AdaCore | Build Software that Matters

Tech Paper

Managing the Proof Context in SPARK

CLAIRE DROSS, AdaCore, France
JOFFREY HUGUET, AdaCore, France
JOHANNES KANIG, AdaCore, France

adacore.com

Managing the Proof Context in SPARK

Claire Dross
dross@adacore.com
AdaCore
France

Joffrey Huguet
huguet@adacore.com
AdaCore
France

Johannes Kanig
kanig@adacore.com
AdaCore
France

Abstract

SPARK performs verification of Ada programs in an auto-active style: the verification is done by automated solvers, but users need to write annotations in the source code - in general contracts - for the proof to succeed. For auto-active verification of programs to scale, managing the size of the proof context given to automated solvers is key. In this talk, we will describe the various mechanisms and heuristics used in SPARK to reduce the size of the proof context. They range from completely automated to manual, taking full advantage of the auto-active verification style.

ACM Reference Format:
Claire Dross, Joffrey Huguet, and Johannes Kanig. 2026. Managing the Proof Context in SPARK. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The SPARK tool for formal verification for Ada [2] translates Ada into the intermediate language Why3ML, then uses the Why3 tool [4] to generate verification conditions for SMT solvers [3] to prove properties of programs. Input programs can be large, and formalizing types, objects and assumptions tends to produce even larger logical formulas. If care is not taken, the context (part of the verification condition that is not the goal) can become very large indeed.

In the presence of large contexts, SMT solvers tend to take more resources (time or resource steps) to prove a goal, and exhibit instability, where small, logically irrelevant input changes can drastically change the resources needed to prove the goal. Instability has been identified as one of the big challenges for users of formal verification tools, as it makes maintaining proofs harder across code changes or tool changes [13].

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In addition, Why3 internal data structures mimic the proof context that is then ultimately given to the SMT solver. If the context is reduced during or even before Why3 takes on its work, tool performance can be greatly improved.

In this paper we show how we strive to minimize the proof context along the entire tool chain of SPARK front-end and Why3 backend, with the goal of improving tool performance, proof performance, and proof stability. The three axes of improvement presented in this paper are:

- Automated pruning of unused symbols and hypotheses in (the Why3 equivalent of) SMT files, taking into account user-provided information;
- Automated pruning/hiding of composite types during the generation of Why3ML;
- Annotations for user-guided context management.

To assess efficiency of the different methods, we have done benchmarks on a subset of our testsuite. This includes:

- a pathological test coming from an industrial use case, with complex data structures whose internal format does not matter for verification, where SPARK was abnormally slow;
- a proved implementation of System.Arith_Double, a GNAT run-time library unit on arithmetics over double word signed integer values, at the limit of what automated solvers can achieve. Its verification requires manual guidance in the form of user-written assertions;
- a proved implementation of red-black trees using arrays [9]. It is structured in a layered way to enforce separation of concern for verification, using separate packages. This is the standard way of handling abstraction when programming in Ada.;
- other tests that we think are representative of Ada programs that we want to prove.

Unless specified, the prover used to count the number of VCs and assess proof time is CVC5 [1] version 1.2.0. Another prover used through the article is Z3 [8] version 4.13.4.

2 Automated pruning of the proof files

SPARK generates one Why3 file per subprogram (function or procedure). The structure of this file is a list of Why3 modules, with the final module being the translation of the Ada subprogram into a WhyML program. The other modules only contain declarations and axioms; these correspond to Ada subprograms, types, and objects which are defined inside or outside of the subprogram under analysis. They never

contain WhyML program code, though some declarations may use references, pre- and postconditions or other WhyML features. Why3 then builds the verification conditions by resolving module references and essentially concatenating all used modules.

The division of declarations into modules ensures that only necessary modules end up in the context of the VCs. For example, if a subprogram doesn't use floating-point operations (nor contains data types that, directly or indirectly, contain floating-point types), then the module containing floating-point operations is not included.

While the module mechanism is powerful, it is also quite coarse-grained in multiple ways. First, modules usually contain multiple symbols and their associated axioms. For example, most floating-point operations are in a single big module. Using a module for each floating-point operation would make these operations harder to use.

