Comparison between Boogie2 and Why3 for the verification of *MiniMaple* programs

Muhammad Taimoor Khan

Formal Methods Seminar

January 25, 2012

FWF
Der Wissenschaftsfonds.

Doctoral Program
Computational Mathematics
Numerical Analysis and Symbolic Computation

Outline

1. Introduction
2. Boogie2
3. Why3
4. My Work
5. Conclusions

Introduction

Let’s say, we have

- *n* programming languages and
- *m* theorem provers

Introduction

Let's say, we have

- n programming languages and
- m theorem provers

For program verification, we need

- $n \times m$ translations to generate verification conditions

Introduction

Let's say, we have

- n programming languages and
- m theorem provers

For program verification, we need

- $n \times m$ translations to generate verification conditions

Better solution is to translate n programs

- into a common intermediate (verification) language
  - common to m provers
- requires $n + m$ translations

Introduction

Let's say, we have

- n programming languages and
- m theorem provers

For program verification, we need

- $n \times m$ translations to generate verification conditions

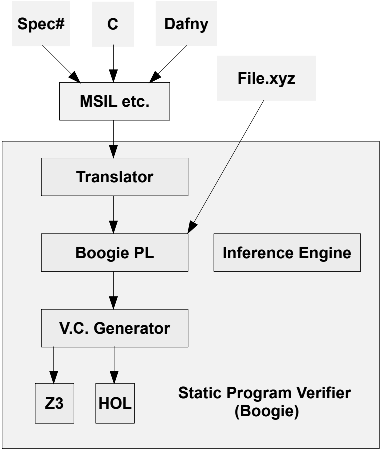
Better solution is to translate n programs

- into a common intermediate (verification) language
    - common to m provers
- requires $n + m$ translations
- benefits
    - automatically generate verification conditions
    - these conditions can be proved by the prover of one's choice

Intermediate Verification Languages

- Boogie (by Microsoft 2006)
    - SPEC#
    - VCC
    - Dafny
- Why (by LRI, France 2003)
    - Krakatoa
    - Caduceus
    - Frama-C

Boogie2



Spec#

C

Dafny

File.xyz

MSIL etc.

Translator

Boogie PL

Inference Engine

V.C. Generator

Z3

HOL

Static Program Verifier
(Boogie)

Elements of the Boogie Language

- Mathematical components specify
  - types
  - constants
  - functions
  - axioms
- Imperative components specify
  - global variables
  - procedure declarations
  - procedure implementations
  - also described respective constrained states by mathematical components
  - sets of execution traces
    - e.g. in case of procedure, caller and callee traces
- Expressions
  - typical logical, boolean, arithmetic etc.

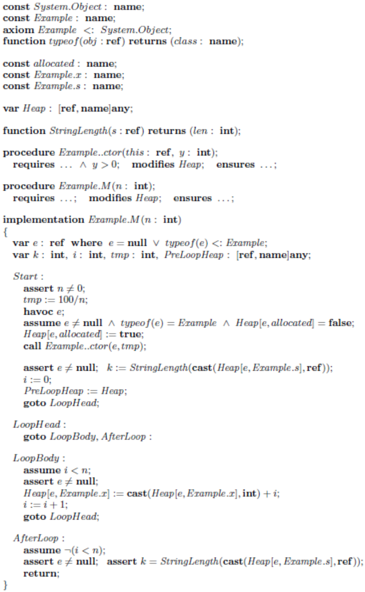
Features of the Boogie Language

- Parametric polymorphism
  - function can take polymorphic type parameters
- Partial ordering ($<:$)
  - for constants of the same type
- Nondeterminism (havoc statements)
  - can assign arbitrary values to a set of variables
- Flow-chart like language
  - non-imperative and only supports goto statements

An Example Spec# Programs

```
public class Example {
  int x;
  string! s;
  invariant s.Length >= 12;
  public Example(int y) requires y > 0; { ... }
  public static void M(int n) {
    Example e = new Example(100/n);
    int k = e.s.Length;
    for (int i = 0; i < n; i++) { e.x += i; }
    assert k == e.s.Length;
  }
}
```

