq due to the fact that it has a relatively small core which has been vetted by many experts in the programming language community (small trust-base).

The core language includes features typically found in imperative languages such as arrays, records, and procedure calls, as well as SPARK-specific structures, such as nested procedures and subtypes. One major difference between SPARK and other programming languages (e.g., C) is that verification for absence of run-time errors is required by the semantics of the language itself. Thus, the constraints associated with the required run-time checks are specified within the operational semantics rules for the language; that is, as the rules are used to "evaluate" the program, the semantics of the run-time checks are enforced at appropriate points in the program, and the evaluation will terminate with a run-time error message as soon as one of its run-time checks fails.

The formalization includes: (a) a SPARK AST representation (symbols and types are fully resolved), and (b) a rule-based "big step" operational semantics for SPARK (including state/value representations, expression evaluation, and statement execution).

SPARK AST Representation: SPARK ASTs are represented using inductive type definitions in Coq, where each constructor takes as an argument a number used to uniquely identify and reference the particular AST node being constructed. The AST numbers are useful as keys for symbol tables, type tables, and mappings from source code line/-column positions. The following is an excerpt of the inductive definition for expressions

```
Inductive exp: Type :=  | BinOp: astnum→ binOp→ exp→ exp→ exp | ...
```

where `BinOp` is the constructor for binary expressions that takes as arguments a unique AST number `astnum`, a symbol for the particular operator being used (*e.g.*, `Add`, `Sub`, `Mul`, *etc.*.), and the expressions for the left and right arguments to the operator. The last `exp` is the resulting type of the constructor (*i.e.*, the constructor is building an `exp`).

SPARK supports range constrained integer types that are useful as array index types; that is, the range constraints are used to determine in-bounds/out-of-bounds array operations (instead of using a special `.length` field such as in Java). Range constrained types can be declared using either a subtype declaration (e.g., **`subtype`** `T10` **`is`** `Integer` **`range`** `1 .. 10`), a derived type definition (e.g., **`type`** `U10` **`is new`** `Integer` **`range`** `1 .. 10`), or an integer type definition (e.g., **`type`** `W10` **`is range`** `1 .. 10`); they

semantically differ in that the last two introduce a new type, while the first one does not; the differences have to be taken into account in the formalization. This illustrates the non-trivial number of language features that one has to cover when formalizing a real programming language that can be directly leveraged for developing high-integrity industrial tools.

State/Value: Due to the semantics enforcing the run-time checks, evaluating either an expression or a statement may produce an error state when the run-time check fails (otherwise a value or a state is produced, respectively). The following definition defines a generic return type `Ret`:

```
Inductive Ret (A: Type): Type :=  | OK: A→ Ret A | RTE: errorType→ Ret A.
```

Type parameter `A` is either the value/state type, and `errorType` is the run-time error state type (e.g., division by zero, overflow, out of range).

Expression Semantics: The mechanized big-step operational semantics definition consists of an inductively defined type with constructors corresponding to the individual "rules" of the semantics. Intuitively, each constructor takes as arguments: (a) Coq objects representing operational semantics derivations corresponding to evaluation of expression/statement sub-components, and (b) Coq objects representing derivations establishing that some necessary "side conditions" hold. Each expression rule relates a symbol table, the current state, and the expression to be evaluated, to a return value. The expression rules are complicated by the fact that sub-expression evaluation can produce an error state, as follows.

```
Inductive evalExp: symTab→ state→ exp→ Ret value→ Prop :=
| EvalBinOpE1_RTE: ∀ st s e1 msg n op e2,     (* e1 returns error *)
   evalExp st s e1 (RTE msg)→ evalExp st s (BinOp n op e1 e2) (RTE msg)
| EvalBinOpE2_RTE: ∀ st s e1 v1 e2 msg n op, (* e2 returns error *)
   evalExp st s e1 (OK v1)→ evalExp st s e2 (RTE msg)→
     evalExp st s (BinOp n op e1 e2) (RTE msg)
| EvalBinOp: ∀ st s e1 v1 e2 v2 op v n,       (* no error from e1 & e2 *)
   evalExp st s e1 (OK v1)→ evalExp st s e2 (OK v2)→ evalBinOp op v1 v2 v→
     evalExp st s (BinOp n op e1 e2) v
...
Inductive evalBinOp: binOp→ value→ value→ Ret value→ Prop :=
| CheckBinops: ∀ op v1 v2 v v', (* binop results in overflow *)
   op = Add ∨ op = Sub ∨ op = Mul→ Denotational.binOp op v1 v2 = Some (Int v)→
     overflowCheck v v'→  evalBinOp op v1 v2 v'
| CheckDivRTE: ∀ v1 v2,         (* check for div by zero *)
   divCheck v1 v2 (RTE DivByZero)→ evalBinOp Div (Int v1) (Int v2) (RTE DivByZero)
| CheckDiv: ∀ v1 v2 v v',       (* no div by zero, check result overflow *)
   divCheck v1 v2 (OK (Int v))→ overflowCheck v v'→ evalBinOp Div (Int v1) (Int v2) v'
...
```