Second, the WhyML module for the subprogram under analysis includes all Why3 modules that are needed anywhere in the subprogram. But individual VCs might not need all of them. For example, if another subprogram is called in the "if" branch of a conditional statement, then VCs that arise from checks in the "else" branch might not need information about this call.

Essentially, we want to remove unused symbols and hypotheses from the VC, which is a well-known problem that has been tackled quite often in VCG. One recurring problem is the following. Take as an example the following Why3 declarations (taken from a Why3 library used with SPARK):

```
val function cos (x : t) : t
axiom cos_range:
  forall x : t. abs_le_one (cos x)
axiom cos_zero:
  forall x : t. is_zero x -> is_one (cos
    x)
axiom cos_finite :
  forall x : t [is_finite (cos x)].
    is_finite x -> is_finite (cos x)
```

It is clear that the axioms should only be included if the symbol "cos" is used anywhere else. But a naive hypotheses elimination algorithm would have no reason to remove the axioms (on what basis?), and would count the axioms as usages of "cos", so in the end all declarations would be retained.

The Why3 solution to this problem is to annotate axioms as belonging to a symbol as follows:

```
val function cos (x : t) : t
axiom cos_range:
  forall x : t. abs_le_one (cos x)
meta "remove_unused:dependency" axiom
    cos_range, function cos
axiom cos_zero:
  forall x : t. is_zero x -> is_one (cos
    x)
meta "remove_unused:dependency" axiom
    cos_zero, function cos
axiom cos_finite :
  forall x : t [is_finite (cos x)].
    is_finite x -> is_finite (cos x)
meta "remove_unused:dependency" axiom
    cos_finite, function cos
```

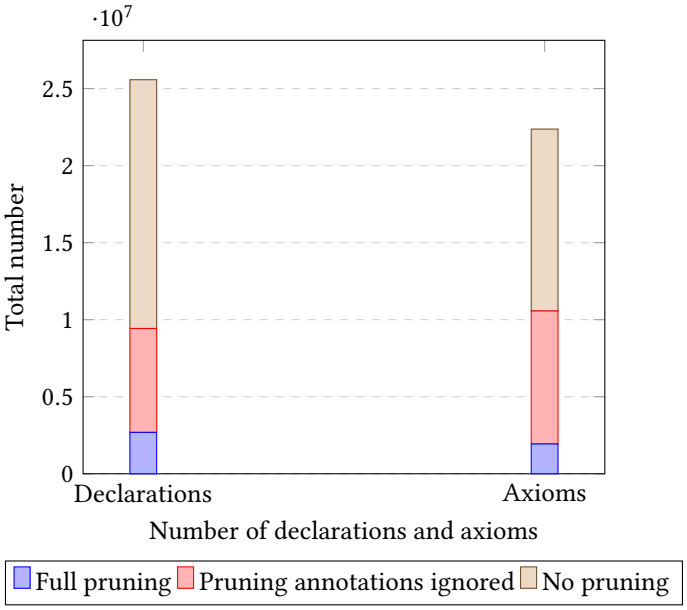

Figure 1. Evolution of number of declarations and axioms depending on the pruning method on the proof files

For the hypotheses elimination algorithm, these axioms now don't count as usages of "cos". Whenever "cos" is retained for other reasons (is used elsewhere), the axioms are automatically included as well.

Almost all axioms in the SPARK prelude and Why3 standard library are annotated in this way, attaching them to the symbol they help define. The same is true for axioms that are generated by SPARK for any Ada types, functions, contracts, and so on.

Results on our benchmarks can be observed in figure 1. The coarse-grained module mechanism (i.e., when pruning naively, and ignoring the pruning annotations) retains 42% of declarations and 73% of axioms. When pruning annotations are taken into account to remove unused dependencies, this remaining percentage decreases for both, settling at 17% of the original numbers of declarations and axioms. In terms of proof time and performance, this is translated in figure 2, by an increase in proved VCs at low time; more difficult, longer to prove VCs seem to be less affected. This could be explained by the fact that they use more information in the context to be proved thus are less affected by context pruning.

