fortiss

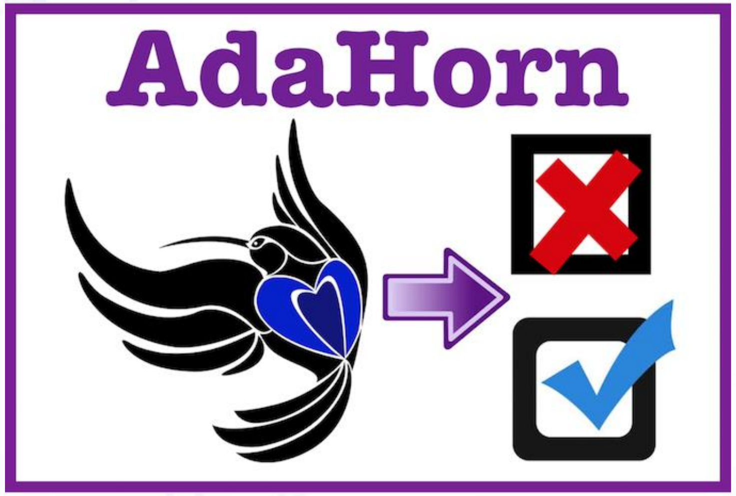
Verification of Ada Programs with AdaHorn

T.A. Beyene, C. Herrera and V. Nigam

Ada-Europe

24th International Conference on Reliable Software Technologies

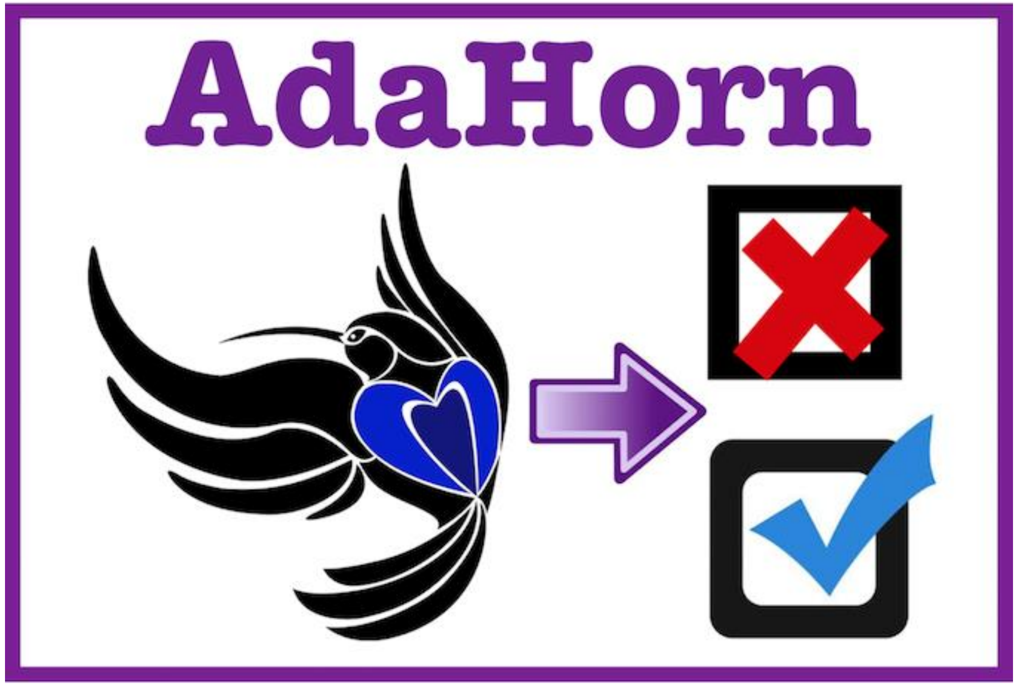
11-14 June 2019, Warsaw, Poland



AdaHorn

An Ada verification tool based which uses ASIS-based Ada compiler infrastructure as a frontend and horn constraints solving technology as a backend.

