Verification of Ada Programs with AdaHorn

Tewodros A. Beyene, Christian Herrera, Vivek Nigam
fortiss GmbH, Forschungsinstitut des Freistaats Bayern für softwareintensive Systeme und Services
Guerickestr. 25, 80805 München, Germany; email: {beyene, herrera, nigam}@fortiss.org

Abstract

We propose AdaHorn*, a model checker for verification of* Ada *programs with respect to correctness properties given as assertions.* AdaHorn *translates an Ada program together with its assertion into a set of Constrained Horn Clauses, and feeds it to a Horn constraints solver. We evaluate the performance of* AdaHorn *on a set of Ada programs inspired by C programs from the software verification competition (SV-COMP). Our experimental results show that* AdaHorn *outputs correct results in more cases than GNATProve, which is a widely used Ada verification framework.*

Keywords: Ada Verification, Model Checking, Horn Constraints Solving.

1 Introduction

Ada [1] is widely used by systems developers in the avionics, space, military and railways domains due to its features like strong typing, explicit concurrency, support for design-by-contract, non-determinism, etc., that enable developers to build robust and dependable safety critical systems. Prominent Ada analysis tools that support the development of dependable systems in Ada include *GNATProve* [2] and *Polyspace* [3]. These tools perform static analysis on Ada programs for detecting runtime errors, such as *array out-of-bounds*, *arithmetic overflow* and *division by zero*. It is known that static analysis tools often yield *false positives*, i.e. wrongly concluding that errors occur in a program, and sometimes even *false negatives*, i.e. wrongly concluding that a program does not have any error.

In this work, we aim to advance the support available for the analysis and verification of Ada programs by proposing AdaHorn, a Horn constraints-based model checker for verifying Ada programs with respect to correctness properties written as assertions. Similar to the *SeaHorn* [4] and *JayHorn* [5] frameworks, which respectively verify *C* and *Java* programs, AdaHorn translates Ada programs into a set of Constrained Horn Clauses (CHCs) [6, 7, 8] which are solved by well-known constraint solvers such as Eldarica [9] and Z3 [10]. In general, a CHC correspond to a clause with at most one positive occurrence of an uninterpreted predicate. One can also think of a CHC as a fragment of first-order formulas modulo background theories, where its constraints are formulated using a given background theory [6].

In this work, AdaHorn supports a small but non-trivial subset of Ada data types, namely integer, floating-point and boolean data types as well as arrays of these types, and Ada program constructs, which include *procedures*, *functions*, *for/while loops*, *if-then-else statements*, *case statements*, *procedure/-function calls* and *assertions*.

The contribution of this work is twofold: (1) AdaHorn, which is the first Horn constraints-based model checker for Ada programs. (2) A Horn constraints generator for Ada programs, that takes Ada programs as input and produces a set of CHCs. This makes various Horn constraints-based program analysis, verification and synthesis techniques available to Ada programs.

2 Preliminiaries

In this section, we introduce the set of Constrained Horn Clauses (CHC) that AdaHorn uses as an intermediate language to encode an Ada verification problem. We also discuss the class of Ada programs that can be handled by the current implementation of AdaHorn.

2.1 Constrained Horn Clauses

In general, a Constrained Horn Clause correspond to a clause that has at most one positive occurrence of an uninterpreted predicate. One can also think of a Constrained Horn Clause as a fragment of first-order formulas modulo background theories, where its constraints are formulated using a given background theory [6]. We use the symbol $\mathcal{A}$ to denote a background theory. In this paper, we let $\mathcal{A}$ be quantifier-free linear arithmetic.

Definition 1. CHC is defined by the following grammar:

$$
\begin{aligned}
\Pi &::= HC \wedge \Pi \mid \top \\
HC &::= \forall vars : body \rightarrow head \\
pred &::= upred \mid \Phi \\
head &::= pred \\
body &::= \top \mid pred \mid body \wedge body \\
vars &::= \text{the set of all variables in a given clause} \\
upred &::= \text{an uninterpreted predicate applied to terms} \\
\Phi &::= \text{a formula whose terms and predicates are interpreted over } \mathcal{A}
\end{aligned}
$$

We use Π to denote a set (conjunction) of CHC, while HC refers to a single Constrained Horn Clause.