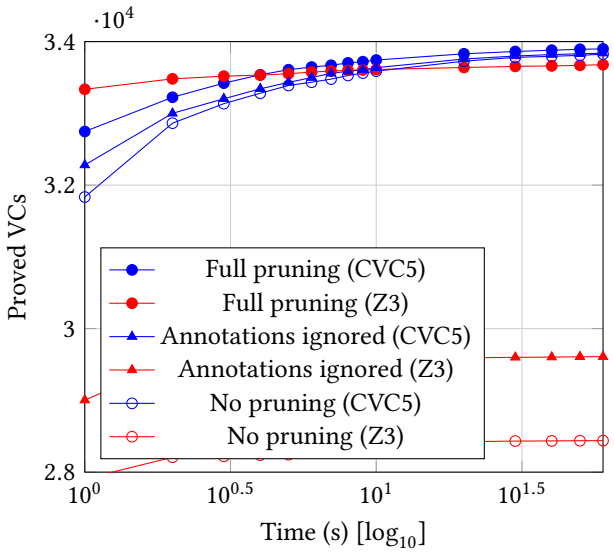

Figure 2. A graph showing number of VCs proved over time depending on the pruning method on the proof files

3 Automated pruning of the Ada code

In addition to the proof-file-level techniques presented in the previous section, SPARK also has mechanisms to prune information before reaching the level of Why3, based on the structure of the Ada code. The first feature we present is similar to those from the previous section, as it is also concerned with removing unused symbols and axioms. It is based on the observation that types, and in particular composite types, lead to the generation of multiple symbols that cannot easily be removed at the why3 level even if the form of the type does not matter in the program. We focus in particular on records that are ubiquitous in Ada. The idea is to replace Ada records whose fields are never mentioned by abstract types in Why3, removing all together the components themselves, record specific symbol declarations, and indirectly all symbols that are used by these components.

Doing this optimization at the Ada level has drawbacks, as pruning will be done for the whole unit, instead of once per verification condition if it was done in Why3. As a result, record fields will not be abstracted away in a given verification condition if they are used elsewhere in the unit. However, even though determining whether the fields of a record are unused at the Ada level is not completely straightforward - record components might be used implicitly through default initialization, Ada equality... -, we believe that it is still notably simpler. Indeed, it makes it easier to get all the relevant symbols even if they are scattered across several modules, and allows for the reuse of the regular handling of types hidden from analysis in SPARK. In addition, hiding the types early in the analysis means less work for the translation and verification condition generation as well.

Let us take the pathological customer test as an example. This test features complex data stuctures, and most part of these structures is unused for verification. This leads SPARK to generate lots of declarations for parts of types that are never referenced in the program. The mechanism detailed above has decreased the number of declarations and axioms from 20799 and 14189 to 12375 and 8203 respectively. We have measured time spent in translation to Why3ML and calls to Why3 and observed a reduction of 33 seconds over the 67 seconds it used to take to perform these steps, i.e. an almost 50% decrease in an optimal test. Note that other tests of the benchmark are less affected by this optimisation; the featured test showcases few VCs and has several big types whose fields are mostly unused throughout the program.

The other mechanism we discuss takes advantage of the usual handling of abstraction in Ada. It is based on packages that are namespaces used to group together the declarations of various entities (objects, functions, types...). They can have up to three parts:

- a (public) specification, grouping the declarations of entities that users of the package can access,
- a body, where the subprograms declared in the specification are defined, but it is also possible to declare data or additional subprograms in the body of a package, if they are not needed in the specification,
- a private (specification) part, used in particular to declare a type as private and then complete it in a way that does not allow other units to access its full view.

Entities declared in the body or the private part of a package cannot be accessed from outside of this package. In Ada, packages are used at top level for libraries, but also nested inside subprograms or other packages for abstraction. As an example, the type R below is private, so it is not possible to access its A field from outside of the package P. In the same way, the function Property_1 can be used from outside whereas Property_2 and Property_3 can only be accessed from inside of P's private part and body (for Property_2) or private part (for Property_1):

```
package P is
  type R is private;
  function Property_1 (X : R) return
      Boolean;
private
  type R is record
     F1, F2 : Integer;
  end record;
  function Property_2 (X : R) return
      Boolean;
end P;

package body P is
  function Property_3 (X : R) return
      Boolean;
```

```
   [...]
end P;
```

