Verification of Ada Programs with AdaHorn

Christian Herrera
fortiss GmbH
Munich, Germany
herrera@fortiss.org

Tewodros A. Beyene
fortiss GmbH
Munich, Germany
beyene@fortiss.org

Vivek Nigam
fortiss GmbH
Munich, Germany
nigam@fortiss.org

We propose *AdaHorn*, a model checker for verification of *Ada* programs wrt. correctness properties. AdaHorn translates Ada programs together with their related correctness properties into a set of *Constrained Horn Clauses* which are solved by well-known SMT solvers such as *Eldarica* and *PDR-Z3*. We propose a set of Ada programs inspired by *C* programs from the competition SV-COMP, and use them to compare the effectiveness of AdaHorn and *GNATProve*, a well-known static analyzer for Ada programs. Our experiments show that AdaHorn outperforms GNATProve by yielding correct results in more cases than GNATProve.

1 Introduction

Ada is a programming language widely used by the avionics, space, military, railways, among other communities of systems developers because its features, e.g. extremely strong typing, explicit concurrency, built-in language support for design-by-contract and non-determinism [1], allow developers to build robust and dependable safety critical systems.

Two prominent tools that support the development of such systems are GNATProve[1] [2] and Polyspace[2]. Those tools perform *static analysis* on Ada programs for detecting runtime errors, e.g. *arrays out-of-bounds*, *arithmetic overflows*, *division by zero*, etc. Static analysis is a technique that analyses programs without executing them. It is well known that tools based on this technique often yield *false positives*, i.e. results wrongly indicating that errors in programs occur, and *false negatives*, i.e. results wrongly indicating that errors in programs do not occur.

Model checking is a technique that can provide conclusive results wrt. the conditions that hold or do not hold in programs. In this work we aim to complement the support available for the development of those systems by proposing AdaHorn, a model checker for verifying Ada programs wrt. correctness properties. AdaHorn translates Ada programs together with their related correctness properties into a set of Constrained Horn Clauses (CHC) [3, 4, 5] which are solved by well-known SMT solvers such as Eldarica [6] and PDR-Z3 [7].

Similar to other tools, e.g. SeaHorn [8] and JayHorn [9], the design of AdaHorn consists of three main modules: (1) *Front-End Module* which enables translating Ada programs into an intermediate representation, namely, in *XML* format, which does not alter their original behaviour, and that allows program constructs, e.g. data types, procedures, functions, loops, etc., to be further translated into a suitable intermediate logic language, namely, CHC, in order to express system's behaviour, (2) *CHC Generator* which performs the translation of Ada programs together with their related correctness properties into CHC and, (3) *Back-End Module* which uses the mentioned SMT solvers to solve generated CHC. All three modules are easy to extend, and specially the Front-end Module and the Back-end Module allow an easy replacement of their underlying tools.

[1] GNATProve is a product of AdaCore available in academic and commercial versions.
[2] Polyspace is a commercial product of MathWorks.

Submitted to:
WPTE 2017

Although AdaHorn currently supports a small but useful subset of program constructs, e.g. integer, floating-point and boolean data types, loops, assertions, conditionals, arrays, procedures and functions, we are able to propose a set of Ada programs inspired by C programs from the competition SV-COMP 2017, and use them to compare the effectiveness of AdaHorn and GNATProve, a well-known static analyzer for Ada programs. Our experiments show that AdaHorn outperforms GNATProve by yielding correct results in more cases than GNATProve.

The contribution of this paper is twofold, namely, (1) we propose AdaHorn which to the best of our knowledge is the first CHC-based model checker for Ada programs, and which yields correct results in more cases than GNATProve. In our experiments GNATProve outputs false positives and false negatives, while AdaHorn neither outputs false positives nor false negatives and, (2) we propose a useful set of Ada programs inspired by C programs from the competition SV-COMP 2017. These Ada programs can pave the way for extending the SV-COMP competition for Ada verification tools.

This paper is organized as follows. In Section 2, we describe the architecture of AdaHorn. In Section 3, we present experimental results. In Section 4, we discuss related work. In Section 5 we draw conclusions and propose future work.

2 Architecture of AdaHorn

Ada is a rich programming language which includes sophisticated features, for instance, protected objects [1] for mutual exclusion problems, or the *Ravenscar profile* [10] for real-time and high-integrity applications. We aim to incrementally develop AdaHorn along with the support for Ada constructs. For the moment we do not support those sophisticated features. In the current version of AdaHorn we support the following Ada constructs:

- Integer, floating-point and boolean data types (and their assignments).
- Self-defined ranges of the above mentioned data types.
- Arrays (array attributes, e.g. 'First, 'Last, etc., are not supported yet).
- While and for loops.
- Procedures and functions (and their respective calls).
- Case and if-then-else statements.
- Assertions.

