On the Formal Verification of Maple Programs*

Muhammad Taimoor Khan
Doktoratskolleg Computational Mathematics
and
Research Institute for Symbolic Computation
Johannes Kepler University
Linz, Austria
Muhammad.Taimoor.Khan@risc.jku.at

June 28, 2013

Abstract

In this paper, we present an example-based demonstration of our recent results on the formal verification of programs written in the computer algebra language *(Mini)Maple* (a slightly modified subset of Maple). The main goal of this work is to develop a verification framework for behavioral analysis of *MiniMaple* programs. For verification, we translate an annotated *MiniMaple* program into the language Why3ML of the intermediate verification tool Why3 developed at LRI, France. We generate verification conditions by the corresponding component of Why3 and later prove the correctness of these conditions by various supported by the Why3 back-end automatic and interactive theorem provers. We have used the verification framework to verify some parts of the main test example of our verification framework, the Maple package *DifferenceDifferential* developed at our institute to compute bivariate difference-differential polynomials using relative Gröbner bases.

*The research was funded by the Austrian Science Fund (FWF): W1214-N15, project DK10.

Contents

1 Introduction

We report on the progress of a project whose goal is to develop a verification framework for the programs that are written in the language of the computer algebra system Maple. Here, we are mainly interested to find behavioral errors in Maple programs such as detection of type inconsistencies and violations of methods preconditions: for this purpose, these programs need to be annotated with the types of variables and methods contracts.

As presented earlier in [8] and [9], we started by defining *MiniMaple* as a substantial subset of the computer algebra language Maple. We have formalized a type system for *MiniMaple* and implemented a corresponding type checker. The type checker has been applied to the main test example for our verification framework, the Maple package *DifferenceDifferential* [3] developed at our institute for the computation of bivariate difference-differential dimension polynomials. Also we have defined a specification language to formally specify the behavior of *MiniMaple* programs. Moreover, the formal semantics of *MiniMaple* and its specification language has been defined as a pre-requisite for further formal treatment.

In this paper, we report on new results concerning a verification framework for *MiniMaple*. This framework employs Why3 [2] as an intermediate verification tool for the generation of verification conditions and proving their correctness by various automatic decision procedures and interactive theorem provers, e.g. Z3 and Coq. For this purpose, we have defined the translation of annotated *MiniMaple* into a semantically equivalent Why3ML program; the proof of soundness of this translation is still an open goal. More details on project and all software are free available from `http://www.risc.jku.at/people/mtkhan/dk10/`.

The rest of the paper is organized as follows: in Section 2, we briefly discuss the formal specification of *MiniMaple* by an example. In Section 3, we demonstrate our framework for the formal verification of *MiniMaple* by an example. Section 4 presents conclusions and future work.

2 Formal Specification

Since type safety is a pre-requisite of program correctness, we have formalized a type system for *MiniMaple*. Furthermore, to formally specify the behavior of *MiniMaple* programs, we have defined a specification language for *MiniMaple*. Listing 1 gives an example of a *MiniMaple* program which is type checked and formally specified. We will use the same example in the following section for the discussion of formal verification.

This example procedure takes a list of integers and floats and computes the sum of the integers and the sum of the floats; it returns as a result a tuple of both sums. The procedure may also terminate prematurely for certain inputs: if an integer value 0 or a float value less than 0.5 is encountered in the list, the procedure returns the sums computed so far.

The formal specification of the procedure states that the procedure has no pre-condition (**requires** clause); a global variable *status* can be modified by the body of the procedure (**global** clause). The behavior (post condition) of the procedure is specified in the **ensures** clause. The loop of this example procedure is specified by the user defined invariant (**invariant** clause) and the termination

term (**decreases** clause).

```
1. status:=0;
(*@
requires true;
global status;
ensures
 ( status = -1 and RESULT[1] = add(e, e in l, type(e,integer))
 and RESULT[2] = add(e, e in l, type(e,float))
 and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],integer) implies l[i]<>0)
 and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],float) implies l[i]>=0.5) )
 or
 ( 1<=status and status<=nops(l)
 and RESULT[1] = add(l[i], i=1..status-1, type(l[i],integer))
 and RESULT[2] = add(l[i], i=1..status-1, type(l[i],float))
 and ( (type(l[status],integer) and l[status]=0) or
        (type(l[status],float) and l[status]<0.5) )
 and forall(i::integer, 1<=i and i<status and type(l[i],integer) implies l[i]<>0)
 and forall(i::integer, 1<=i and i<status and type(l[i],float) implies l[i]>=0.5) );
@*)
2. sum := proc(l::list(Or(integer,float)))::[integer,float];
3.          global status;
4.          local i, x::Or(integer,float), si::integer:=0, sf::float:=0.0;
5.          for i from 1 by 1 to nops(l) do
            (*@ invariant ...; decreases ...; @*)
6.              x:=l[i]; status:=i;
7.              if type(x,integer) then
8.                      if (x = 0) then return [si,sf]; end if; si:=si+x;
9.              elif type(x,float) then
10.                     if (x < 0.5) then return [si,sf]; end if; sf:=sf+x;
11.             end if;
12.         end do;
13.         status:=-1;
14.         return [si,sf];
15.         end proc;
```

Listing 1: The example *MiniMaple* procedure

For further related technical details on the typing and formal specification of above example, see [9]. The application of the type checker to our main test example, the Maple package *DifferenceDifferential*, has identified some bad code parts that can cause problems, e.g. variables that are declared but not used (and therefore cannot be type-checked) and variables that have duplicate (global and local) declarations. Also we have formally specified a substantial part of *DifferenceDifferential* as the main test for the specification language. We have specified the low-level (called by high-level) procedures of the test package by low-level (concrete type based) specifications, while some of the high-level (which call low-level) procedures are specified by abstract data type-based specifications.