2.2 Classes of Target Ada Programs

The current implementation of AdaHorn does not support all language features and constructs of Ada. However, all basic constructs of Ada that can be used to write programs of medium complexity are supported. These include (1) integer, floating-point and boolean data types, and self-defined ranges over these types, (2) arrays, (3) assertions, (4) while and for loops, (5) procedures and functions (together with their corresponding calls), and (6) if-then-else statements.

The complete class of supported constructs is given in Figure 1, which presents the grammar for the set of supported Ada programs. In that grammar, *range* stands for a finite range of the underlying data type. An example integer range can be $0..4$. *Id* corresponds to a typical identifier that can be used as function names, variable reference, etc. The symbol * denotes the Kleene closure of the underlying grammar item.

We aim to develop AdaHorn further by incrementally adding support not only for more constructs but additional Ada language features. One candidate language construct to add is *protected object* [1], which is useful for mutual exclusion problems. Similarly, a candidate language feature to consider in the future is the *Ravenscar profile* [11], which is useful for real-time and high-integrity applications.

3 Architecture of AdaHorn

Inspired by similar approaches for verifying *C* and *Java* programs [4, 5], the architecture of AdaHorn consists of three layers. The architecture is shown in Figure 2.

- *Front-end*: This layer makes use of the *gnat2xml* utility in the *GNAT Compiler Tool* [12] to obtain an XML serialisation of an abstract syntax tree of the input Ada program.
- *Middle-end (glue code)*: In this layer, AdaHorn takes the abstract syntax tree generated by the *gnat2xml* utility as input, and translates them into a set of CHCs. AdaHorn performs a top-down, recursive descent through the syntax tree of the given Ada program. It introduces auxiliary predicates and generates a set CHCs over these predicates.

  The interested reader finds in [6] a reference for translating program constructs into CHCs. Moreover, this step needs to also make special considerations for constructs that are not directly supported by the Horn constraints solvers used in this work. For example, as arrays may result in Horn clauses with nonlinear structure or higher orders, further processing needs to be done to translate resulting constraints into array-free Horn constraints [13].

  As this constraints generating middle-end layer, also called the glue code, is the main contribution of this work, it is explained in more detail in Section 4.
- *Back-end*: This layer takes the generated CHCs as input and pass them to a CHC solver to get a result, which is the final result of AdaHorn's model checking procedure. Any CHC solver can be employed in principle. However, we have used the Eldarica and Z3 solvers in this work. Possible results of the solving step are: SAT(the CHCs are satisfiable), UNSAT(the CHCs are unsatisfiable), and UNKNOWN(the solver is not able to output any conclusive result).