`EvalBinOpE1_RTE` specifies the evaluation of a binary expression (`e1 op e2`) where the evaluation of `e1` produces an error state (similarly, `EvalBinOpE2_RTE` for when `e2` fails). `EvalBinOp` specifies the situation where evaluations of both `e1` and `e2` produce operand values, which are then evaluated using `evalBinOp`; `evalBinOp` incorporates various run-time checks such as division by zero and overflow/underflow by using `divCheck` and `overflowCheck`; `divCheck` produces a value if the second operand is non-zero (otherwise, it produces the error state `RTE DivByZero`), and `overflowCheck` produces a value if the given value fits within the (platform) integer type value range (otherwise, it produces `RTE Overflow`). Run-time checks for other language features such as array indexing are specified in the same spirit as the above.

Statement Semantics: Range checks are enforced during statement executions of, for example, assignments and procedure calls. We describe the intuition behind statement semantic rules using examples instead of verbosely listing the Coq specifications.

For an assignment, a range check is enforced for its right hand side expression if the left hand side expression's type is a range constrained type. For example,

```
subtype MyInt is Integer range 1 .. 10;   X: MyInt;   ...;   X := X + 1;
```

That is, `X` is a variable of type `MyInt`, which is defined as a subtype of `Integer` ranging from 1 to 10. The assignment increments `X` by 1, as follows. First, `X + 1` is evaluated; if it returns a value (instead of an error state), the value is checked against the range of `MyInt` before updating `X`.

For a procedure call, range checks are required for both input arguments and output parameters if the types of input parameters and output arguments are range constrained types because input arguments are assigned to the procedure input parameters, and output parameters are assigned to the output arguments.

In general, there are three categories of run-time checks in the core SPARK subset: (1) overflow (including underflow) run-time checks (for integer arithmetic operations), (2) division by zero run-time checks (for modulus and division operations), and (3) range run-time checks (for integer variable assignments, array assignments, array accesses, and procedure calls).

Evaluation: We designed the semantics rules including the specification of run-time checks by referring to the SPARK [21] and Ada [17] reference manuals. The rules were subsequently inspected and refined by various experts including SPARK/Ada designers and developers. We then proved that our SPARK mechanized semantics enjoys a form of type safety (Section 2.3), which guarantees, to some extent, its internal consistency.

2.2 Certified Run-Time Check Generator

Given a SPARK program, the GNAT compiler front-end builds the program fully-resolved (symbol/type) AST decorated with flags that indicate the position and nature of the run-time checks to be performed. When down-stream tools process the ASTs, they interpret/transform the decorations. For example, a later phase of the GNAT compiler replaces each decoration with an assertion AST representing code that implements the corresponding run-time check. In contrast, the Why3 [22]-based GNATprove verification tool uses the decorations to generate verification conditions. Both tools assume that the run-time check decorations inserted by the GNAT compiler front-end are correct.

To formally capture the notion of decorating ASTs with run-time check information, we implemented in Coq a run-time check decoration generator (RT-GEN in Figure 1) whose consistency with the mechanized SPARK reference semantics was established via a Coq proof. Hence, the correctness of RT-GEN is *certified*. To achieve this, a different set of operational semantic rules is needed (called EVAL-RT) – one that "evaluates" an AST with run-time check decorations and only enforces the checking semantics where a decoration occurs. Then, one can prove that, for any program and for any program initial state, EVAL-RT supplied with run-time check decorations generated by RT-GEN produces *exactly the same* state as EVAL (*i.e.*, the SPARK reference semantics).

EVAL-RT: is a modified EVAL that accepts AST-RT where run-time check decorations are represented as tree attributes. For example, AST-RT expression is defined as follows.

```
Inductive expRT: Type :=
| BinOpRT: astnum→ binOp→ expRT→ expRT→ interiorChecks→ exteriorChecks→ expRT
...
```

The difference from AST is that two additional fields `interiorChecks` and `exteriorChecks` are introduced; `interiorChecks` are intended for run-time checks associated with the

binary operator (e.g., addition requires `overflowCheck`), while `exteriorChecks` are checks associated with expression's context (e.g., if the expression is used for array indexing, then it should be range-checked against the array size). Once AST-RT is defined, one can then define EVAL-RT that accepts AST-RT and enforces the explicitly listed run-time checks (e.g., in `interiorChecks` and `exteriorChecks`), as illustrated below.

```
Inductive evalExpRT: symTabRT→ state→ expRT→ Ret value→ Prop :=
| EvalBinOpRT: ∀ st s e1 v1 e2 v2 ins op v n exs,
   evalExpRT st s e1 (OK v1)→ evalExpRT st s e2 (OK v2)→ (* no error on e1, e2 *)
   evalBinOpRTS ins op v1 v2 v→                          (* process RT checks *)
   evalExpRT st s (BinOpRT n op e1 e2 ins exs) v
...
```

`evalBinOpRTS` iterates over the run-time check decorations to enforce the `interiorChecks` for a binary expression. The binary operation is performed if none of the run-time checks produces an error state; otherwise, it returns the error state. (Note that enforcement of `exteriorChecks` is not presented above as it involves arrays, which is not presented due to space constraint.)