Currently, AdaHorn supports correctness properties of programs in the form of assertions. In our experiments we use assertions, for instance, to express the following runtime properties: integer and floating-point arithmetic over/underflows, division by zero and array out-of-bounds. In the future we aim to extend the support to temporal properties as in [11, 12].

The architecture of AdaHorn which consists of three main modules: Front-End Module, CHC Generator and Back-End Module, is shown in Figure 1. The internal functionality of each module and the communication among them is written in *Java*. This language, for instance, allows us to access libraries, e.g. *javax.xml*, for translating and manipulating objects in an intermediate representation, e.g. XML. In the following we describe each of those modules as well as their inputs and outputs:

1. *Front-End Module*. This module enables the translation of a collection of input Ada programs (recall that Ada programs consist of specification and implementation files) into an intermediate representation that enables further transformation. In this module we use the *GNAT Compiler*

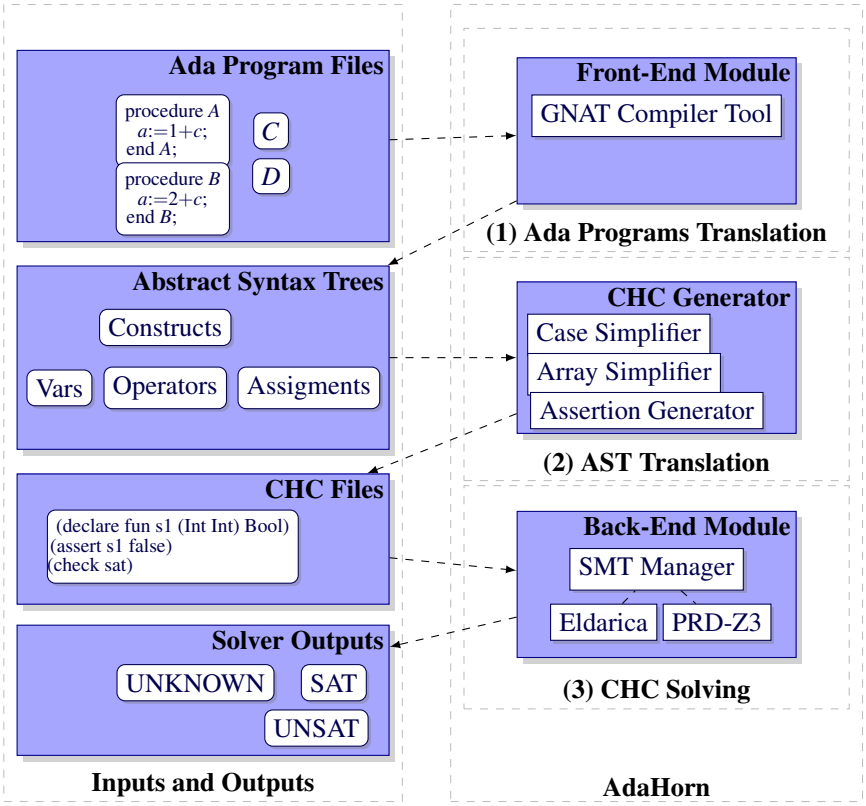

Figure 1: AdaHorn architecture with respective inputs and outputs. Information in inputs and outputs is illustrative. Dashed boxes enclose modules within AdaHorn and their inputs and outputs. Dashed arrows denote exchange of information. Rectangles (within modules) denote tools/submodules and their interactions are denoted with dashed lines. Round-cornered rectangles within Solver Outputs denote messages, while within Ada Program Files, AST and CHC Files denote physical objects.

Tool[3] *(GCT)* to obtain *Abstract Syntax Trees (AST)* from a collection of input Ada programs. The GCT provides functionality to generate XML-based AST from Ada programs where their original behaviour is preserved. At this point we have not applied yet any simplification of constructs for facilitating the generation of CHC.