In SPARK, we have decided to take advantage of this Ada feature to enforce actual abstraction by proof by hiding from the analysis of user code everything occurring inside a (nested) package body. We have decided however to keep the private part visible, to avoid forcing users to introduce models for private types. It makes it possible for a user to decide whether or not some information should be abstracted away for proof by putting it in the body or in the private part of the enclosing package. As an example, the functions Property_1 and Property_2 below can be called outside of P. When analyzing such calls, the body of Property_1 will be available but not the body of Property_2. Note that if a call to Property_2 occurs inside of P, be it in its specification or body, then the analysis will be able to use its definition:

```
package P is
   type R is private;
   function Property_1 (X : R) return
       Boolean;
   function Property_2 (X : R) return
       Boolean;
private
   type R is record
      F1, F2 : Integer;
   end record;
   function Property_1 (X : R) return
       Boolean is (X.F1 = 0);
end P;

package body P is
   function Property_2 (X : R) return
       Boolean is (X.F2 = 0);
end P;
```

Note that, thanks to the automated hiding of unused records, having the private part of libraries visible by default does not cause too much overhead in general if abstraction is enforced by the user at this level - mainly, if there are no functions declared in the public specification and defined in the private part of the package that access the fields of the record.

An annotation allows users to override the default visibility rules on a case by case basis. It is possible to hide the private part of a package - when verifying other units only. It reuses the same mechanism as the one used to hide unused record types. It is also possible to annotate the body of a nested package so it is visible when verifying the enclosing unit - but not the body of another unit, as SPARK does not see the bodies of other units during analysis.

Figure 3 shows the effect of hiding package bodies from other units on the red black trees examples. We can see that CVC5 and Z3 react differently to the context size. Hiding package bodies seems to be a slight improvement at lower

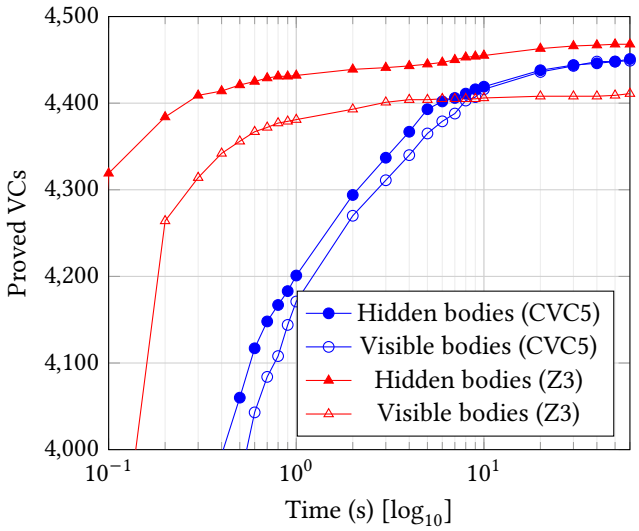

Figure 3. Number of proved VCs per time unit in the red black trees test. Total number of VCs to prove is 4494.

times for CVC5, and it barely makes a difference for more difficult VCs. However, Z3 seems to be more sensitive to context size, as it is able to prove 57 more VCs when package bodies are hidden. The results also show a positive effect on proof time for easier VCs. The difference in behavior between CVC5 and Z3 is not abnormal, because they have different handlings of quantification for example.

4 Manual handling of the proof context

Using the mechanisms presented in the previous section, it is possible for users to influence the proof context of the generated proof obligations, either by choosing where they give the definition of a function, by introducing nested packages or by manually hiding or disclosing private parts or nested package bodies. In this section, we present specific functionalities designed to give users control over the proof context.