RT-GEN: translates AST to AST-RT. In developing RT-GEN, we first specified its behavior declaratively as a Coq inductively defined relation (e.g., `toExpRT` below) between AST to AST-RT (with the symbol table as an auxiliary component). Then, we implemented the translation as a Coq function (e.g., `toExpRTImpl`).

```
Inductive toExpRT: symTab→ exp→ expRT→ Prop :=
| ToBinOpO: ∀ st op e1 e1RT e2 e2RT n, (* insert overflow checks on op result *)
   op = Add ∨ op = Sub ∨ op = Mul → toExpRT st e1 e1RT → toExpRT st e2 e2RT →
   toExpRT st (BinOp n op e1 e2) (BinOpRT n op e1RT e2RT [OverflowCheck] nil)
| ToBinOpDO: ∀ st e1 e1RT e2 e2RT n,   (* Div: div by 0 + overflow *)
   toExpRT st e1 e1RT → toExpRT st e2 e2RT →
   toExpRT st (BinOp n Div e1 e2)
              (BinOpRT n Div e1RT e2RT [DivCheck, OverflowCheck] nil)
...
Function toExpRTImpl(st:symTab)(e:exp): expRT :=...
```

As can be observed, `ToBinOpO` specifies that RT-GEN should generate (interior) `OverflowCheck` for addition, substration, or multiplication, and both `DivCheck` and `OverflowCheck` for division; `toExpRT` is implemented by `toExpRTImpl` using Coq's programming language features (like ML's) which is extractable to OCaml to produce an executable.

Evaluation: To certify RT-GEN, we proved that its specification is consistent (*sound* and *complete*) with respect to the SPARK mechanized semantics. For example, for expressions, we proved the following consistency lemma:

```
Lemma toExpRTConsistent: ∀ e eRT st stRT s v,
  toExpRT st e eRT→ toSymTabRT st stRT→ (evalExpRT stRT s eRT v ↔ evalExp st s e v).
```

where `toSymTabRT` transforms `symTab` to `symTabRT`, which, among other things, maps procedure names to their AST-RT. We then proved that the RT-GEN implementation is consistent with respect to its specification, for example:

```
Lemma toExpRTImplConsistent: ∀ e eRT st, toExpRTImpl st e = eRT ↔ toExpRT st e eRT.
```

Therefore, the implementation is transitively consistent with respect to the SPARK semantics (by transitivity of implication $\rightarrow$ /$\leftrightarrow$).

2.3 Certified Run-Time Check Optimizer

While RT-GEN generates a sufficient set of run-time checks, some of them may not be necessary. In fact, the GNAT front-end uses optimization techniques to reduce the set of run-time checks that it generates; in practice, we expect the set generated by GNAT

to be a subset of the RT-GEN generated set (we confirmed through experiments that this is indeed the case in Section 2.4). The question is then whether GNAT's optimizations are (certifiably) sound. Our approach to answer this is to have a certified optimizer (RT-OPT) that reduces the run-time checks generated by RT-GEN. It is widely known that, in general, an optimizer cannot actually ever be optimal (due to the halting problem). Thus, the best we can hope for is to have RT-OPT reduce to the same (or even better, i.e., smaller) set as GNAT's (a smaller set implies that GNAT can be improved further); Section 2.4 confirms that this is indeed the case through validation.

RT-OPT: transforms AST-RT to another AST-RT by removing some run-time checks whose corresponding verification conditions (VCs) can be discharged; RT-OPT discharges the VCs by employing a (certified) abstract interpretation [7] analysis with interval numeric domain (Section 2.4 shows that RT-OPT is on par or better than GNAT's runtime-check optimizations). Similar to RT-GEN, we first specified RT-OPT as inductively defined relation and then implemented it as a function. For expressions, RT-OPT produces AST-RT along with the expression's interval domain (if any) as follows:

```
Inductive optExp: symTabRT→ expRT→ (expRT * interval)→ Prop = ...
Function optExpImpl(st:symTabRT) (e:expRT): option(expRT * interval) = ...
```

where `optExp` is typeset in Figure 2 for readability. One invariant of RT-OPT is that integer expression optimization should produce an interval that fits within the compilation target platform-specific two's complement integer range, which makes up the default interval $[INT_{MIN}, INT_{MAX}]$. Γ holds the abstract interpretation context such as symbol table, etc. For notational convenience, `interiorChecks` and `exteriorChecks` are not explicitly shown; *EraseOverflowCheck* and *EraseDivCheck* remove overflow and division `interiorChecks`, respectively.

The INT1 rule in Figure 2 optimizes away the overflow check in the case of an integer literal AST-RT n where n is within the platform's integer range; a single-value interval $[n, n]$ is returned along with the optimized AST-RT (i.e., the tight single-value interval allows for concrete interpretation). On the other hand, INT2 specifies the case where the overflow check is kept whenever n is outside the range, thus, the default interval is returned (in this case, an error message can be generated to notify the developer). ADD1 and ADD2 first try to optimize the two operands and compute the expression interval bounds (i.e., $[u, v]$). ADD1 specifies the case where the bounds are within the platform's integer range, hence, the overflow check associated the binary operation can be safely removed; otherwise, ADD2 specifies that run-time checks are preserved, and the resulting interval is the platform's integer range.