⟨*program*⟩ ::= ⟨*with_use*⟩* ⟨*pkg*⟩ | ⟨*with_use*⟩* ⟨*pkg_body*⟩ | ⟨*with_use*⟩* ⟨*proc_fun_body*⟩
⟨*with_use*⟩ ::= 'with' ⟨*id*⟩';' 'use' ⟨*id*⟩';'
⟨*pkg*⟩ ::= 'package' ⟨*id*⟩ 'is' ⟨*decl*⟩* ⟨*proc_fun_decl*⟩* 'end' ⟨*id*⟩';'
⟨*pkg_body*⟩ ::= 'package body' ⟨*id*⟩ ⟨*proc_fun_body*⟩* 'end' ⟨*id*⟩';'
⟨*proc_fun_body*⟩ ::= ⟨*proc_body*⟩ | ⟨*fun_body*⟩
⟨*proc_body*⟩ ::= 'procedure' ⟨*id*⟩ ['('⟨*formal_part*⟩')'] 'is' ⟨*decl*⟩* ⟨*proc_fun_body*⟩* 'begin' ⟨*stmt*⟩* 'end' ⟨*id*⟩';'
⟨*fun_body*⟩ ::= 'function' ⟨*id*⟩ ['('⟨*formal_part*⟩')'] 'return' ⟨*type*⟩ 'is' ⟨*decl*⟩* ⟨*proc_fun_body*⟩* 'begin' ⟨*stmt*⟩* 'return' ⟨*stmt_rtn*⟩';' 'end' ⟨*id*⟩';'
⟨*formal_part*⟩ ::= ⟨*formal_param_spec*⟩ (';' ⟨*formal_param_spec*⟩)*
⟨*formal_param_spec*⟩ ::= ⟨*var*⟩ (',' ⟨*var*⟩)* ':' ⟨*mode*⟩ ⟨*type*⟩ [':=' ⟨*expr*⟩]
⟨*decl*⟩ ::= ⟨*var*⟩ ':' 'array' '(' **range** ')' 'of' ⟨*type*⟩ [':=' '(' **initial values for array** ')'] ';' | ⟨*var*⟩ ':' ⟨*type*⟩ [':=' ⟨*expr*⟩]';'
⟨*var*⟩ ::= ⟨*id*⟩
⟨*proc_fun_decl*⟩ ::= 'procedure' ⟨*id*⟩ '('⟨*formal_part*⟩')'';' | 'function' ⟨*id*⟩ '('⟨*formal_part*⟩')' 'return' ⟨*type*⟩';'
⟨*mode*⟩ ::= 'in' | 'in out' | 'out'
⟨*type*⟩ ::= 'Integer' | 'Float' | 'Boolean'
⟨*expr*⟩ ::= arithmetic logical expression
⟨*stmt*⟩ ::= ⟨*var*⟩ ':=' ⟨*expr*⟩ | ⟨*stmt1*⟩ ';' ⟨*stmt2*⟩ | 'pragma assert' '('⟨*expr*⟩')' ';' | 'if' ⟨*expr*⟩ 'then' ⟨*stmt1*⟩ 'else' ⟨*stmt2*⟩ 'end if' ';' | ⟨*proc_fun_call*⟩ ';' | 'while' ⟨*expr*⟩ 'loop' ⟨*stmt*⟩ 'end loop' ';' | 'for' ⟨*id*⟩ 'in' ⟨*type*⟩ 'range' **range** 'loop' ⟨*stmt*⟩ 'end loop'';' | 'case' ⟨*expr*⟩ 'is' ⟨*case_option*⟩ (⟨*case_option*⟩*) 'end case' ';'
⟨*proc_fun_call*⟩ ::= procedure_name | procedure_name '('⟨*actual_part*⟩')' | function_name | function_name '('⟨*actual_part*⟩')'
⟨*actual_part*⟩ ::= ⟨*actual_param_spec*⟩ (',' ⟨*actual_param_spec*⟩)*
⟨*actual_param_spec*⟩ ::= ⟨*expr*⟩ | ⟨*var*⟩ | function_name | function_name '('⟨*actual_part*⟩')'
⟨*stmt_rtn*⟩ ::= ⟨*var*⟩ ':=' ⟨*expr*⟩ | ⟨*expr*⟩
⟨*case_option*⟩ ::= 'when' ⟨*discrete_choice*⟩ '=>' ⟨*stmt*⟩ ';'
⟨*discrete_choice*⟩ ::= ⟨*expr*⟩ | **range** | 'others'

Figure 1: Grammar for supported Ada programs.

CHCs have enjoyed recent success as languages of intermediate representation, and a promising set of frameworks for verification tasks ranging from temporal verification to synthesis and game solving have recently been proposed [14, 15, 16]. The Horn constraints generator alone in AdaHorn can also be useful in making all these Horn constraints-based verification and synthesis technologies available to Ada programmers and verification engineers. AdaHorn is implemented in Java.