2. *CHC Generator*. After obtaining XML-based AST from input Ada programs, we translate supported Ada constructs (occurring in XML-based AST) into CHC. In order to facilitate the generation of CHC the following submodules perform the following simplifications:
   (a) *Case Simplifier*. Case statements are manipulated similarly as if-then-else statements. Case statements may use the reserved word *others*, which refers to other values not cased. In CHC the negation of the conjunction of all cased values replaces this reserved word.
   (b) *Array Simplifier*. We perform a *"lazy flattening"* strategy for arrays in CHC. That is, for all arrays defined in input Ada programs we introduce auxiliary variables which are only instantiated with the indices accessed in those programs. As opposed to an *"eager flattening"* strategy, our lazy flattening avoids us to artificially generate CHC where unused array indices

[3] https://www.gnu.org/software/gnat

```
1: with DivZero;
2: procedure gmain
3: is
4: begin
5:   DivZero.Div;
6: end gmain;
```

```
1: package DivZero
2: is
3:  j,k: Integer:= 1;
4:  procedure Div;
5: end DivZero;
```

```
1: package body DivZero;
2: procedure Div is
3:  begin
4:    pragma assert(k ≠ 0);
5:    j := 10/k;
6:  end Div;
7: end DivZero;
```

Figure 2: Ada program *DivByZero* consisting of files: **gmain.adb** (left), **DivZero.ads** (center) and **DivZero.adb** (right).

occur, and which could have a negative impact in the performance of our experiments.

(c) *Assertion Generator*. This submodule automatically generates correctness properties for input programs. These properties are generated in the form of assertions in CHC for all divisions, array accesses and arithmetic operations occurring in input programs. These assertions aim to prove that a particular condition in a given input program holds. For instance, if a division occurs in a given input program, this module will generate an assertion aimed to prove that for that division its denominator is different from zero. Similarly, assertions are generated for array out-of-bounds, and for integer and floating-point arithmetic over/underflows.

3. *Back-End Module*. CHC generated from input Ada programs together with related correctness properties are passed to both solvers, Eldarica and PDR-Z3, whose execution is managed by an *SMT Manager*. The execution of these solvers is bounded by a given amount of seconds provided by the user. Within that bound the solver which is able to firstly conclude *SAT*, *UNSAT* or *UNKNOWN* (for the input CHC) kills the execution of the other. We point out that AdaHorn over-approximates floats with reals for performance purposes of the used solvers. This does not affect the validity of our experimental results.

We would like to remark that it is easy to extend each of the described modules, and the underlying tools can be easily replaced as well.

3 Experiments

In this section we propose a set of Ada programs inspired by C programs from the competition SV-COMP 2017[4], and use them to compare the effectiveness of AdaHorn and GNATProve. Note that for financial reasons we are not able to compare AdaHorn against commercial tools, e.g. Polyspace.

We classify our Ada benchmarks, which contain at most 60 lines of code each, into four different classes: *Arrays*, *Floats*, *Loops*, and *RT-Properties*. The benchmarks in Arrays, Floats and Loops are inspired by programs written in C from SV-COMP 2017. From that competition we have selected benchmarks that exclusively contain the constructs (or equivalent ones) described in Section 2. Note that Ada benchmarks in those classes contain similar assertions as those found in their original counterparts. Note that our Ada benchmarks do not contain library functions, e.g. random number generators, as used in C benchmarks, since AdaHorn offers no support for those functions. Moreover, our Ada benchmarks were adapted accordingly.

[4] https://github.com/sosy-lab/sv-benchmarks

Benchmarks	# Problems	GNATProve				AdaHorn			
		TP	TN	FP	FN	TP	TN	FP	FN
Arrays	8	1	0	7	0	1	7	0	0
Floats	8	1	2	3	2	3	5	0	0
Loops	8	2	1	5	0	2	6	0	0
RT-Properties	8	0	5	2	1	2	6	0	0

Table 1: Evaluation of GNATProve and AdaHorn on Ada benchmarks. Experimental environment: Intel Core $i7$, 64-bit, 2.60Ghz, 16GB, Ubuntu 16.04LTS.

Note that benchmarks in the class RT-Properties are proposed by us. Those benchmarks consist of Ada programs where assertions related to the runtime properties: division by zero, integer and floating-point over/underflows and array out of bounds, are proved. Those assertions could be automatically generated for benchmarks verified by AdaHorn, but not for benchmarks verified by GNATProve. Thus, all assertions for benchmarks in this class were manually generated.

In Figure 2, we show program *DivByZero* which is an example of a benchmark from the class RT-Properties. This program consists of implementation files: gmain.adb and DivZero.adb, and specification file: DivZero.ads. It is easy to see that program DivByZero stores in the integer variable j the result of the division $10/k$, which is 10 since variable k is initialized to 1.