The first two constructs we present are used to control the local proof context. They allow some level of control on how information coming from instructions inside the analyzed subprogram are added to the context. The first one, called Assert_And_Cut, provides a property that should be proved by the tool, and then used to abstract away the sequence of statements leading to the assertion from the beginning of the enclosing block: the proof context will be pruned of all information about what happened during the block, except for the supplied property. As an example, after the following block of code, the SPARK tool will know that X has a non-negative value, but not that it is 0:

```
begin
   X := 0;
```

```
   pragma Assert_And_Cut (X >= 0);
end;
```

The cut operation By, inspired by a similar feature in Why3 [4], is used to insert intermediate steps that can be used to prove a particular assertion but won't be kept in the proof context afterwards. In SPARK, By is not a keyword, but a function with two boolean parameters that is specifically recognized by the tool. When By is called, the second parameter is used to prove the first, but then it does not occur in the proof context anymore. It is interesting in particular in combination with quantified expressions. It allows in particular for providing a witness to prove an existential quantifier, or to guide the proof on a universally quantified lemma element per element without introducing a loop. As an example, proving the following property is out-of-reach of the SMT solvers used at the backend of SPARK, as it defeats basic E-matching - there is no reason for the solver to know to try F (X) as a candidate for the existentially quantified formula:

```
function P (X, Y : Integer) return
    Boolean;
function F (X : Integer) return Integer
    with
  Post => P (X, F'Result);
pragma Assert
  (for all X in Integer => (for some Y in
      Integer => P (X, Y)));
```

The following version however is proved easily, as it simply requires proving P (X, F (X)):

```
pragma Assert
  (for all X in Integer => (for some Y in
      Integer => By (P (X, Y), Y = F
      (X))));
```

For the verification of the rest of the program however, the call to F will not be added to the proof context and the assertion will effectively be equivalent to the previous one.

In figure 4, we observe similar results as the red black trees example. CVC5 shows a slight improvement in proof performance for easier VCs, while Z3 shows a more distinct improvement. This indicates that integrating manual handling of context as part of the proof process is beneficial for proof performance. However, the manual nature of this technique might be a downside for users, as this can require trial-and-error to reach full proof.

Finally, it is possible to prune the global proof context when verifying a subprogram by removing the definition of specific function symbols. This can be done on a per subprogram basis, it is not possible to remove information for the proof of a part of a subprogram only. This is especially useful when dealing with recursive specification functions, as it allows hiding the recursive definition, that can cause matching loops when verifying the verification conditions, and only unfolding it as needed using a lemma. When using

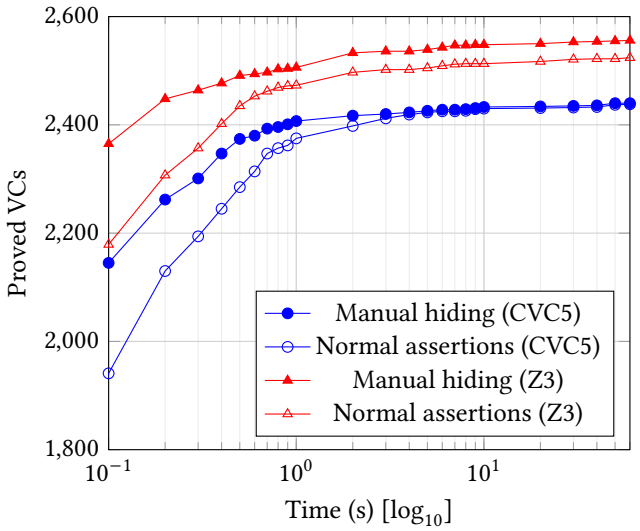

Figure 4. Effect of manual handling of context on System.Arith_Double proof

this feature, users can decide to either make the function definition visible by default and then hide it when necessary or hide it by default and disclose it on demand. As an example, the definition of Property_1 is visible by default in the example below, while the definition of Property_2 is hidden. The annotations in Main cause the definition of Property_1 to be hidden and the definition of Property_2 to be disclosed for its verification, leading to the second assertion being proved while the first is unprovable:

```
function Property_1 (X : R) return
    Boolean is (X.F1 = 0);
function Property_2 (X : R) return
    Boolean is (X.F1 = 0) with
  Annotate => (GNATprove, Hide_Info,
      "Expression_Function_Body");

procedure Main is
  pragma Annotate (GNATprove, Hide_Info,
      "Expression_Function_Body",
      Property_1);
  pragma Annotate (GNATprove,
      Unhide_Info,
      "Expression_Function_Body",
      Property_2);
  X : R := (0, 0);
begin
  pragma Assert (Property_1 (X));
  pragma Assert (Property_2 (X));
end;
```

5 Related Work

Reducing irrelevant proof context for SMT provers, with the goal of obtaining more and faster proofs, or reducing proof instability, is a long-standing challenge. It was recently acknowledged again by the Everest project [13].