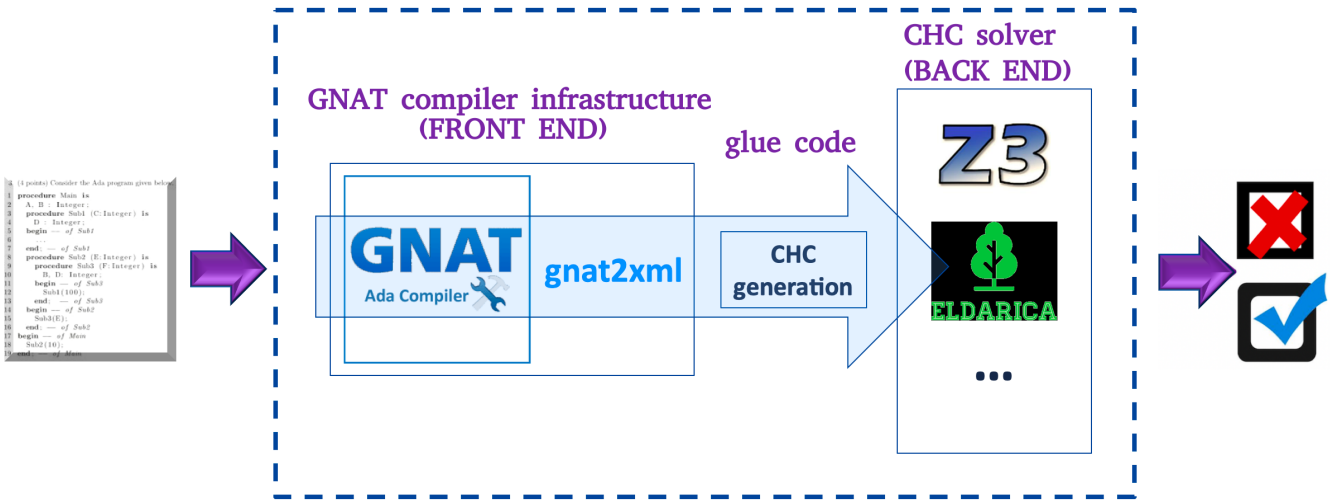

Figure 2: Architecture of AdaHorn

4 Constraints Generation

In this section, we describe AdaHorn's constraint generation process and illustrate how supported Ada constructs are encoded as CHCs by AdaHorn.

There are two considerations in the Ada to CHC translation:

1. The first one is encoding program states, which are valuations of program variables at certain critical locations of the program. These include loop entries and exits, procedure call and return locations, function call and return locations, etc.
2. The second one is encoding state transitions that occur during the execution of the program by translating involved Ada constructs into their corresponding CHC.

In the rest of this section, we illustrate the constraint generation process using two simple Ada programs that contain non-trivial constructs from the grammar in Figure 1. Both of our example Ada programs are structured into 3 files; a main Ada project file, a specification file and an implementation file. The constraint generation procedure needs to take into consideration all three files to generate a collective set of CHCs. We provide the corresponding CHCs encoding for each example, together with discussion on how the translation is done for supported Ada language constructs. AdaHorn generates CHC in the *SMT-Lib Standard* [17].

Example 1. Our first example Ada program, which is shown in Figure 3, simply increments an input integer value by 10 and returns the result. As mentioned above, this program is structured into a project file (*Figure 3a*), an implementation file (*Figure 3b*), and a specification file (*Figure 3c*).

The program initialises its two integer variables, *res* and *x*, to 0 (*line* 3 *in Figure 3a*), and calls the function *sum* over *x* (*line* 5 *in Figure 3a*). Note that the specification file for the function *sum* is given in Figure 3c, and its corresponding implementation file is given in Figure 3b. Finally, the value

```
1  with prog1;
2  procedure gmain is
3     res, x: Integer := 0;
4  begin
5     res := prog1.sum(x);
6  end gmain;
```

(a) project file

```
1  package prog1 is
2     function sum(j: in out Integer)
3     return Integer;
4  end prog1;
```

(b) specification file

```
1  package body prog1;
2     function sum (j: in out Integer)
3     return Integer is
4     begin
5        return j+10;
6     end sum;
7  end prog1;
```

(c) implementation file

Figure 3: A simple example program.

returned by the function is stored in the variable *res* (*line* 5 *in Figure 3a*).