Note that for verifying program DivByZero wrt. division by zero we have manually placed an assertion in line 4 of the file DivZero.adb, right before the division in line 5. This assertion aims to prove that for that division the denominator k is different from zero. This assertion is used by both, AdaHorn and GNATProve for verifying the correctness of program DivByZero.

Evaluation Method. The task of the verification tools is to prove that an assertion occurring in a given Ada benchmark is always true, in which case the tools should return SAT, or demonstrate that it is possible to falsify that assertion, in which case the tools should return UNSAT.

According to the expected result and the result returned by the verification tools, we classify the results of our experiments into the following four categories:

- **True Positive (TP)**: Whenever the expected result is *UNSAT* and the underlying tool returns *UNSAT*. That is, the tool correctly indicates that a given error, e.g. a division by zero, in the program holds.
- **True Negative (TN)**: Whenever the expected result is *SAT* and the underlying tool returns *SAT*. That is, the tool correctly indicates that a given error in the program does not hold.
- **False Positive (FP)**: Whenever the expected result is *SAT* and the underlying tool returns *UNSAT*. That is, the tool wrongly indicates that a given error in the program holds.
- **False Negative (FN)**: Whenever the expected result is *UNSAT* and the underlying tool returns *SAT*. That is, the tool wrongly indicates that a given error in the program does not hold.

Note that false negative results are worse than false positives ones because, for instance, the underlying tool fails to identify an existing error in the program.

Experimental Results. Table 1 summarizes our experimental results with GNATProve and AdaHorn obtained from verifying our Ada benchmarks. In all experiments GNATProve and AdaHorn terminates in less than 30 seconds.

- **GNATProve**: From Table 1 we know that this tool outputs false positives in all classes, 17 false positives to be precise. There are even 3 false negatives output by GNATProve. The following reason can explain these outputs. In our experiments we have noticed that GNATProve analyses programs in isolation. That is, given that an Ada program can consist of specification files (.ads) and implementation files (.adb), GNATProve does not analyse that program as a single unit, but it analyses .ads and .adb files separately (one by one). This strategy of analysing programs in isolation leads to the following false positive.

  If an integer variable is declared and initialised to, say 1, in the .ads file (see line 3 in DivZero.ads from Figure 2), and that variable is nowhere else updated, and used as the denominator of a division occurring in the .adb file (see line 5 in DivZero.adb from Figure 2), that initial value is not used during the analysis, and a false positive regarding a division by zero is output by GNATProve. On the contrary, if that variable is declared and used (in the same way as before) in the same file, say in the .adb, GNATProve outputs no issues regarding a division by zero.

  We have also observed that this strategy leads to the following false negative. Consider program *DivByZeroX* which is program DivByZero from Figure 2 but with variables j and k initialised to 0. When GNATProve verifies DivByZeroX, it does not use the initial value of k, proves the assertion **pragma assert**$(k \neq 0)$, and uses this result in order to prove the division by zero, $j := 10/k$. Clearly, there is a division by zero error in DivByZeroX which is not detected by GNATProve.

  We want to point out that the version of GNATProve used in our experiments is the academic one, and to the best of our knowledge there is no information indicating that the commercial version follows a different strategy.

- **AdaHorn**: Our tool follows a different strategy wrt. the one followed by GNATProve. Given an Ada program, AdaHorn generates CHC for all related .ads and .adb files, and all these generated CHC build a single unit for analysis purposes. In this way we avoid not using information from separate files. Following this strategy, AdaHorn verifies correctly all 32 Ada benchmarks. That is, AdaHorn neither outputs false positives nor false negatives. Note that AdaHorn supports Ada assertions, hence, no modification of Ada programs is required with this regard. Over-approximating floating-point variables with real variables (in CHC) may lead to precision differences. To avoid this, in the future we can integrate the tool *Fluctuat* [13] in the Back-End Module, and use it to support more involved floating-point related properties.

4 Related Work

Horn Clauses Based Tools. We are directly inspired by the successful approach taken by SeaHorn and JayHorn wrt. CHC-based verification of programs. The suitable combination of front-ends and back-ends enables the development of model checking tools that are both, efficient and modular. Comparing AdaHorn to SeaHorn and JayHorn, all three tools are designed to be modular which facilitates the replacement of underlying tools in the front and back-end.

JayHorn supports more sophisticated constructs, e.g. objects and exception handling methods, than AdaHorn. Objects and exception handling methods also exist in Ada, however, supporting those constructs are considered future work. One key design decision that JayHorn follows, is the control-flow simplification and translation for exception handling performed in Java programs. Since these simplification and translation steps may change the underlying behaviour of programs, JayHorn applies automatic tests to show that the underlying behaviour has not been changed. Currently, AdaHorn simplifies case

statements and arrays without modifying the underlying behaviour of programs, as we incrementally support more sophisticated Ada constructs, we may in the future perform more simplification of constructs. Introducing automatic tests is an interesting idea to show that the underlying behaviour of programs is preserved.