An Example Spec# to Boogie Translation

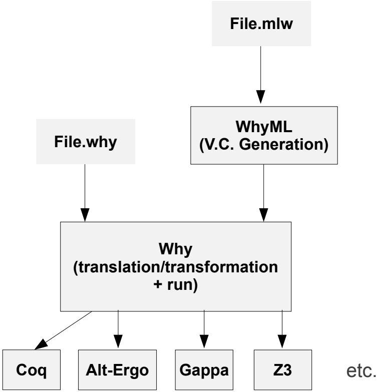


```
const System.Object : name;
const Example : name;
axiom Example <: System.Object;
function typeof(obj : ref) returns (class : name);

const allocated : name;
const Example.x : name;
const Example.s : name;

var Heap : [ref, name]any;

function StringLength(s : ref) returns (len : int);

procedure Example..ctor(this : ref, y : int);
  requires ... ∧ y > 0;   modifies Heap;   ensures ...;

procedure Example.M(n : int);
  requires ...;   modifies Heap;   ensures ...;

implementation Example.M(n : int)
{
  var e : ref where e = null ∨ typeof(e) <: Example;
  var k : int, i : int, tmp : int, PreLoopHeap : [ref, name]any;

  Start :
    assert n ≠ 0;
    tmp := 100/n;
    havoc e;
    assume e ≠ null ∧ typeof(e) = Example ∧ Heap[e, allocated] = false;
    Heap[e, allocated] := true;
    call Example..ctor(e, tmp);

    assert e ≠ null;   k := StringLength(cast(Heap[e, Example.s], ref));
    i := 0;
    PreLoopHeap := Heap;
    goto LoopHead;

  LoopHead :
    goto LoopBody, AfterLoop :

  LoopBody :
    assume i < n;
    assert e ≠ null;
    Heap[e, Example.x] := cast(Heap[e, Example.x], int) + i;
    i := i + 1;
    goto LoopHead;

  AfterLoop :
    assume ¬(i < n);
    assert e ≠ null;   assert k = StringLength(cast(Heap[e, Example.s], ref));
    return;
}
```

Strengths and Weaknesses

- Strengths
  - many front-end tools support Boogie
    - Spec# compiler
  - imperative style syntax
- Weaknesses
  - no rich theory language
  - full verification is hard
    - only have very good supports of Z3
  - no sufficiently documented semantics definition

Why3


File.mlw
WhyML
(V.C. Generation)
File.why
Why
(translation/transformation
+ run)
Coq
Alt-Ergo
Gappa
Z3
etc.

Simplicity and Collaborative Proofs

- Generates simple verification conditions
  - no memory store
  - conditions about the contents of the data structures
- Still captures sufficient details
  - termination and array bound checking etc.
- Provides collaborative proofs
  - to handle unproved verification conditions with interactive provers
  - but provides as much proof automation as possible
- Also WP-based semantics

Some more features

Influenced by ML

- Why3 supports
  - algebraic data types
  - pattern matching
- WhyML supports
  - type inference
  - currying
  - abstract data types

Theories and Modules

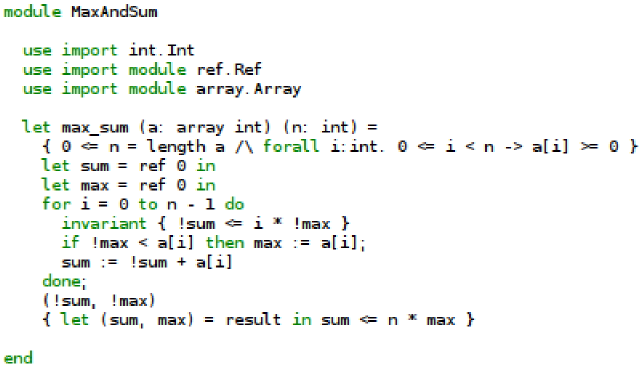
- Built-in theories
  - e.g., List, Int etc.
- Built-in modules
  - e.g., Ref etc.
- Can be used directly or by cloning

```
(* Theory Definition *)
theory Orty
        use import list.List

        type orty
        ...
end

(* Module Definition *)
module MyModule

        use import int.Int
        use import module ref.Ref
        use import Orty

        ...
end
```

Abstract and Algebraic Data Types

```
(* Abstract Data Type *)
theory Orty

        type orty
        ...
end

(* Algebraic Data Type *)
theory List

        type list 'a = Nil | Cons 'a (list 'a)

end
```