For division, four cases (DIV1-4) specify the different situations where division by zero and/or operation overflow (i.e., when dividing INT_{MIN} by -1) could occur; in all the cases, the resulting interval is specified by $divInterval$ that does case analysis on the positivity/negativity of the interval operands. For example, (5) specifies the case where both of the operand intervals $[v_1, w_1]$ and $[v_2, w_2]$ are positive (i.e, the low bounds v_1 and v_2 are positive); in this case, the resulting interval is $[v_1/w_2, w_1/v_2]$ where its low bound v_1/w_2 is computed by dividing the smallest value of the first operand's interval with the largest value of the second operand's interval, and its high bound w_1/v_2 is computed by dividing the largest value of the first operand's interval with the smallest value of the second operand's interval. The $divInterval$ specification illustrates a slice of the RT-OPT's complexity for computing tight intervals in order to optimize away many run-time checks; rest assured however that they are proven to be correct in Coq.

$$\frac{n \in [INT_{MIN}, INT_{MAX}]}{\Gamma \vdash optExp(n) = (EraseOverflowCheck(n),\ [n, n])}\ \textsc{Int1} \qquad \frac{n \notin [INT_{MIN},\ INT_{MAX}]}{\Gamma \vdash optExp(n) = (n,\ [INT_{MIN},\ INT_{MAX}])}\ \textsc{Int2}$$

$$\frac{\Gamma \vdash optExp(e_i) = (e'_i, [v_i, w_i]),\ i \in \{1,2\} \quad v = v_1 + v_2 \quad w = w_1 + w_2 \quad \{v,w\} \subseteq [INT_{MIN}, INT_{MAX}]}{\Gamma \vdash optExp(e_1 + e_2) = (EraseOverflowCheck(e'_1 + e'_2),\ [v, w])}\ \textsc{Add1}$$

$$\frac{\Gamma \vdash optExp(e_i) = (e'_i, [v_i, w_i]),\ i \in \{1,2\} \quad v = v_1 + v_2 \quad w = w_1 + w_2 \quad \{v,w\} \not\subseteq [INT_{MIN}, INT_{MAX}]}{\Gamma \vdash optExp(e_1 + e_2) = (e'_1 + e'_2,\ [max(v, INT_{MIN}), min(w, INT_{MAX})])}\ \textsc{Add2}$$

$$\frac{\Gamma \vdash optExp(e_i) = (e'_i,\ [v_i, w_i]),\ i \in \{1,2\} \quad 0 \notin [v_2,\ w_2] \quad INT_{MIN} \notin [v_1,\ w_1] \vee -1 \notin [v_2,\ w_2]}{\Gamma \vdash optExp(e_1/e_2) = (EraseOverflowCheck(EraseDivCheck(e'_1/e'_2)),\ divInterval(v_1, w_1, v_2, w_2))}\ \textsc{Div1}$$

$$\frac{\Gamma \vdash optExp(e_i) = (e'_i,\ [v_i, w_i]),\ i \in \{1,2\} \quad 0 \notin [v_2,\ w_2] \quad INT_{MIN} \in [v_1,\ w_1] \wedge -1 \in [v_2,\ w_2]}{\Gamma \vdash optExp(e_1/e_2) = (EraseDivCheck(e'_1/e'_2),\ divInterval(v_1, w_1, v_2, w_2))}\ \textsc{Div2}$$

$$\frac{\Gamma \vdash optExp(e_i) = (e'_i,\ [v_i, w_i]),\ i \in \{1,2\} \quad 0 \in [v_2,\ w_2] \quad INT_{MIN} \notin [v_1,\ w_1] \vee -1 \notin [v_2,\ w_2]}{\Gamma \vdash optExp(e_1/e_2) = (EraseOverflowCheck(e'_1/e'_2),\ divInterval(v_1, w_1, v_2, w_2))}\ \textsc{Div3}$$

$$\frac{\Gamma \vdash optExp(e_i) = (e'_i,\ [v_i, w_i]),\ i \in \{1,2\} \quad 0 \in [v_2,\ w_2] \quad INT_{MIN} \in [v_1,\ w_1] \wedge -1 \in [v_2,\ w_2]}{\Gamma \vdash optExp(e_1/e_2) = (e'_1/e'_2,\ divInterval(v_1, w_1, v_2, w_2))}\ \textsc{Div4}$$

$$\frac{\Gamma(x) = \tau_{INT} \qquad [\![\tau_{INT}]\!] = [\tau_{MIN},\ \tau_{MAX}]}{\Gamma \vdash optExp(x) = (x,\ [max(\tau_{MIN}, INT_{MIN}), min(\tau_{MAX}, INT_{MAX})])}\ \textsc{VarInt}$$