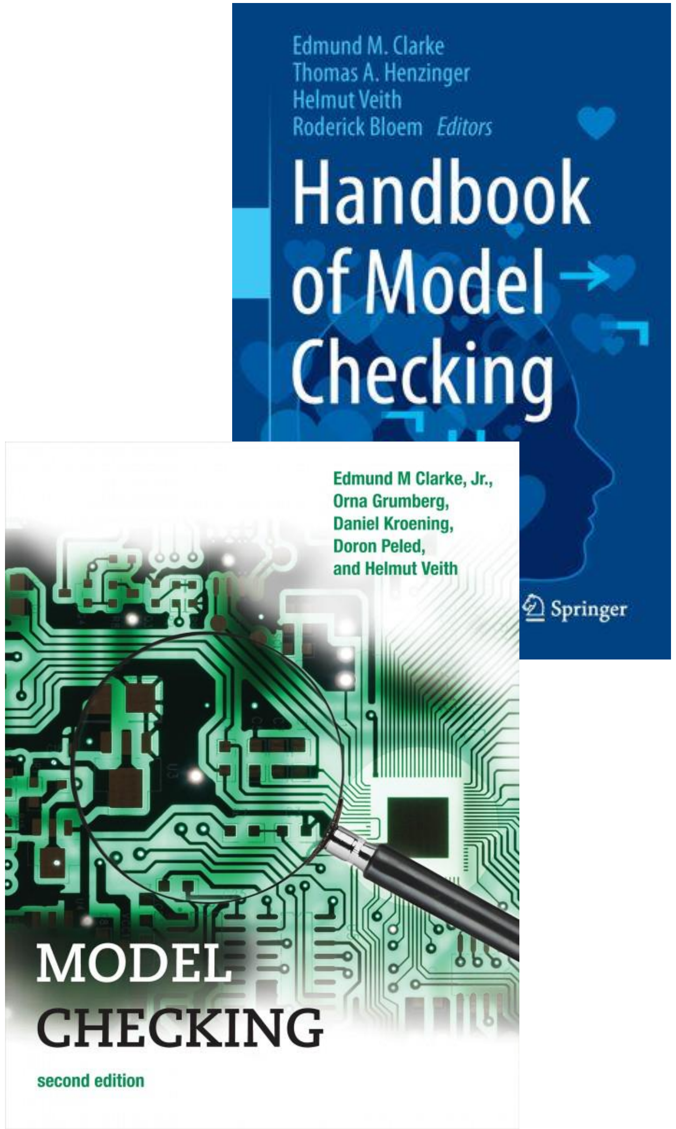
- translates Ada programs, together with properties encoded as assertion, to a set of Constrainted Horn Clauses (CHCs)
- employs off-the-shelf solvers for CHCs to verify the generated constraints and, consquently, to verify the original Ada program
- compares favourably with state of the art Ada verification technologies, albeit for a subset of Ada programs and for a class of properties that can be specified as assertions



Motivation

Model checking for Ada

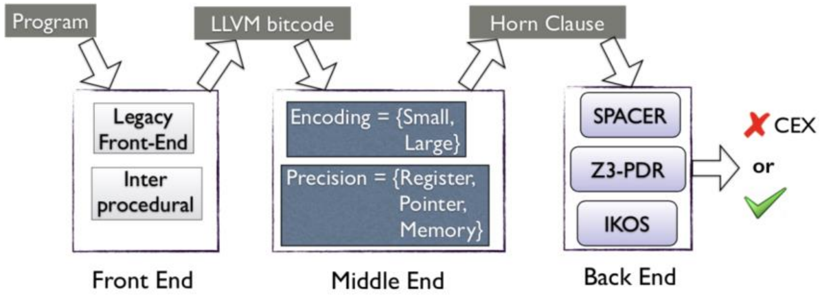
- **Model checking**
  - a verification technique, along with static analysis, deductive verification and theorem proving, in formal methods - notably one of the most successful
  - given a model of a system and a property to verify, model checking answers yes or no to the question “**does the model satisfy the property?**”
- **Ada model checking efforts**
- **2 approaches towards deveoping model checking tools**
  - model checking tools follow two main approaches:
    - generate input models for existing model checkers
    - design their own model checking algorithms, e.g., BLAST, SLAM
  - Building a model checkers is a labout intensive effort, as can be seen with established tools
    - tools like CBMC or Java Path-Finder have amassed countless person-months of engineering and testing.