The set of CHCs generated by AdaHorn for this program is given in Figure 4. Each state of the Ada program is encoded as an uninterpreted predicate over the program variables. For example, the predicate $gmain\_call\_init$, which is defined over the empty set of variables, encodes the state from which the program starts execution. The start of the program execution is encoded as the clause $(assert\ (=>\ true\ gmain\_call\_init))$ with just $true$ in the body (*line 8*). Once the program starts running, the first statement it executes is initialisation of its variables *res* and *x* to *0* (*line 9*). The call to the function *sum* (line 5 in Figure 3a) is encoded in *line 10*. The clause in *line 12* of Figure 4 encodes the addition operation in *return j+10* (line 5 in Figure 3b) of the Ada program. The return statement itself is encoded In *line 12*, where that result of the sum operation is stored in the auxiliary variable $\_RetV$. The value of auxiliary variable $\_RetV$ is assigned back to the variable *res* of the caller *gmain* function

```
1   ( declare -fun gmain_s0 (Int  Int ) Bool)
2   ( declare -fun gmain_s1 (Int  Int ) Bool)
3   ( declare -const gmain_call_init  Bool)
4   ( declare -const gmain_call_end Bool)
5   ( declare -fun Prog1_Sum_s0 (Int) Bool)
6   ( declare -fun Prog1_Sum_call_init ( Int ) Bool)
7   ( declare -fun Prog1_Sum_call_ret (Int ) Bool)
8   ( assert  (=> true  gmain_call_init ))
9   ( assert  ( forall  (( res  Int ) (x  Int )) (=> (and (= res  0)
        (= x  0)  gmain_call_init ) (gmain_s0 res  x))))
10  ( assert  ( forall  (( res  Int ) (x  Int )) (=> (gmain_s0 res  x)
        (Prog1_Sum_call_init  x))))
11  ( assert  ( forall  ((j  Int )) (=> (Prog1_Sum_call_init  j) (
        Prog1_Sum_s0 j))))
12  ( assert  ( forall  ((j  Int ) (_RetV Int )) (=> (and (= _RetV
        (+ j  10)) (Prog1_Sum_s0 j)) (Prog1_Sum_call_ret
        _RetV))))
13  ( assert  ( forall  (( res  Int ) (x  Int ) ( res ' Int ) (_RetVV
        Int))(=> (and (gmain_s0 res  x) (Prog1_Sum_call_ret
        _RetVV) (= res '  _RetVV)) (gmain_s1 res '  x))))
14  ( assert  ( forall  (( res  Int ) (x  Int )) (=> (gmain_s1 res  x)
        gmain_call_end)))
```

Figure 4: Generated CHCs for the program in Figure 3

```
1  with prog2;
2  procedure gmain is
3  begin
4     prog2. initArray ;
5  end gmain;
```

(a) project file

```
1  package prog2 is
2     procedure initArray ;
3  end prog2;
```

(b) specification file

```
1   package body prog2;
2      procedure initArray
3         arr  :  array  (1..10)  of  Integer ;
4         temp:  Integer ;
5      begin
6         arr (1):= 0;
7         temp:= arr (1) ;
8         arr (3):= temp;
9         pragma Assert ( arr (1)  =  arr (3) ) ;
10     end  initArray ;
11  end prog2;
```

(c) implementation file

Figure 5: A program with array read and write operations.

as encoded in *line 13*. The constraint in *line 14* denotes the end of the call to procedure *gmain*.

Example 2: In this example, we show how a set of CHC is generated for a program with arrays. The program is given in Figure 5. Like the previous example, here also we have three files. Our main focus, however, will be on the implementation file (*Figure 5c*) as it contains all the important variables and statements of the program from users' perspective. The program is defined over an array variable *arr*, which defines an array of length 10, and an integer variable *temp*. During execution, the program calls the *initArray* procedure which is specified and implemented in the package *prog2*. As shown in Figure 5c, the procedure does three assignments: the assignments *arr(1):= 0;* and *arr(3):= temp;* (on lines 6 and 8) involve array write operations, whereas the assignment *temp:=arr(1)* (on line 7) involves array read operation. Finally, as a correctness property the procedure ensures the array has equal values on indices 1 and 3 with the assert statement *pragma Assert (arr(1) = arr(3))*.

This example is particularly interesting as verification of many properties over arrays often require inferring universally quantified invariants over arrays. However, no general algorithms exist for checking if such universally quantified array invariants hold, let alone inferring them. In this paper, we have applied a system of rules for transforming atomic array read and write statements into a system of non-linear Horn Clauses over scalar variables only [13]. We find this approach efficient compared to other approaches for handling arrays as it does not introduce a new abstract domain or a new interpolation procedure for arrays. Instead, it generates an abstraction as a scalar problem, that can be fed to any solver that can handle non-linear Horn Clauses.