3 Formal Verification

For the verification of an annotated *MiniMaple* program, we first need to generate verification conditions from the program and then to prove the correctness of these conditions by some automatic decision procedure or interactive theorem prover. In principle, we could generate verification conditions either by our own as in the RISC ProgramExplorer [10] or by some existing verification frameworks, e.g. Why3 [2] developed at LRI, France or Boogie [1] developed by Microsoft. Based on preliminary investigations, we decided to use Why3, which is discussed in the following section.

3.1 Why3

Why3 is a verification tool for the programming language Why3ML whose core is a verification condition generator as depicted in Figure 1. The generated verification conditions are translated into a logical specification language called Why for which translation to various back-end theorem provers is provided [4].

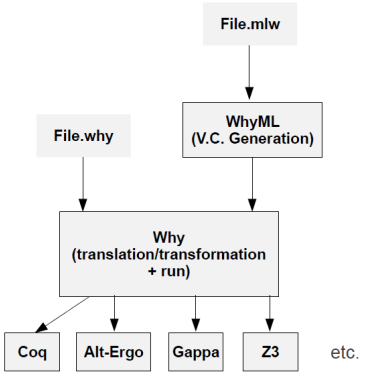

Figure 1: Overview of Why3

Why3 was developed as a generic intermediate verification platform supporting various front-end tools, e.g. Krakatoa (for Java programs) and Frama-C (for C programs). Why3ML is a first order functional language influenced by ML that supports pattern matching, inductive predicates, algebraic data types and also supports typical imperative constructs (loops, sequence, exceptions etc.). Why3 does support various automated provers (e.g. Z3 and CVC3) and proof assistants (e.g. Coq). The wide range of proof support was one the reasons why we chose Why3, as we are, e.g. dealing with non-linear arithmetic which requires in general an interactive prover. The existence of a big-step operational semantics of Why3ML [6, 5] is the other reason for choosing this system, because one can precisely argue whether the generated verification conditions are sound with respect to the *MiniMaple* semantics.

3.2 Translation

In this section we discuss the translation of annotated *MiniMaple* to Why3ML. The goal here is to automatically translate a *MiniMaple* program into a semantically equivalent Why3ML program such that verification conditions generated by Why3 are sound w.r.t. *MiniMaple* semantics.

The formal definition of our translation has 40 valuation functions, approx. 50 auxiliary functions and predicates and contains 45 pages [7]. Some of the main features of translation are as follows:

- *MiniMaple* supports a **return** statement which is not supported in Why3ML. The **return** statement is translated with the help of the Why3ML exception-handling mechanism: where-ever **return** statement occurs, we assign values to the corresponding exception-object and then raise an exception which is caught by a corresponding handler in the program.
- In contrast to Why3ML, *MiniMaple* supports a multi-assignment command. We translate this statement by a nested local binding in Why3ML.
- Why3ML supports very limited data types, e.g. integers, reals, strings, tuples and lists. We axiomatize all other *MiniMaple* types and their corresponding operations. For example, type **set**(T) is axiomatized with the underlying Why3ML list representations where the elements of the set are some permutation of the list elements.
- The union type **Or**(*Tseq*) is defined as an algebraic data type with one constructor for each type in *Tseq*.

In the following we give an example of a command translation function, which takes a *MiniMaple* command C, a *MiniMaple* type environment Env_m, a Why3 environment Env_w, global declarations $Decl_w$ and a theory $Thry_w$ and returns the correspondingly translated Why3ML expression Exp_w, declarations $Decl_w$ and theory $Thry_w$:

$$\begin{aligned}&\mathrm{T}\ [\![C]\!]\colon Env_m \times Env_w \times Decl_w \times Thry_w \to\\ &\qquad Exp_w \times Env_w \times Decl_w \times Thry_w\end{aligned}$$