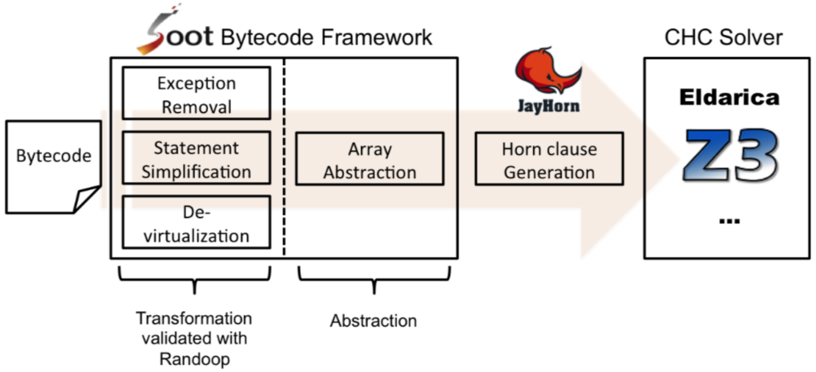
Motivation

Three-step model checking technologies

- over the recent years, this task has become a lot simpler with the increasing availability of off-the-shelf front-ends (such as LLVM for c programs) and verification back-ends (such as Z3 or Eldarica)
- with the increasing availability of such tools, the task of building a software model checker becomes just a matter of
  (1) picking a ***front-end*** and a ***back-end***, and
  (2) writing ***glue code*** to connect them
- Recent verification competitions have shown that this approach is feasible in practice.
  - The SeaHorn C verification framework, with LLVM front-end and off-the-shelf verification back-end, was able to outperform established tools in many categories.
  - The Java model checker JayHorn uses the Soot optimization framework as a front-end and a Horn contsraints solver as a backend.
- ***Motivated by these developments, we have implemented AdaHorn, a software model checking tool for Ada programs.***

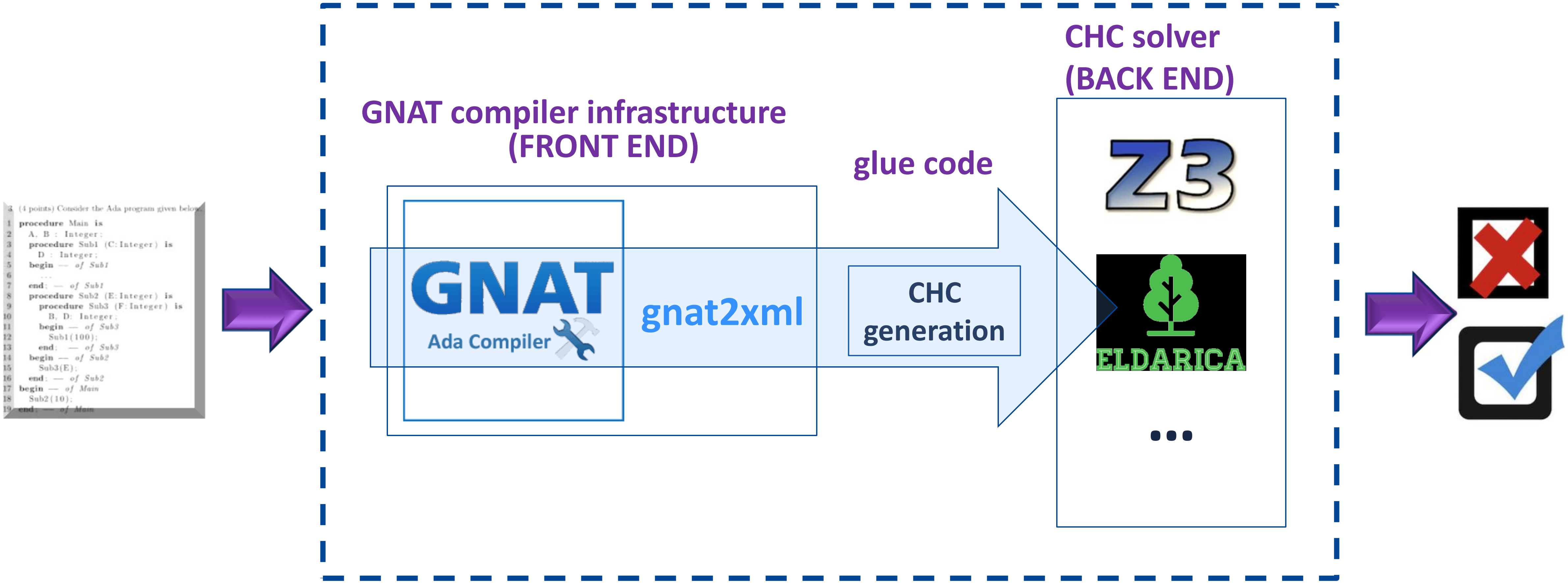


SeaHorn (Gurfinkel et.al. CAV15)



JayHorn (Kahsai et.al. CAV16)

Architecture of AdaHorn



Ada language subset

- current implementation does not support all language features and constructs of Ada
- basic constructs of Ada that can be used to write programs of medium complexity are supported.
  - (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.