An Example Why3 Program - MaxAndSum



```
module MaxAndSum

  use import int.Int
  use import module ref.Ref
  use import module array.Array

  let max_sum (a: array int) (n: int) =
    { 0 <= n = length a /\ forall i:int. 0 <= i < n -> a[i] >= 0 }
    let sum = ref 0 in
    let max = ref 0 in
    for i = 0 to n - 1 do
      invariant { !sum <= i * !max }
      if !max < a[i] then max := a[i];
      sum := !sum + a[i]
    done;
    (!sum, !max)
    { let (sum, max) = result in sum <= n * max }

end
```

Strengths and Weaknesses

- Strengths
    - rich logic, readily usable in programs
    - support collaborative proofs by many beck-end provers
    - modularity and abstract data types
    - close to specification-based programming
- Weaknesses
    - program and specification are tied together
        - even w.r.t. syntax
    - some data structures cannot be defined (but signatures)
        - e.g. mutable trees etc.

My Work

- Formal specification respectively verification of programs written in (the most widely used) untyped computer algebra languages
  - Mathematica and Maple
- Develop a tool to find errors by static analysis
  - for example type inconsistencies
  - and violations of methods preconditions
- Also
  - to realize the gap between the example computer algebra algorithm and its implementation
  - to formalize properties of computer algebra
- Demonstration example
  - Maple package *DifferenceDifferential* developed by Christian Dönch
- *MiniMaple*
  - A simple but substantial subset of Maple
  - Covers all syntactic domains of Maple but fewer expressions

A *MiniMaple* Example Program

```
sumproc := proc(l: Or(integer, list(integer)))::integer;
           local sum::integer:=0, el::list(integer), x::integer;
           if type(l,integer) then
                 if l <> 0 then
                           sum := sum + l;
                 else
                      return sum;
                 end if;
           elif type(l,list(integer)) then
                   for x from 1 by 1 to nops(l) do
                        el := l[x];
                        if el <> 0 then
                             sum:=sum+el;
                        else
                             return sum;
                        end if;
                   end do;
           end if;
           return sum;
end proc;
```

Special features of the *MiniMaple* Type System

- Uses only *Maple* type annotations
  - *Maple* uses them for *dynamic type checking*
  - *MiniMaple* uses them for *static type checking*
- Context (global vs local)
  - *global*
    - may introduce new identifiers by assignments
    - types of identifiers may change arbitrarily by assignments
  - *local*
    - identifiers only introduced by declarations
    - types of identifiers can only be *specialized*
- Type tests in Maple, i.e. **type**(I, T)
  - branches may have different type information for the same variable
    - track type information to allow satisfiable tests only
  - number of loop iterations might influence the type information
    - least fix point as an upper bound on the types of the variable
    - as a special case the declared type is the least fixed point

Elements of the Specification Language

- Mathematical theories
  - Types
    - User defined data-types
    - Abstract data types
  - Functions and predicates (declared/defined)
  - Axioms
- Procedure specifications
  - Pre-post conditions
  - Exceptions
  - Global variables
- Loop specifications
  - Invariants
  - Termination terms
- Assertions
  - To constrain the state of execution

Challenges of Specification Language for *MiniMaple*

- Support of some non-standard types of objects
  - e.g. symbols, unevaluated expressions etc.
- Additional functions and predicates
  - e.g. type test, **type**(I, T)
- Specification of abstract mathematical concepts by an abstract data type
  - Weaker support in current classical specification languages
  - e.g., ring, variables and ordering of a polynomial
  - ADDO as an abstract data type represented by list of tuples
    - Abstract Difference Differential Operator

An example utility procedure of *DifferenceDifferential*

```
(*@
    'type/ADDO';
    define(terms, terms(ad::ADDO)=...);
    define(getTerm, getTerm(ad::ADDO,i::nat, j::nat)=...);
    isADDO(d);
    isADDOTerm(c,n,z,e);
    ...
    assume(isADDO(d) equivalent forall(i::integer, 1<=i and i<=terms(d) implies
             isADDOTerm(getTerm(d,i,1), getTerm(d,i,2), getTerm(d,i,3), getTerm(d,i,4]));
    assume(isADDOTerm(c,n,z,e) equivalent inField(c) and isGenerator(e));
   ...
    define(power, power(a::integer,0)=1, power(a::integer,b::integer)= mul(a,1...b));
    define(maps, maps(d::DDO)=...);
   @*)
global noauto, generators, ...;
   ...
(*@
    requires 1 <= z and z <= power(2,length(noauto)) and
              forall(i::integer, 1<=i and i<=terms(maps(a)) implies isGenerator(getTerm(maps(a),i,4))) and
              forall(i::integer, 1<=i and i<=terms(maps(b)) implies isGenerator(getTerm(maps(b),i,4)));
    global EMPTY;
    ensures
       ( forall(j::integer, 1<=j and j<=nops(RESULT) implies isGenerator(RESULT[j][1],maps(a),maps(b)) and
                      RESULT[j][2] = isLT(maps(a),z) and RESULT[j][3] = isLT(maps(b),z)) )
       or
       (nops(RESULT) = 0 and ...);
    @*)

VGB := proc (z::integer, a::DDO, b::DDO)::list([symbol,list(symbol),list(symbol)]) ... return v; end proc;
```

Why *Why3*?