SeaHorn supports as well other sophisticated constructs, e.g. pointers, that also exists in Ada, and implements different techniques, e.g. *abstract interpretation*, for verifying programs. We would like to point out that SeaHorn and JayHorn are more mature tools than AdaHorn, and this can explain the support given for sophisticated constructs.

A translation from Ada programs to C or Java programs may seem reasonable given the level of maturity of both tools. However, even for the basic constructs that AdaHorn is able to support, for instance, translating self-defined ranges for data types, to constructs in those other languages is not straightforward, and raise questions regarding the preservation of the underlying semantics of Ada programs. Moreover, sophisticated Ada constructs like protected objects may have no "semantically equivalent" constructs in Java or C. Thus, we believe that intermixed translations, e.g. from Ada to Java, should be avoided.

5 Conclusions and Future Work

We introduced AdaHorn, a model checker for verification of Ada programs wrt. correctness properties. AdaHorn translates Ada programs together with their related correctness properties into a set of Constrained Horn Clauses which are solved by Eldarica and PDR-Z3. We proposed a set of Ada programs inspired by C programs from the competition SV-COMP, and use them to compare the effectiveness of AdaHorn and GNATProve. Our experiments showed that AdaHorn outperforms GNATProve by yielding correct results in more cases than GNATProve. In our experiments, GNATProve outputs a high number of false positives and false negatives. AdaHorn neither outputs false positives nor false negatives.

Future Work: We will continue our work in the following directions:

- We want to extend the support for Ada constructs used in concurrent programs, as well as extending the support beyond assertion-based properties, for instance, for proving termination or deadlock freedom.
- We want to integrate the tool Fluctuat in the Back-End Module, and use it to support more involved floating-point related properties.
- Currently AdaHorn relies on the GCT to obtain XML-based AST from input Ada programs. Since Ada supports the programming in the large paradigm, for huge programs (millions of lines of code) reading XML-based AST may be inefficient. We could increase efficiency by directly accessing AST through *Ada Semantic Interface Specification (ASIS)* [14] libraries.

References

[1] S. T. Taft, R. A. Duff, R. Brukardt, E. Plödereder, P. Leroy, and E. Schonberg. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E)*, volume 8339 of *LNCS*. Springer, 2013.

[2] J. G. P. Barnes. *High Integrity Software - The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[3] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. *Horn Clause Solvers for Program Verification*. volume 9300 of *LNCS*. Springer, 2015.

[4] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. *Horn Clauses as an Intermediate Representation for Program Analysis and Transformation*. *TPLP*, 15(4-5), 2015.

[5] N. Bjørner, K. L. McMillan, and A. Rybalchenko. *Program Verification as Satisfiability Modulo Theories*. In *SMT 2012*, volume 20 of *EPiC Series in Computing*. EasyChair, 2012.

[6] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. *A Verification Toolkit for Numerical Transition Systems - Tool Paper*. volume 7436 of *LNCS*. Springer, 2012.

[7] L. Mendonça de Moura and N. Bjørner. *Z3: An Efficient SMT Solver*. volume 4963 of *LNCS*. Springer, 2008.

[8] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. *The SeaHorn Verification Framework*. volume 9206 of *LNCS*. Springer, 2015.

[9] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. *JayHorn: A Framework for Verifying Java Programs*. volume 9779 of *LNCS*. Springer, 2016.

[10] A. Burns, B. Dobbing, and G. Romanski. *The Ravenscar Tasking Profile for High Integrity Real-Time Programs*. volume 1411 of *LNCS*. Springer, 1998.

[11] T. A. Beyene, C. Popeea, and A. Rybalchenko. *Efficient CTL Verification via Horn Constraints Solving*. volume 219 of *EPTCS*, 2016.

[12] T. A. Beyene, M. Brockschmidt, and A. Rybalchenko. *CTL+FO Verification as Constraint Solving*. In *SPIN 2014*. ACM, 2014.

[13] E. Goubault and S. Putot. *Static Analysis of Finite Precision Computations*. In *VMCAI 2011*, volume 6538 of *LNCS*. Springer, 2011.

[14] C. Colket. *Ada Semantic Interface Specification (ASIS): Frequently Asked Questions*. *Ada Letters*, XV(4), July 1995.