Constraint Generation

Contrained Horn Clauses (CHCs)

- clause that has at most one positive occurrence of an uninterpreted predicate.
- fragment of first-order formulas modulo background theories, where its constraints are formulated using a given background theory.
- In this work, we assume the background theory be quantifier-free linear arithmetic.

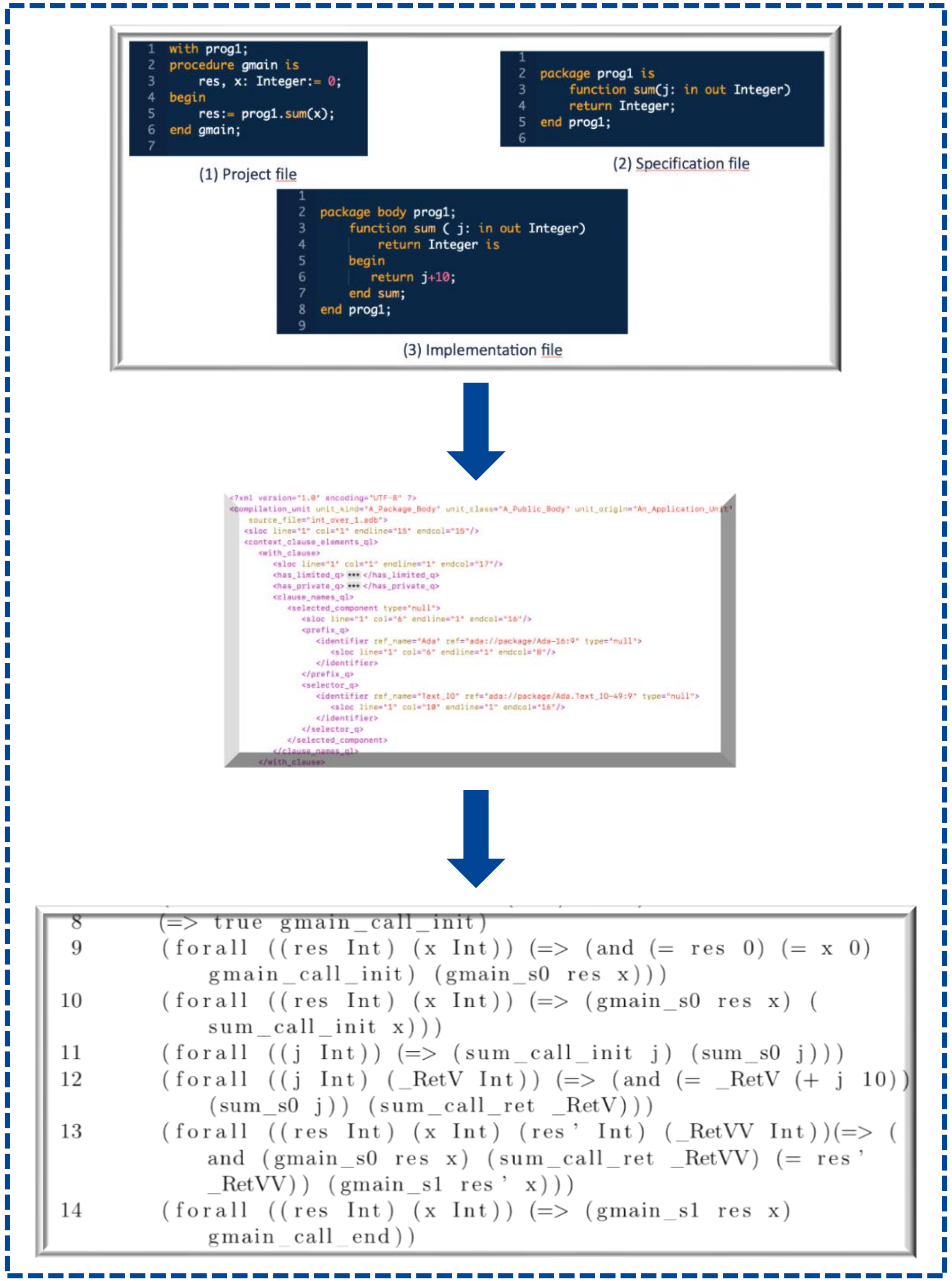
$$
\begin{aligned}
\Pi &::= HC \wedge \Pi \mid \top \\
HC &::= \forall vars : body \rightarrow head \\
pred &::= upred \mid \Phi \\
head &::= pred \\
body &::= \top \mid pred \mid body \wedge body \\
vars &::= the\ set\ of\ all\ variables\ in\ a\ given\ clause \\
upred &::= an\ uninterpreted\ predicate\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