The set of CHCs for this example is given in *Figure 6*. The declarations for the boolean variables and predicates used in the generated CHCs are given in *lines 1 - 9*. The first important clause to consider is at *line 13* that defines the predicate *Prog2_InitArray_s0*. This predicate represents the state of the program before the input array variable arr is updated. The auxiliary variable *arrIND* denotes an index of *arr*. Note that this index takes only the values of the indices of the array accessed in the corresponding Ada program. As the aim of this example is to illustrate the handling of arrays during CHCs generation, our focus will be the array read and write operations in the example Ada program.

Let us now discuss the crux of the implemented array handling method. A predicate *pred(arr, temp)*, such as the one encoding the states for our example program, is assumed to internally have a table structure with a set of entries $(ind_1, val_1, temp)$, $(ind_2, val_2, temp)$, $\ldots (ind_n, val_n, temp)\}$, where $arr[ind_i] = val_i$ for each $1 \leq i \leq n$. However, we do not want to flatten the array eagerly. Rather, what we want to do is to keep the array as abstract as possible and refer to concrete values only when the need arises. With this in mind, let us try to see how we handle array read and write operations one by one:

- write operation: Assume we want to encode $arr[5] := 10$. Here we will replace val_5 by 10 in $(ind_5, val_5, temp)$, and for the rest of the entries, i.e., $(ind_j, val_j, temp)$ for each $j \neq 5$, val_j stays the same. Let us assume predicates $pred_0$ and $pred_1$ represent states of the program before and after the write operation $arr[5] := 10$. Our encoding of the write operation consists of two CHCs: (1) $pred_0(ind, val, temp) \wedge ind = 5 \rightarrow pred_1(ind, 10, temp)$ (2) $pred_0(ind, val, temp) \wedge ind \neq 5 \rightarrow pred_1(ind, val, temp)$. In our example, there are the two write operations on lines 6 and 8 in Figure 5. Their corresponding CHCs can be found on lines 14-15 and 18-19 in Figure 6, respectively.
- read operation: Assume we want to encode $temp := arr[5]$. Here we will have to replace $temp$ by val_5

```
1   ( declare -const gmain_s0 Bool)
2   ( declare -const gmain_s1 Bool)
3   ( declare -const gmain_call_init Bool)
4   ( declare -const gmain_call_end Bool)
5   ( declare -fun Prog2_InitArray_s0 ( Int Int ) Bool)
6   ( declare -fun Prog2_InitArray_s1 ( Int Int ) Bool)
7   ( declare -fun Prog2_InitArray_s2 ( Int Int ) Bool)
8   ( declare -const Prog2_InitArray_call_init Bool)
9   ( declare -const Prog2_InitArray_call_end Bool)
10  ( assert (= true gmain_call_init ))
11  ( assert (= gmain_call_init gmain_s0))
12  ( assert (= gmain_s0 Prog2_InitArray_call_init ))
13  ( assert ( forall ((arrIND Int ) ( arr Int ) (temp Int ))(=>
          Prog2_InitArray_call_init (Prog2_InitArray_s0
        arrIND arr temp))))
14  ( assert ( forall ((arrIND Int ) ( arr Int ) ( arr' Int ) (
        temp Int ))(=> (and (Prog2_InitArray_s0 arrIND arr
        temp)(= arrIND 1)(= arr' 0))(Prog2_InitArray_s1
        arrIND arr' temp))))
15  ( assert ( forall ((arrIND Int ) ( arr Int ) ( arr' Int ) (
        temp Int ))(=> (and (Prog2_InitArray_s0 arrIND arr
        temp)(not (= arrIND 1))) (Prog2_InitArray_s1
        arrIND arr temp))))
16  ( assert ( forall ((arrIND Int ) ( arr Int ) (temp Int ) (
        temp' Int )) (=> (and (Prog2_InitArray_s1 arrIND
        arr temp)(= arrIND 1)(= temp' arr ))(
        Prog2_InitArray_s3 arrIND arr temp'))))
17  ( assert ( forall ((arrIND1 Int ) (arrIND2 Int ) ( arr2 Int )
        (arr1 Int ) (temp Int ) (temp' Int ))(=> (and (
        Prog2_InitArray_s1 arrIND1 arr1 temp)(
        Prog2_InitArray_s1 arrIND2 arr2 temp)(= arrIND 1)
        (not (= arrIND1 arrIND2))(= temp' arr1 ))(
        Prog2_InitArray_s3 arrIND2 arr2 temp'))))
18  ( assert ( forall ((arrIND Int ) ( arr Int ) (temp Int ) ( arr'
         Int )) (=> (and (Prog2_InitArray_s3 arrIND arr
        temp)(= arrIND 3)(= arr' temp))(Prog2_InitArray_s4
         arrIND arr' temp))))
19  ( assert ( forall ((arrIND Int ) ( arr Int ) ( arr' Int ) (
        temp Int )) (=> (and (Prog2_InitArray_s3 arrIND arr
         temp)(not (= arrIND 3))) (Prog2_InitArray_s4
        arrIND arr temp))))
20  ( assert ( forall ((arrIND1 Int ) (arrIND2 Int ) (arr1 Int )
         (arr2 Int ) (temp Int )) (=> (and (
        Prog2_InitArray_s4 arrIND1 arr1 temp) (
        Prog2_InitArray_s4 arrIND2 arr2 temp) (= arrIND1
        1)(= arrIND2 3))(= arr1 arr2 ))))
21  ( assert ( forall ((arrIND Int ) ( arr Int ) (temp Int )) (=>
        (Prog2_InitArray_s4 arrIND arr temp)
        Prog2_InitArray_call_end )))
22  ( assert (= Prog2_InitArray_call_end gmain_s1))
23  ( assert (= gmain_s1 gmain_call_end))
```