$$divInterval\ (v_1, w_1, v_2, w_2) = \begin{cases} [w_1/v_2, min(v_1/w_2, INT_{MAX})], & \text{if } w_2 < 0 \wedge w_1 < 0 \quad (1) \\ [w_1/w_2, v_1/v_2], & \text{if } w_2 < 0 \wedge v_1 > 0 \quad (2) \\ [w_1/w_2, min(v_1/w_2, INT_{MAX})], & \text{if } w_2 < 0 \wedge p_1 \quad (3) \\ [v_1/v_2, w_1/w_2], & \text{if } v_2 > 0 \wedge w_1 < 0 \quad (4) \\ [v_1/w_2, w_1/v_2], & \text{if } v_2 > 0 \wedge v_1 > 0 \quad (5) \\ [v_1/v_2, w_1/v_2], & \text{if } v_2 > 0 \wedge p_1 \quad (6) \\ [v_1, min(|v_1|, INT_{MAX})], & \text{if } p_2 \wedge w_1 < 0 \quad (7) \\ [-w_1, w_1], & \text{if } p_2 \wedge v_1 > 0 \quad (8) \\ [-max(|v_1|, |w_1|), min(max(|v_1|, |w_1|), INT_{MAX})], & \text{if } p_1 \wedge p_2 \quad (9) \end{cases}$$

where $p_i = \neg(w_i < 0 \vee v_i > 0), \quad i \in \{1,2\}$

Fig. 2. RT-OPT Specification for Expression (excerpts)

Lastly, the VARINT rule specifies that an integer variable reference's interval is its integer type range intersected by the platform's integer range (i.e., leveraging the RT-OPT invariant that all computed integer values are always checked for overflows).

Well-Typed State: VARINT assumes that it can use the variable's integer type range for the variable reference's interval. This holds if all values in the state are well-typed. To discharge this assumption, we first specified the meaning for a state to be well-typed:

```
Inductive wellTypedState: symTabRT→ state→ Prop:=
| WellTypedState: ∀ strT s,
   (∀ x v, fetch x s = Some v→ ∃ t, lookup strT x = Some t∧ wellTypedValue t v)→
      wellTypedState strT s.
```

then proved that EVAL-RT specification (hence, by virtue of consistency transitivity, EVAL specification) preserve state well-typed-ness, for example, for EVAL-RT statement semantics that may incur state changes, we proved the following preservation lemma:

```
Lemma wellTypedStatePreservation: ∀ s s' stmt strT,
  wellTypedState strT s→ evalStmtRT strT s stmt s'→ wellTypedState strT s'.
```

Evaluation: To certify RT-OPT, we proceeded similarly to RT-GEN certification (albeit much more complex to prove); that is, we proved that RT-OPT specification is consistent with respect to the RT-GEN specification described in Section 2.2, and that RT-OPT

implementation is consistent with respect to its specification. Therefore, the implementation is transitively consistent with respect to the SPARK mechanized semantics.

2.4 Certifying GNAT RT Check Generator

Now that we have RT-GEN and RT-OPT, we can implement a conformance checker that can establish that, for a SPARK 2014 program p, the run-time check decoration insertion of the GNAT front-end for p conforms to the mechanized SPARK 2014 reference semantics. Specifically, for program p, the GNAT front-end generates a fully resolved AST with run-time check decorations, and we developed a tool called Jago that takes the GNAT AST and produces: (1) a Coq object of type AST (ast), where the GNAT run-time decorations are erased, and (2) a Coq object of type AST-RT (ast-rt-gnat), where the GNAT run-time decorations are preserved (Jago also applies some program transformations to desugar language constructs that lie outside of the language subset to fall within the language subset). Then, applying RT-GEN on ast produces ast-rt-gen, and applying RT-OPT on ast-rt-gen produces ast-rt-opt, both of which are of type AST-RT.

To automate the actual AST conformance check, we implemented a tool in Coq – $\subseteq$, that given two objects of type AST-RT, it determines whether the set of run-time checks in the first object is a subset of the second's. Thus, GNAT run-time decoration insertion on program p is conformant to the SPARK 2014 reference semantics if ast-rt-opt $\subseteq$ ast-rt-gnat $\subseteq$ ast-rt-gen. This toolchain essentially turns the GNAT front-end into a certifying run-time check decoration generator; that is, for a given program p, it generates evidence of "conformity to SPARK 2014 reference semantics for p's run-time check decorations" that is automatically machine-checked by certified tools.

Note that this does not guarantee that the actual binary run-time check assertion code for p subsequently generated by the GNAT compiler back-end is correct; it simply means that decorations indicating what assertions should be produced is correct. This, alone has significant value because, for example, it goes a long way toward establishing the correspondence between GNAT and GNATprove's (as well as any other SPARK backend tools') treatment of run-time checks. Moreover, since there are only three categories of run-time checks relevant for this language subset, since each of these categories can be represented by a simple code pattern involving a few numerical comparisons, since the pattern itself can be easily inspected and tested, and since the generation of binary code for the pattern is reasonable straightforward and can also be easily tested, one might argue that establishing the correctness of the decorations is one of the most important steps in establishing trust in the overall end-to-end production of the executable run-time checks.