Constraint Generation

- implemented by our **glue code**, the constraint generation procedure is key contribution of this work
  - employs **gnat2xml** utility from the GNAT compliler infrastructure to generate an **XML abstract syntax tree (AST)** for a given Ada program
  - performs a **top-down, recursive descent** through the XML syntax tree generating corresponding CHCs along the way
- two main considerations in this Ada to CHCs translation

  (1) encoding program states, which are valuations of program variables, at certain critical parts of the program such as loop entry and exit, procedure/function call and return, etc.,

  (2) encoding state transitions that occur during the execution of the program by translating involved Ada constructs into their corresponding CHC.



Constraint Generation

Example Ada program

```
with prog1;
procedure gmain is
    res, x: Integer:= 0;
begin
    res:= prog1.sum(x);
end gmain;

```

(1) Project file

```

package prog1 is
    function sum(j: in out Integer)
    return Integer;
end prog1;

```

(2) Specification file

```

package body prog1;
    function sum ( j: in out Integer)
        return Integer is
    begin
       return j+10;
    end sum;
end prog1;

```

(3) Implementation file

Constraint Generation

Generated CHCs

```
(declare-fun gmain_s0 (Int Int) Bool)
(declare-fun gmain_s1 (Int Int) Bool)
(declare-const gmain_call_init Bool)
(declare-const gmain_call_end Bool)

(declare-fun Prog1_Sum_s0 (Int) Bool)
(declare-fun Prog1_Sum_call_init (Int) Bool)
(declare-fun Prog1_Sum_call_ret (Int) Bool)

(assert (=> true gmain_call_init))
(assert (forall ((res Int) (x Int)) (=> (and (= res 0) (= x 0) gmain_call_init)
    (gmain_s0 res x)))))
(assert (forall ((res Int) (x Int)) (=> (gmain_s0 res x) (Prog1_Sum_call_init x)
    )))
(assert (forall ((j Int)) (=> (Prog1_Sum_call_init j) (Prog1_Sum_s0 j))))
(assert (forall ((j Int) (_RetV Int)) (=> (and (= _RetV (+ j 10)) (Prog1_Sum_s0
    j)) (Prog1_Sum_call_ret _RetV)))))
(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)))))
(assert (forall ((res Int) (x Int)) (=> (gmain_s1 res x) gmain_call_end)))
```

Constraint Generation

Generated CHCs

```
with prog1;
procedure gmain is
    res, x: Integer:= 0;
begin
    res:= prog1.sum(x);
end gmain;

```

(1) Project file

```

package prog1 is
    function sum(j: in out Integer)
    return Integer;
end prog1;

```

(2) Specification file

```

package body prog1;
    function sum ( j: in out Integer)
        return Integer is
    begin
        return j+10;
    end sum;
end prog1;

```

(3) Implementation file

```
(declare-fun gmain_s0 (Int Int) Bool)
(declare-fun gmain_s1 (Int Int) Bool)
(declare-const gmain_call_init Bool)
(declare-const gmain_call_end Bool)

(declare-fun Prog1_Sum_s0 (Int) Bool)
(declare-fun Prog1_Sum_call_init (Int) Bool)
(declare-fun Prog1_Sum_call_ret (Int) Bool)

(assert (=> true gmain_call_init))
(assert (forall ((res Int) (x Int)) (=> (and (= res 0) (= x 0) gmain_call_init)
    (gmain_s0 res x)))))
(assert (forall ((res Int) (x Int)) (=> (gmain_s0 res x) (Prog1_Sum_call_init x)
    )))
(assert (forall ((j Int)) (=> (Prog1_Sum_call_init j) (Prog1_Sum_s0 j)))))
(assert (forall ((j Int) (_RetV Int)) (=> (and (= _RetV (+ j 10)) (Prog1_Sum_s0
    j)) (Prog1_Sum_call_ret _RetV)))))
(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)))))
(assert (forall ((res Int) (x Int)) (=> (gmain_s1 res x) gmain_call_end)))
```

Constraint Generation

Generated CHCs

```
with prog1;
procedure gmain is
    res, x: Integer:= 0;
begin
    res:= prog1.sum(x);
end gmain;
```