Figure 6: Corresponding set of CHC for the Ada program in Figure 5.

in $(ind_i, val_i, temp)$, for all i. Note that since $temp$ is not an array variable, its value is independent of an index. Let's assume predicates $pred_0$ and $pred_1$ represent states of the program before and after the read operation. Our encoding of the read operation consists of two CHCs: (1) $pred_0(ind, val, temp) \wedge ind = 5 \rightarrow pred_1(ind, val, val)$: this constraint ensures that if the entry has the index involved in the read, we simply copy val to $temp$. (2) $pred_0(ind_1, val_1, temp) \wedge pred_0(ind_2, val_2, temp) \wedge ind1 \neq 5 \wedge ind2 = 5 \rightarrow pred_1(ind_1, val_1, val_2)$: this constraint ensures that if the entry does not have the index involved in the read, it looks for another entry with the index involved in the read operation, and then it copies $val2$ from the other entry to $temp$. Note that this constraint is non-linear as there are two instances of $pred_0$ in the body of the CHC. This is where the role of non-linear Horn constraints encoding comes into play. In our example, there is a read operation on lines 7 in Figure 5, and its corresponding CHCs can be found on lines 16-17 in Figure 6.

5 Experiments

We evaluate AdaHorn on a set of Ada benchmarks that consists of four categories of programs: *arrays*, *floats*, *loops*, and *simple* [1]. The first three categories are inspired by C programs from the software verification competition SV-COMP 2017 [18]. For the C programs that can exclusively be translated to the subset of Ada handled in this work, we have manually created equivalent Ada programs. Programs in the *simple* category are written by the authors of this paper.

The arrays and floats categories contain programs whose assertion involves array and float variables, respectively. In the loops category, the program assertions are placed within while and for loop constructs. Programs in the simple category were constructed with integer variables and without any complex Ada construct like loop or logic statements. While assertions in the first three categories do not capture specific classes of properties, programs in the simple category contain assertion that captures runtime properties such division by zero, integer and floating-point over(under)-flow and array out of bounds.

The verification task is to prove if an assertion placed in a given program is valid or not. A timeout is reached if the verification task can not complete in 1000 seconds (indicated by **TO**). Results for each tool are classified into one of the following four classes by comparing it with the expected result: (1) **True Positive (TP)** - tool correctly indicates an assertion is not valid, (2) **True Negative (TN)** - tool correctly indicates an assertion is valid, (3) **False Positive (FP)** - tool wrongly indicates an assertion is not valid, and (4) **False Negative (FN)** - tool wrongly indicates an assertion is valid. In addition, AdaHorn may not be able to solve its generated constraints leaving the final result unknown (indicated by **UN**).