Need to verify the implementation of some computer algebra algorithm along-with reasonable proof/details about the algorithm itself

Why *Why3*?

Need to verify the implementation of some computer algebra algorithm along-with reasonable proof/details about the algorithm itself

- *MiniMaple* and its specification language
    - symbolic programs are close to algorithms

Why *Why3*?

Need to verify the implementation of some computer algebra algorithm along-with reasonable proof/details about the algorithm itself

- *MiniMaple* and its specification language
  - symbolic programs are close to algorithms
- Arguments in favor of Why3
  - rich theory language
  - algebraic and abstract data types
  - inductive predicates
  - both automated and interactive proof

My Current Work

My Current Work

Developing verification calculus for *MiniMaple* programs

- to generate verification conditions
- also to prove verification conditions

My Current Work

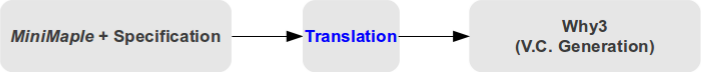
Developing verification calculus for *MiniMaple* programs

- to generate verification conditions
- also to prove verification conditions

My Current Work

Developing verification calculus for *MiniMaple* programs

- to generate verification conditions
- also to prove verification conditions

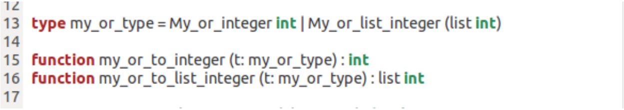


MiniMaple + Specification → Translation → Why3 (V.C. Generation)

Translation to corresponding semantically equivalent Why3 constructs

An Example Translation (*MiniMaple* to Why3)

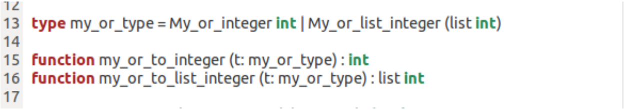
- Union-type, i.e. Or(integer, list(integer))



```
type my_or_type = My_or_integer int | My_or_list_integer (list int)

function my_or_to_integer (t: my_or_type) : int
function my_or_to_list_integer (t: my_or_type) : list int

```

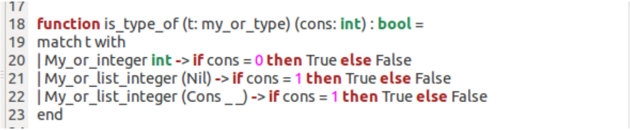
An Example Translation (*MiniMaple* to Why3)

- Union-type, i.e. Or(integer, list(integer))



```
type my_or_type = My_or_integer int | My_or_list_integer (list int)

function my_or_to_integer (t: my_or_type) : int
function my_or_to_list_integer (t: my_or_type) : list int
```

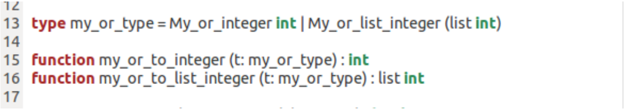
- Type-tests, i.e. type(l, integer) and type(l, list(integer))



```
function is_type_of (t: my_or_type) (cons: int) : bool =
match t with
| My_or_integer int -> if cons = 0 then True else False
| My_or_list_integer (Nil) -> if cons = 1 then True else False
| My_or_list_integer (Cons _ _) -> if cons = 1 then True else False
end
```

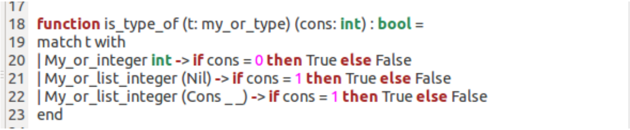
An Example Translation (*MiniMaple* to Why3)