(1) Project file

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

(2) Specification file

```
package body prog1;
    function sum ( j: in out Integer)
        return Integer is
    begin
        return j+10;
    end sum;
end prog1;
```

(3) Implementation file

```
(declare-fun gmain_s0 (Int Int) Bool)
(declare-fun gmain_s1 (Int Int) Bool)
(declare-const gmain_call_init Bool)
(declare-const gmain_call_end Bool)

(declare-fun Prog1_Sum_s0 (Int) Bool)
(declare-fun Prog1_Sum_call_init (Int) Bool)
(declare-fun Prog1_Sum_call_ret (Int) Bool)

(assert (=> true gmain_call_init))
(assert (forall ((res Int) (x Int)) (=> (and (= res 0) (= x 0) gmain_call_init)
    (gmain_s0 res x))))
(assert (forall ((res Int) (x Int)) (=> (gmain_s0 res x) (Prog1_Sum_call_init x)
    )))
(assert (forall ((j Int)) (=> (Prog1_Sum_call_init j) (Prog1_Sum_s0 j))))
(assert (forall ((j Int) (_RetV Int)) (=> (and (= _RetV (+ j 10)) (Prog1_Sum_s0
    j)) (Prog1_Sum_call_ret _RetV))))
(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))))
(assert (forall ((res Int) (x Int)) (=> (gmain_s1 res x) gmain_call_end)))
```

Constraint Generation

Generated CHCs

```ada
with prog1;
procedure gmain is
    res, x: Integer:= 0;
begin
    res:= prog1.sum(x);
end gmain;
```

(1) Project file

```ada
package prog1 is
    function sum(j: in out Integer)
    return Integer;
end prog1;
```

(2) Specification file

```ada
package body prog1;
    function sum ( j: in out Integer)
        return Integer is
    begin
        return j+10;
    end sum;
end prog1;
```

(3) Implementation file

```
(declare-fun gmain_s0 (Int Int) Bool)
(declare-fun gmain_s1 (Int Int) Bool)
(declare-const gmain_call_init Bool)
(declare-const gmain_call_end Bool)

(declare-fun Prog1_Sum_s0 (Int) Bool)
(declare-fun Prog1_Sum_call_init (Int) Bool)
(declare-fun Prog1_Sum_call_ret (Int) Bool)

(assert (=> true gmain_call_init))
(assert (forall ((res Int) (x Int)) (=> (and (= res 0) (= x 0) gmain_call_init)
    (gmain_s0 res x))))
(assert (forall ((res Int) (x Int)) (=> (gmain_s0 res x) (Prog1_Sum_call_init x)
    )))
(assert (forall ((j Int)) (=> (Prog1_Sum_call_init j) (Prog1_Sum_s0 j))))
(assert (forall ((j Int) (_RetV Int)) (=> (and (= _RetV (+ j 10)) (Prog1_Sum_s0
    j)) (Prog1_Sum_call_ret _RetV))))
(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))))
(assert (forall ((res Int) (x Int)) (=> (gmain_s1 res x) gmain_call_end)))
```

Constraint Generation

Generated CHCs

```
with prog1;
procedure gmain is
    res, x: Integer:= 0;
begin
    res:= prog1.sum(x);
end gmain;
```

(1) Project file

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

(2) Specification file

```
package body prog1;
    function sum ( j: in out Integer)
        return Integer is
    begin
        return j+10;
    end sum;
end prog1;
```