Coming to the results, GNATProve takes less than 3 seconds for verifying each benchmark, whereas AdaHorn timed out for one example and took less than 60 seconds to verify the remaining examples. GNATProve concludes false positives in 48 occasions (out of 68 benchmarks) and 2 false negatives! After reporting those false negatives to developers of GNATProve, we were told that GNATProve generally does not output correct results if intermediate checks have failed, which is the case for the programs related to those false negatives. This is mainly due to GNATprove's assumption that any prior check must be correct in order to prove the next ones. While AdaHorn does not result in any false negatives, it results in 4 false positives. The reason behind the false positives was the difference in precisions for floating-point numbers between the Ada compiler and the Horn constraint solvers used by

[1] https://bitbucket.org/umaya/adabenchmarksfromsvcomp17

benchmark category	number of programs	GNATProve					AdaHorn					
		TP	TN	FP	FN	TO	TP	TN	FP	FN	TO	UN
arrays	20	1	0	19	0	0	8	10	1	0	1	0
floats	20	1	4	13	2	0	5	8	3	0	0	4
loops	20	4	2	14	0	0	7	13	0	0	0	0
simple	8	0	5	2	0	0	2	6	0	0	0	0

Table 1: Comparison of results between GNATProve and AdaHorn

AdaHorn (Eldarica and Z3). AdaHorn, however, is not able to verify 4 benchmarks due to failure of the Horn constraints solvers to conclude whether input CHCs are satisfiable or unsatisfiable.

Finally, we would like to point out that AdaHorn has been created as a subproject in the context of a cooperation with one of our partners from the aviation industry. As a next step in our cooperation we plan to evaluate AdaHorn on our partner's industrial Ada code.

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)*, vol. 8339 of *LNCS*. Springer, 2013.

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

[3] "Polyspace." `https://www.mathworks.com/products/polyspace.html`.

[4] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "*The SeaHorn Verification Framework*," vol. 9206 of *LNCS*, pp. 343–361, Springer, 2015.

[5] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf, "*JayHorn: A Framework for Verifying Java Programs*," vol. 9779 of *LNCS*, pp. 352–358, Springer, 2016.

[6] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko, "*Horn Clause Solvers for Program Verification*," vol. 9300 of *LNCS*, pp. 24–51, Springer, 2015.

[7] 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*, vol. 15, no. 4-5, pp. 526–542, 2015.

[8] N. Bjørner, K. L. McMillan, and A. Rybalchenko, "*Program Verification as Satisfiability Modulo Theories*," in *SMT 2012*, vol. 20 of *EPiC Series in Computing*, pp. 3–11, EasyChair, 2012.

[9] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer, "*A Verification Toolkit for Numerical Transition Systems - Tool Paper*," vol. 7436 of *LNCS*, pp. 247–251, Springer, 2012.

[10] L. M. de Moura and N. Bjørner, "*Z3: An Efficient SMT Solver*," vol. 4963 of *LNCS*, pp. 337–340, Springer, 2008.

[11] A. Burns, B. Dobbing, and G. Romanski, "*The Ravenscar Tasking Profile for High Integrity Real-Time Programs*," vol. 1411 of *LNCS*, pp. 263–275, Springer, 1998.

[12] Project Hi-Lite / GNATprove, 2014.

[13] D. Monniaux and L. Gonnord, "*An Encoding of Array Verification Problems into Array-Free Horn Clauses*," *CoRR*, vol. abs/1509.09092, 2015.

[14] T. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko, "A constraint-based approach to solving games on infinite graphs," *SIGPLAN Not.*, vol. 49, pp. 221–233, Jan. 2014.

[15] T. A. Beyene, C. Popeea, and A. Rybalchenko, "*Efficient CTL Verification via Horn Constraints Solving*," vol. 219 of *EPTCS*, pp. 1–14, 2016.

[16] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, *Horn Clause Solvers for Program Verification*, pp. 24–51. Cham: Springer International Publishing, 2015.

[17] C. Barrett, P. Fontaine, and C. Tinelli, "*The SMT-LIB Standard: Version 2.6*," tech. rep., Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[18] D. Beyer, "*Software Verification with Validation of Results - (Report on SV-COMP 2017)*," in *TACAS 2017*, vol. 10206 of *LNCS*, pp. 331–349, 2017.