3 Evaluation: Certifying GNAT

We evaluated GNAT according to the methodology described in Section 2.4 on a collection of programs. Table 1 presents the experiment data for various program units (packages/procedures) from the test programs. The first two SPARK programs come from the Ada Conformity Assessment Test Suite (ACATS) [16] that all Ada compilers must pass. SPARKSkein is an implementation of the Skein hash algorithm in SPARK, which was proved free of run-time errors [4]. Tetris is the motivating example described in Section 1, which is implemented partly in SPARK and partly in Ada (we only checked the SPARK part). All other examples are representative code from AdaCore, Altran

Unit	LoC	Base				GNAT				Opt				Diff			
		D	O	R	T	D	O	R	T	D	O	R	T	D	O	R	T
ACATS_c53007a	143	–	16	–	**16**	–	14	–	**14**	–	14	–	**14**	–	–	–	**–**
ACATS_c55c02b	74	–	2	5	**7**	–	2	–	**2**	–	2	–	**2**	–	–	–	**–**
array_record_package	54	1	11	2	**14**	1	11	2	**14**	1	11	2	**14**	–	–	–	**–**
array_subtype_index	12	–	1	1	**2**	–	1	–	**1**	–	1	1	**2**	–	–	+1	**+1**
arrayrecord	43	1	9	2	**12**	1	9	–	**10**	1	9	–	**10**	–	–	–	**–**
assign_subtype_var	10	–	1	1	**2**	–	1	–	**1**	–	1	1	**2**	–	–	+1	**+1**
binary_search	40	1	6	12	**19**	1	–	4	**5**	–	–	4	**4**	-1	–	–	**-1**
bounded_in_out	17	–	1	4	**5**	–	–	3	**3**	–	–	4	**4**	–	–	+1	**+1**
dependence_test_suite_01	164	–	2	–	**2**	–	2	–	**2**	–	2	–	**2**	–	–	–	**–**
dependence_test_suite_02	249	–	15	–	**15**	–	15	–	**15**	–	15	–	**15**	–	–	–	**–**
division_by_non_zero	12	1	2	1	**4**	1	–	–	**1**	–	–	–	**–**	-1	–	–	**-1**
faultintegrator	25	–	2	–	**2**	–	2	–	**2**	–	2	–	**2**	–	–	–	**–**
gcd	18	1	3	–	**4**	1	3	–	**4**	1	3	–	**4**	–	–	–	**–**
linear_div	21	–	3	–	**3**	–	3	–	**3**	–	3	–	**3**	–	–	–	**–**
modulus	24	1	2	3	**6**	1	1	–	**2**	–	1	–	**1**	-1	–	–	**-1**
odd	14	1	2	–	**3**	1	1	–	**2**	–	1	–	**1**	-1	–	–	**-1**
p_simple_call	36	–	5	–	**5**	–	5	–	**5**	–	5	–	**5**	–	–	–	**–**
prime	21	1	2	–	**3**	1	2	–	**3**	1	2	–	**3**	–	–	–	**–**
quantifiertest	14	–	1	2	**3**	–	1	–	**1**	–	1	–	**1**	–	–	–	**–**
SPARKSkein	646	7	94	246	**347**	7	58	29	**94**	–	52	25	**77**	-7	-6	-4	**-17**
sort	43	–	5	6	**11**	–	5	6	**11**	–	5	6	**11**	–	–	–	**–**
Tetris	373	–	29	58	**87**	–	–	25	**25**	–	–	25	**25**	–	–	–	**–**
the_stack	42	–	4	6	**10**	–	–	6	**6**	–	–	6	**6**	–	–	–	**–**
the_stack_praxis	35	–	2	4	**6**	–	–	4	**4**	–	–	4	**4**	–	–	–	**–**
two_way_sort	49	–	4	17	**21**	–	–	4	**4**	–	–	4	**4**	–	–	–	**–**

Table 1. Experiment Data (excerpts)

and our own designed benchmark covering the core language subset. For each unit, **LoC** gives the line number of code. **Base**, **GNAT** and **Opt** give the number of run-time checks in ast-rt-gen, ast-rt-gnat and ast-rt-opt respectively, and **Diff** represents the number of run-time checks in **GNAT** that differs from the ones in **Opt**. Dash ("–") means "none"; a negative number -n in **Diff** means that **Opt** removes n more run-time checks than **GNAT**; and, a positive number +m means **Opt** has m more number of run-time checks than **GNAT** "somehow". Sub-column **D** gives the number of division by zero run-time checks; **O** and **R** give the number of overflow run-time checks and range run-time checks; and, **T** is the total number of run-time checks (i.e., **D+O+R**). RT-GEN and RT-OPT run fast (within seconds) and the data are omitted here due to space constraints.