Removing irrelevant hypotheses at the VC level is one obvious way to reduce proof context, and many different approaches have been studied in the past [7, 10, 11], and more recently [16]. The main novelty of the approach in Why3 is the guidance of the elimination process by user annotations.

Slicing [15] is a common technique to simplify a program before processing it further, including for program verification [12]. Our approach of abstracting composite types whose structure is not used by the program is similar to existing slicing techniques [14], but applies them to formal verification.

Reducing context through human annotations has received little attention to our knowledge. The By operator was first introduced in Why3 [6]. Both the original SPARK tool [5] as well as Why3 have a construct somewhat similar to Assert_And_Cut which abstracts all details from a program part, except for a user-provided assertion. The original SPARK tool also had a variant of the Hide_Info annotation, but no equivalent of Unhide_Info.

6 Conclusion

For deductive verification to scale to real world programs and become applicable to the industry, managing the growth of the proof context is key. The SPARK tool applies a number of heuristics to prune this context, removing axioms along with the function they define and abstracting away unused parts of data-structures. But what can be done automatically has limits, as finding what is useful or not might require advanced knowledge of what the program is doing. So, in addition, the SPARK tool provides features to allow users to help manage the context, consistently with the auto-active way deductive verification tools commonly favor. This includes high level mechanisms for handling abstractions and low level fiddling with local hypotheses. These techniques have shown an improvement on proof time and proof performance on our benchmarks.

As scaling up remains an important issue for adoption, we plan to continue improving in this area. We will investigate additional heuristics for automated hypothesis pruning, for example based on a notion of distance, as is done in SHAKE. We would also like to further develop our manual hiding techniques, to allow removing more information from the global context. It could include postconditions of functions, and values of constants for example.

References

[1] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

[2] John Barnes. 2012. *SPARK: The Proven Approach to High Integrity Software.* Altran Praxis.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages.* Wrocław, Poland, 53–64.

[5] Bernard Carré and TJ Jennings. 1988. *SPARK: The SPADE Ada Kernel: Version 1.0.* HM Stationery Office.

[6] Martin Clochard. 2017. Preuves taillées en biseau. In *vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA).* Gourette, France. https://inria.hal.science/hal-01404935

[7] Jean-François Couchot, Alain Giorgetti, and Nicolas Stouls. 2009. Graph Based Reduction of Program Verification Conditions. In *Automated Formal Methods (AFM'09), colocated with CAV'09*, Hassen Saïdi and N. Shankar (Eds.). Hassen Saïdi and N. Shankar, ACM Press, Grenoble, France, 40–47. https://inria.hal.science/inria-00402204

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08).* Springer-Verlag, Berlin, Heidelberg, 337–340.

[9] Claire Dross and Yannick Moy. 2017. Auto-Active Proof of Red-Black Trees in SPARK. 68–83. doi:10.1007/978-3-319-57288-8_5

[10] Kryštof Hoder and Andrei Voronkov. 2011. Sine qua non for large theory reasoning. In *International Conference on Automated Deduction.* Springer, 299–314.

[11] Jia Meng and Lawrence C Paulson. 2009. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic* 7, 1 (2009), 41–57.

[12] Benjamin Monate and Julien Signoles. 2008. Slicing for Security of Code. In *TRUST (Lecture Notes in Computer Science, Vol. 4968)*, Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch (Eds.). Springer-Verlags, 133–142. publis/2008_trust.pdf

[13] Project Everest Team. 2025. Project Everest: Perspectives from Developing Industrial-grade High-Assurance Software. Project Everest Website. https://project-everest.github.io/papers/

[14] Frank Tip, Jong-Deok Choi, John Field, and G Ramalingam. 1996. Slicing class hierarchies in C++. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* 179–197.

[15] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30, 2 (2005), 1–36.

[16] Yi Zhou, Jay Bosamiya, Jessica Li, Marijn JH Heule, and Bryan Parno. 2024. Context pruning for more robust smt-based program verification. In *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2024.* 59–59.

AdaCore | Build Software that Matters

adacore.com