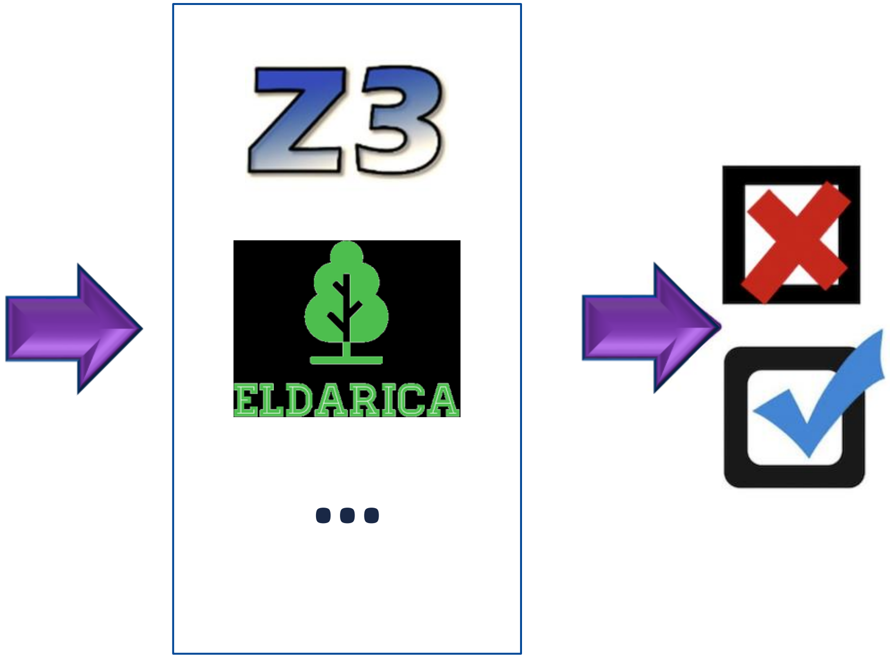
(3) Implementation file

```
(declare-fun gmain_s0 (Int Int) Bool)
(declare-fun gmain_s1 (Int Int) Bool)
(declare-const gmain_call_init Bool)
(declare-const gmain_call_end Bool)

(declare-fun Prog1_Sum_s0 (Int) Bool)
(declare-fun Prog1_Sum_call_init (Int) Bool)
(declare-fun Prog1_Sum_call_ret (Int) Bool)

(assert (=> true gmain_call_init))
(assert (forall ((res Int) (x Int)) (=> (and (= res 0) (= x 0) gmain_call_init)
    (gmain_s0 res x))))
(assert (forall ((res Int) (x Int)) (=> (gmain_s0 res x) (Prog1_Sum_call_init x)
    )))
(assert (forall ((j Int)) (=> (Prog1_Sum_call_init j) (Prog1_Sum_s0 j))))
(assert (forall ((j Int) (_RetV Int)) (=> (and (= _RetV (+ j 10)) (Prog1_Sum_s0
    j)) (Prog1_Sum_call_ret _RetV))))
(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))))
(assert (forall ((res Int) (x Int)) (=> (gmain_s1 res x) gmain_call_end)))
```

Constraint Solving

- Generated constriant is solved using an off-the-shelf solver for horn constrains

```
(=> true gmain_call_init)
(forall ((res Int) (x Int)) (=> (and (= res 0) (= x 0))
    gmain_call_init) (gmain_s0 res x)))
(forall ((res Int) (x Int)) (=> (gmain_s0 res x) (
    sum_call_init x)))
(forall ((j Int)) (=> (sum_call_init j) (sum_s0 j)))
(forall ((j Int) (_RetV Int)) (=> (and (= _RetV (+ j 10))
    (sum_s0 j)) (sum_call_ret _RetV)))
(forall ((res Int) (x Int) (res' Int) (_RetVV Int))(=> (
    and (gmain_s0 res x) (sum_call_ret _RetVV) (= res'
    _RetVV)) (gmain_s1 res' x)))
(forall ((res Int) (x Int)) (=> (gmain_s1 res x)
    gmain_call_end))
```