As can be observed from Table 1, the GNAT frontend is a solid tool for run-time check generation/verification as most of its generated run-time checks match the certified RT-OPT. This is reasonable because our formalization captures the most commonly used run-time checks in SPARK and GNAT is quite mature after many years of effort to improve it, as well as the effort to improve the GNATprove toolchain by AdaCore and Altran (which drives some of the improvements in GNAT). However, RT-OPT edges out GNAT in some cases, especially for SPARKSkein. One reason is that GNAT does not take any advanced optimizations, for the division/modulus binary operator, it does not optimize even with constant; for example, GNAT generates division by zero check for the expression `(R + 1) mod 3` while RT-OPT optimized it away. For SPARKSkein, consider a procedure call `Inject_Key(R * 2)`, (for procedure declaration `Inject_Key(X: in U32)`), `R` is a variable of type `U32`, and `U32` is a subtype of `Integer` with range `0..INT_MAX`; an overflow check for `R * 2` is enough to guarantee the absence of both overflow and range error, while GNAT keeps both overflow check and range check for such cases. There are other cases showing that RT-OPT is better than GNAT's optimizations.

In our initial evaluation (shown in the experiment table), GNAT produces fewer run-time checks than RT-OPT; these inconsistencies turned out to be benign because they are due to differences in how GNAT (vs RT-OPT) reports the need for checks

and in how it assumes down-stream translation will interpret decorations for run-time checks. Once observed, the inconsistencies are rectified by slightly modifying $\subseteq$ to match GNAT's conventions, thus, resulting in a fully automatic approach to justify correctness of GNAT run-time check decorations. For completeness sake, we document the inconsistencies that we found (and fixed) here. In procedure array_subtype_index, there is an assignment `A(0):=0`, where the index type of `A` is a subtype of integer with range `1..10`; thus, accessing `A` with the index `0` is out of its required range, so it will cause a range error. GNAT gives a compile time *warning* as specified in Ada reference manual without generating a range check; on the other hand, RT-OPT keeps this check (a similar issue exists for assign_subtype_var). Another difference is due to a single run-time check decoration in the GNAT AST that can lead downstream translation steps to introduce run-time checking code that implements multiple checks, e.g., a single GNAT run-time check AST decoration for an argument (both **`in`** and **`out`**) is interpreted as giving rise to code for two checks for both passing in argument and passing out return value.

Lessons Learned: The fact that RT-OPT is better in some cases illustrates that, despite its maturity, GNAT can still be improved further, e.g., by adopting the optimization specified and implemented in RT-OPT; that is, the RT-OPT specification can be used as a reference for implementing further optimizations in GNAT, and once implemented, they can then be checked for conformance against the RT-OPT implementation. Furthermore, in the case where new optimizations are added to GNAT that goes beyond RT-OPT as presented here, those new optimizations can be added to RT-OPT in order to: (a) mechanically verify that they are correct, and (b) further keep GNAT as a certifying run-time check generator.

Our research work demonstrates the feasibility of engineering an approach and corresponding tools with mechanized correctness proofs that leverage recent advancements and maturity of various formal method techniques and tools to make a direct impact in significantly increasing confidence in industrial tools; in our case, the industrial tools are used to develop critical systems that require the utmost level of integrity, thus, warranting such effort. From a business perspective, we believe it is desirable as it adds to the value proposition – the trustworthiness of GNAT compiler and associated SPARK 2014 is increased. Furthermore, we believe that the approach can eventually help in tool qualification processes typically done in certifications and regulatory reviews associated with standards (e.g., DO-178C in avionics) that increasingly recognize the value of formal methods and an official tool qualification process.

Threats To Internal Validity: Our approach is predicated on the assumption that practitioners are willing to trust the approach's trust-base, which includes Coq and the SPARK formal language semantics presented in Section 2.1. In addition, our current implementation uses: (a) the parser, symbol resolver, and type checker of GNAT itself, and (b) Jago to build program representations in Coq; both are not certified tools. Ideally, a certified frontend can be developed to address this issue; this certified frontend is orthogonal and out of the scope of the work presented here, and they can be addressed in the future. Moreover, the $\subseteq$ tool that compares AST-RT objects is manually inspected instead of certified (it is small – 172 LoC, and its functionality is very simple); regardless, it should be considered as part of the trust-base at this point of time.

Threats To External Validity: One must also consider the extent to which the results presented for the given test suite would extend to SPARK 2014 programs in general. For this objective, program size and execution time are not really issues – the cost of insertion of run-time checks is in general linear in the number of AST nodes. The in-

terval analysis needed for optimization does add some additional complexity, but not enough to significantly impact performance. On the other hand, a principle concern is that our test suite provides appropriate coverage of all the different types of run-time checks specified in the SPARK 2014 language reference manual. In addition, our language subset needs to be expanded to eventually cover the full programming language (in fact, this work represents our third iteration based on the initial work [6]).

4 Related Work

The idea of a certifying approach that generates evidence that can be machine-checked goes back to proof-carrying code (PCC) [14] for memory safety. Since then, recent advancements and maturity in interactive theorem proving has enabled one to implement certified systems directly inside a theorem prover with what widely acknowledged as a relatively small trust-base footprint. One prominent work is the CompCert project [11], which demonstrates that one can now feasibly develop a certified optimizing compiler in Coq that guarantees the machine code it produces is behaviorally equivalent to its C source code. To provide such guarantee, it starts with formalizing a large subset of the C programming language – Clight [18], which is then used for proving program behavior preservation throughout its compilation pipeline. The main difference to our semantics work is that SPARK requires run-time checks as part of its semantics, which complicated our formalization effort (as described in Section 2.1).