- Union-type, i.e. Or(integer, list(integer))



```
type my_or_type = My_or_integer int | My_or_list_integer (list int)

function my_or_to_integer (t: my_or_type) : int
function my_or_to_list_integer (t: my_or_type) : list int

```

- Type-tests, i.e. type(l, integer) and type(l, list(integer))



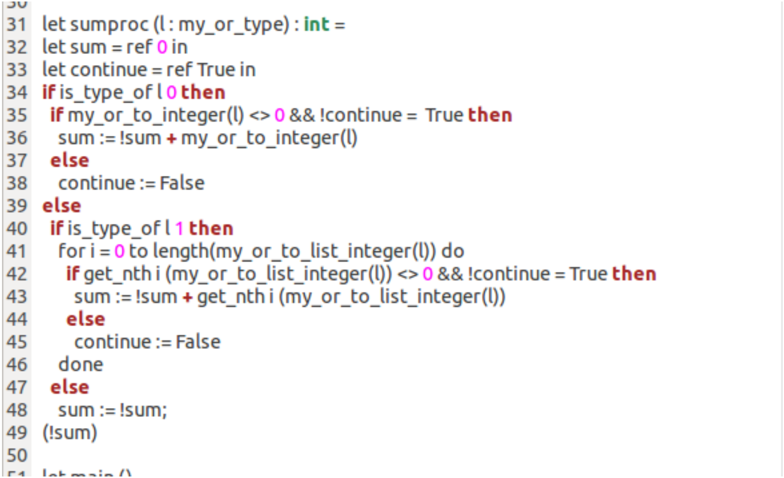
```
function is_type_of (t: my_or_type) (cons: int) : bool =
match t with
| My_or_integer int -> if cons = 0 then True else False
| My_or_list_integer (Nil) -> if cons = 1 then True else False
| My_or_list_integer (Cons _ _) -> if cons = 1 then True else False
end
```

- Utility function to extract *nth* element of a list

```
let get_nth (i: int) (l: list int) =
 match nth i l with
 | None -> absurd
 | Some x -> x
 end
```

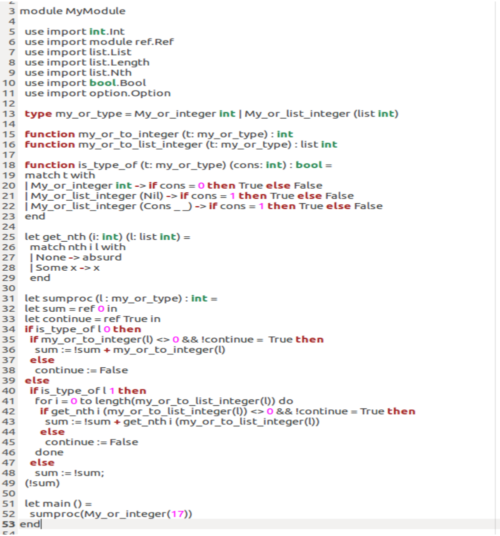
MiniMaple to Why3 - contd.

- Procedure sumproc(l: Or(integer, list(integer)))::integer



```
let sumproc (l : my_or_type) : int =
let sum = ref 0 in
let continue = ref True in
if is_type_of l 0 then
 if my_or_to_integer(l) <> 0 && !continue =  True then
  sum := !sum + my_or_to_integer(l)
 else
  continue := False
else
 if is_type_of l 1 then
  for i = 0 to length(my_or_to_list_integer(l)) do
   if get_nth i (my_or_to_list_integer(l)) <> 0 && !continue = True then
    sum := !sum + get_nth i (my_or_to_list_integer(l))
   else
    continue := False
  done
 else
  sum := !sum;
(!sum)

let main ()
```

Complete Example Translation



```
module MyModule

 use import int.Int
 use import module ref.Ref
 use import list.List
 use import list.Length
 use import list.Nth
 use import bool.Bool
 use import option.Option

 type my_or_type = My_or_integer int | My_or_list_integer (list int)

 function my_or_to_integer (t: my_or_type) : int
 function my_or_to_list_integer (t: my_or_type) : list int

 function is_type_of (t: my_or_type) (cons: int) : bool =
 match t with
 | My_or_integer int -> if cons = 0 then True else False
 | My_or_list_integer (Nil) -> if cons = 1 then True else False
 | My_or_list_integer (Cons _ _) -> if cons = 1 then True else False
 end

 let get_nth (i: int) (l: list int) =
  match nth i l with
  | None -> absurd
  | Some x -> x
  end

 let sumproc (l : my_or_type) : int =
 let sum = ref 0 in
 let continue = ref True in
 if is_type_of l 0 then
  if my_or_to_integer(l) <> 0 && !continue =  True then
   sum := !sum + my_or_to_integer(l)
  else
   continue := False
 else
  if is_type_of l 1 then
   for i = 0 to length(my_or_to_list_integer(l)) do
    if get_nth i (my_or_to_list_integer(l)) <> 0 && !continue = True then
     sum := !sum + get_nth i (my_or_to_list_integer(l))
    else
     continue := False
   done
  else
   sum := !sum;
 (!sum)

 let main () =
  sumproc(My_or_integer(17))
end
```