Evaluation

Experimental setup

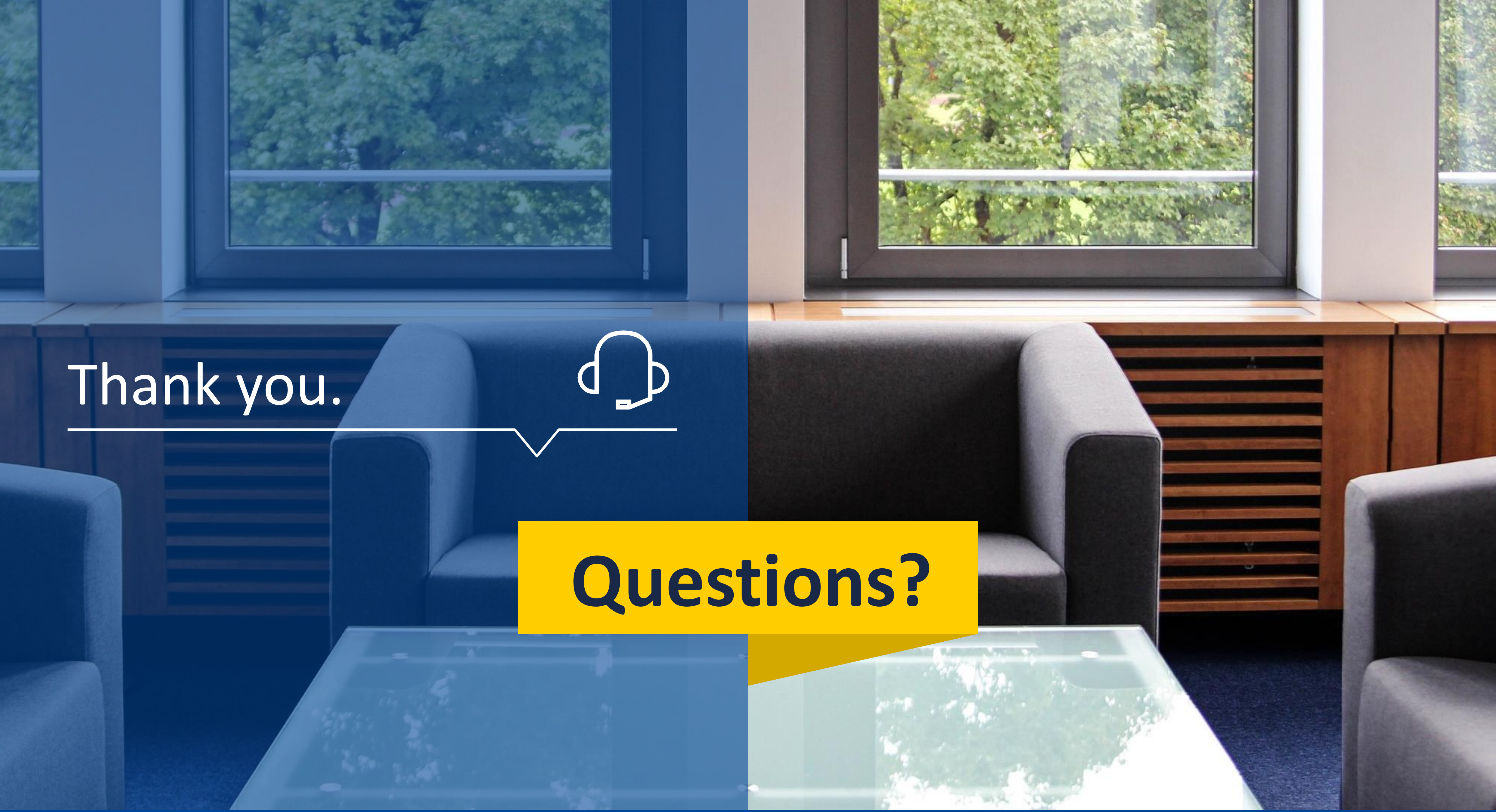
- compare with GNATProve
- propose and uses **Ada benchmarks** inspired by C programs from the competition **SV-COMP 2017**
  - at most 60 lines of code each
  - classifies into four categories: ***Arrays***, ***Floats***, ***Loops***, and ***RT-Properties***
  - selected benchmarks expressible in our subset of Ada
- Given an Ada program with an assertion, the verification tools are tasked with
  (1) proving the assertion occurring is valid, in which case the tools should return SAT, or
  (2) demonstrating the assertion is not valid, i.e., it is possible to violate it, in which case the tools should return UNSAT.
- Comparing the expected result and the actual result, the results of our experiments are classified into the following four categories: TP, TN, FP, and FN.
  - in addition, **Unknown** and **Timout** results are also possible

Evaluation
Experimental result

Benchmarks	Problems	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
RT-Properties	8	0	5	2	0	0	2	6	0	0	0	0

- GNATProve verifies each benchmark within 3 seconds
  - **correct results** only for 17 cases
  - outputs **48 false positives** and **2 false negatives**!
- following obervations can explain these outputs
  - GNATProve performs intermediate checks before checking assertions, and GNATProve assumes previous checks have been successful.
  - succesive checks are not analyzed by GNATProve after previous check has failed (see Paragraph 7.3.4 in [22]).
  - the analysis can be improved by adding manual annotations to the program (this is call “direct justification” [22]).

- AdaHorn verifies each benchmark within 60 seconds
  - **correct results** for 59 cases and **no false** negative
  - 4 **false positives**, 4 **unknown** and 1 **timeout** (use floating-point data types)
- AdaHorn over-approximates floats with reals for performance purposes of the used solvers, and it looks to lead to numerical precision differences with the Ada compiler.



Thank you.

Questions?

Dr. Tewodros A. Beyene

Contact

fortiss GmbH
An-Institut Technische Universität München
Guerickestraße 25 · 80805 München · Germany

tel +49 89 3603522 24   fax +49 89 3603522 50

beyene@fortiss.org
www.fortiss.org