In contrast to CompCert where the certified compiler is developed in Coq, the GNAT compiler is not developed in Coq. In fact, there are often a number of goals driving the development of language tools – performance, scalability, reusability, maintainability, etc. Many, if not most, of these often conflict with the goal of verifiability (and thus "mechanized" verifiability). In addition, it is hard to imagine verifying tools with a lot of legacy software (such as GNAT). Thus, there are strong forces against developing a certified language tool, and in situations like these creating a certifying tool can be an appropriate strategy when high-assurance is needed.

Some advantages of the certifying approach are that: (1) it can be adopted by existing/untrusted (high-performing) tools, and (2) it is much easier and a lot less costly to develop. One main advantage of the certified approach is that its correctness evidence (proof) is for all uses (runs) of the tool, while the certifying one is specific to a particular use; this specificity is sufficient and in line with typical tool qualification processes where tools are qualified for the particular use on the software being regulatorily certified for standard compliance [9].

The closest work on run-time checks to ours is Verasco [10] – a certified run-time check analyzer for C, whose design was inspired by Astrée [8]. RT-OPT employs a simpler abstract numerical domain compared to Verasco; thus, it can potentially discharge more verification conditions associated with run-time checks. This presents opportunities for RT-OPT future improvements. On the other hand, the soundness of Verasco was proven, but not its completeness. Because SPARK (unlike C) includes implicit run-time checks that must be accounted for in the semantics, completeness guarantees that all run-time checks are as prescribed by (traceable to) the reference semantics.

5 Conclusions and Future Work

In this paper, we have illustrated how the formal semantics of SPARK can be used in a mechanized proof infrastructure to check that ASTs produced by the GNAT compiler

frontend having correctly incorporated decorations for run-time checks. This included developing an optimizer with mechanized proofs of correctness that achieves run-time check placement optimizations equal to or better than GNAT. The effectiveness of the approach was demonstrated using programs from AdaCore test suites.

Our next step is to build a mechanically proved translation from SPARK into CompCert's Clight, which would then provide a verified compiler for SPARK 2014 to the target languages supported by CompCert. In addition, the Jago translator also enables one to develop in Coq an integrated verification environment that includes the ability to use Coq to mechanically verify that a SPARK program conforms to its formally specified contracts. In situations where very high confidence is needed, this type of infrastructure could be used directly by verification engineers, or it could enable existing automated tools like Kiasan [3] or GNATprove [13] to emit Coq proofs establishing that their verification results for a particular program are correct.

References

1. Barnes, J.: SPARK: The Proven Approach to High Integrity Software. Altran Praxis (2012)
2. Barnes, J.: Ada 2012 Rationale - The Language, The Standard Libraries, Lecture Notes in Computer Science, vol. 8338. Springer (2013)
3. Belt, J., Hatcliff, J., Robby, Chalin, P., Hardin, D., Deng, X.: Bakar kiasan: Flexible contract checking for critical systems using symbolic execution. In: NFM. pp. 58–72 (2011)
4. Chapman, R., Botcazou, E., Wallenburg, A.: SPARKSkein: A formal and fast reference implementation of Skein. In: SBMF, pp. 16–27. LNCS, Springer (2011)
5. Chlipala, A.: Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant. MIT Press (2013)
6. Courtieu, P., Aponte, M., Crolard, T., Zhang, Z., Robby, Belt, J., Hatcliff, J., Guitton, J., Jennings, T.: Towards the formalization of SPARK 2014 semantics with explicit run-time checks using coq. In: HILT. pp. 21–22 (2013)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252 (1977)
8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: ESOP. pp. 21–30 (2005)
9. Hatcliff, J., Wassyng, A., Kelly, T., Comar, C., Jones, P.L.: Certifiably safe software-dependent systems: challenges and directions. In: FOSE. pp. 182–200 (2014)
10. Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: POPL. pp. 247–259 (2015)
11. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
12. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015)
13. Moy, Y., Ledinot, E., Delseny, H., Wiels, V., Monate, B.: Testing or formal verification: DO-178C alternatives and industrial experience. IEEE Software pp. 50–56 (2013)
14. Necula, G.C.: Proof-carrying code. In: POPL. pp. 106–119 (1997)
15. O'Neill, I.: SPARK – a language and tool-set for high-integrity software development. In: Industrial Use of Formal Methods: Formal Verification. Wiley (2012)
16. Ada conformity assessment test suite (ACATS). http://www.ada-auth.org/acats.html
17. Ada reference manual. http://www.ada-auth.org/standards/ada12.html
18. Clight. http://compcert.inria.fr/doc/html/Clight.html
19. Compcert-c. http://compcert.inria.fr/compcert-C.html
20. The Coq proof assistant. http://coq.inria.fr
21. SPARK 2014 reference manual. http://docs.adacore.com/spark2014-docs/html/lrm/
22. Why3 - where programs meet provers. http://why3.lri.fr/