Experiments and Readings (so far)

- *MiniMaple* (reasonably supported)

Experiments and Readings (so far)

- *MiniMaple* (reasonably supported)
    - Types
        - integer, boolean, string, float etc. (supported)
        - list(T), {T}, [Tseq] (can be specified by the built-in list library)
        - uneval, symbol and union etc. (can also be axiomatized easily)

Experiments and Readings (so far)

- *MiniMaple* (reasonably supported)
  - Types
    - integer, boolean, string, float etc. (supported)
    - list(T), {T}, [Tseq] (can be specified by the built-in list library)
    - uneval, symbol and union etc. (can also be axiomatized easily)
  - Expressions (also can be specified easily)
    - typical arithmetic and logical expressions
    - unevaluated
    - sequence

Experiments and Readings (so far)

- *MiniMaple* (reasonably supported)
  - Types
    - integer, boolean, string, float etc. (supported)
    - list(T), {T}, [Tseq] (can be specified by the built-in list library)
    - uneval, symbol and union etc. (can also be axiomatized easily)
  - Expressions (also can be specified easily)
    - typical arithmetic and logical expressions
    - unevaluated
    - sequence
  - Special constructs (can be specified by pattern matching)
    - type-tests
    - sub-typing relations

Experiments and Readings (so far)

- *MiniMaple* (reasonably supported)
  - Types
    - integer, boolean, string, float etc. (supported)
    - list(T), {T}, [Tseq] (can be specified by the built-in list library)
    - uneval, symbol and union etc. (can also be axiomatized easily)
  - Expressions (also can be specified easily)
    - typical arithmetic and logical expressions
    - unevaluated
    - sequence
  - Special constructs (can be specified by pattern matching)
    - type-tests
    - sub-typing relations
  - Other constructs (supported by the corresponding constructs)
    - procedures, modules
    - for-loop variations
    - exception handling

Experiments and Readings (so far) contd.

- Specification language (almost directly supported)

Experiments and Readings (so far) contd.

- Specification language (almost directly supported)
    - Mathematical theories (supported by the corresponding constructs)
        - user-defined and abstract data types
        - functions and predicates
        - axioms

Experiments and Readings (so far) contd.

- Specification language (almost directly supported)
  - Mathematical theories (supported by the corresponding constructs)
    - user-defined and abstract data types
    - functions and predicates
    - axioms
  - Procedure specifications (partially supported)
    - pre-post conditions
    - exceptions
    - global variables

Experiments and Readings (so far) contd.

- Specification language (almost directly supported)
  - Mathematical theories (supported by the corresponding constructs)
    - user-defined and abstract data types
    - functions and predicates
    - axioms
  - Procedure specifications (partially supported)
    - pre-post conditions
    - exceptions
    - global variables
  - Loop specifications (supported by invariants + variants)
    - invariants
    - termination term

Experiments and Readings (so far) contd.

- Specification language (almost directly supported)
  - Mathematical theories (supported by the corresponding constructs)
    - user-defined and abstract data types
    - functions and predicates
    - axioms
  - Procedure specifications (partially supported)
    - pre-post conditions
    - exceptions
    - global variables
  - Loop specifications (supported by invariants + variants)
    - invariants
    - termination term
  - Assertions (supported)

Experiments and Readings (so far) contd.

- Specification language (almost directly supported)
  - Mathematical theories (supported by the corresponding constructs)
    - user-defined and abstract data types
    - functions and predicates
    - axioms
  - Procedure specifications (partially supported)
    - pre-post conditions
    - exceptions
    - global variables
  - Loop specifications (supported by invariants + variants)
    - invariants
    - termination term
  - Assertions (supported)
  - Other constructs (supported)
    - typed logical quantifiers