The definition of T for the case of C being a *MiniMaple* **for-while** loop is as follows:

$$\begin{aligned}&\mathrm{T}[\![\textbf{for } I \textbf{ in } E_1 \textbf{ while } E_2 \textbf{ do } Cseq \textbf{ end do}]\!](e_m, e_w, d_w, t_w) =\\
&\ (inWhy3_Exp(\textbf{let } I_0 = \textbf{ref } 0 \textbf{ in}\\
&\qquad \textbf{while } I_0 < \mathrm{op_length}(exp1_w) \ \&\ exp2_w \textbf{ do}\\
&\qquad\quad \textbf{let } I = \mathrm{op_nth}(I_0, exp1_w) \textbf{ in}\\
&\qquad\qquad exp3_w; I_0 := !I_0 + 1\\
&\qquad \textbf{done}), e3_w, d3_w, t3_w)\end{aligned}$$

$$\begin{aligned}&(exp1_w, e1_w, d1_w, t1_w) = \mathrm{T}\ [\![E_1]\!](e_m, e_w, d_w, t_w),\\
&(exp2_w, e2_w, d2_w, t2_w) = \mathrm{T}\ [\![E_2]\!](e_m, e1_w, d1_w, t1_w),\\
&(exp3_w, e3_w, d3_w, t3_w) = \mathrm{T}\ [\![Cseq]\!](e_m, e2_w, d2_w, t2_w),\\
&exp_type1 = getExpType(exp1_w, e1_w),\\
&op_length = access(\text{“length”}, exp_type1, e1_w),\\
&op_nth = access(\text{“select”}, exp_type1, e1_w)\end{aligned}$$

The *MiniMaple* **for-while** loop checks at the start of every iteration both loop conditions, i.e. I **in** E_1 and E_2; if any of them is *false* the body of the loop is not executed. Moreover, the identifier I is used in the body of the loop representing the *ith* (iteration) element of expression E_1. For this semantically equivalent translation we proceeded as follows:

1. Declare (locally bound) an auxiliary variable I_0 to track the iteration number and initialize it with value 0.
2. Translate the member-based loop condition, i.e. I **in** E_1 into a corresponding iteration-bounded condition $I_0 < op\_length(exp1_w)$ and combine it with the while-loop condition ($exp2_w$) which we get from the translation of the corresponding *MiniMaple* expression E_2.
3. Declare I as a local variable and at the *ith* iteration (represented by I_0) assign it the *ith* value of the translated expression $exp1_w$.
4. Increment I_0 at the end of the iteration.

The generic operation $access(op, exp\_type, e_w)$ returns the name of the concrete operation *op* as generated by the translator for the type expression exp_type in the environment e_w. In above example, the *access* function returns the names of the concrete operations "length" and "select" of the expression type exp_type1.

```
theory SumList
  ...
  type or_integer_float = Integer int | Real real
  ...
end
module SumListImpl
   ...
  val status: ref int
  ...
  exception Break
  ...
  let sum (l: list or_integer_float) : (int, real) =
  {...} (* pre-condition *)
  try
      for i = 0 to length l - 1 do
        invariant { ... } (* loop-invariants *)
        status := i;
        match get i l with (* type-test translation *)
        | Integer n -> if n = 0 then
                 raise Break; si := !si + n
        | Real r -> if r <. 0.5 then
                 raise Break; sf := !sf +. r
        end
      done;
     status := -1;
      (!si, !sf)
    with Break -> (* exception-handler *)
      (!si, !sf)
   end
   {...} (* post-condition *)

end
```

Figure 2: *MiniMaple* to Why3 Translation

The translator is implemented in Java and contains approximately 80+ classes and 13K+ lines of code. Figure 2 sketches as an example (manually

generated for readability) translation of the *MiniMaple* annotated program presented in Section 2 into a Why3ML program consisting of a theory and a module. For further illustration, the various code parts of the translation are annotated with Why3ML comments (* ... *).

In the theory, we declare the types used in the corresponding module respective *MiniMaple* program; e.g. the *MiniMaple* union type **Or(integer, float)** is translated to an algebraic data type with two corresponding constructors for integers and floats respectively. The module contains the declarations arising from the translation of the *MiniMaple* procedure, a global variable *status*, the auxiliary exception *Break*, and the translation of the procedure *sum* itself. This procedure also contains a translation of *MiniMaple* **for** loop to a corresponding Why3ML loop. The type tests of *MiniMaple* are translated using the pattern matching feature of Why3ML: the **match** construct matches the *ith* element of the list with the corresponding constructor of the type of the list elements. The *MiniMaple* **return** statement is translated into an equivalent exception-handling mechanism by an auxiliary exception object *Break*, i.e. we throw the exception *Break* whereever the **return** statement occurred and then catch this exception in the corresponding handler as shown in the **with** construct. Finally, in the handler, we return the value of the corresponding result tuple.

As an application of our translator, we have translated a substantial part of the *DifferenceDifferential* package to Why3ML.

3.3 Example Verification

In this section we discuss the verification of the example program, which was generated by the translator in the previous section. For this purpose, we use the GUI-based interface of Why3 to generate verification conditions and to prove them as shown in Figure 3.

The Why3 GUI displays three columns:

1. The left column lists the configured theorem provers.
2. The middle column shows the verification conditions generated (respectively required to be proved correct).
3. The right column shows the contents of the goals (verification conditions). Actually, the right column has two parts, the upper part shows the corresponding Why contents of the selected goals, while the lower part highlights the corresponding Why3ML code from which the selected goal is generated.

In our example, the proof of the correctness of the procedure results in the following four goals as shown in the middle column of Figure 3:

1. a normal postcondition,
2. the for-loop (invariant) initialization,
3. the for-loop (invariant) preservation and
4. a normal postcondition.

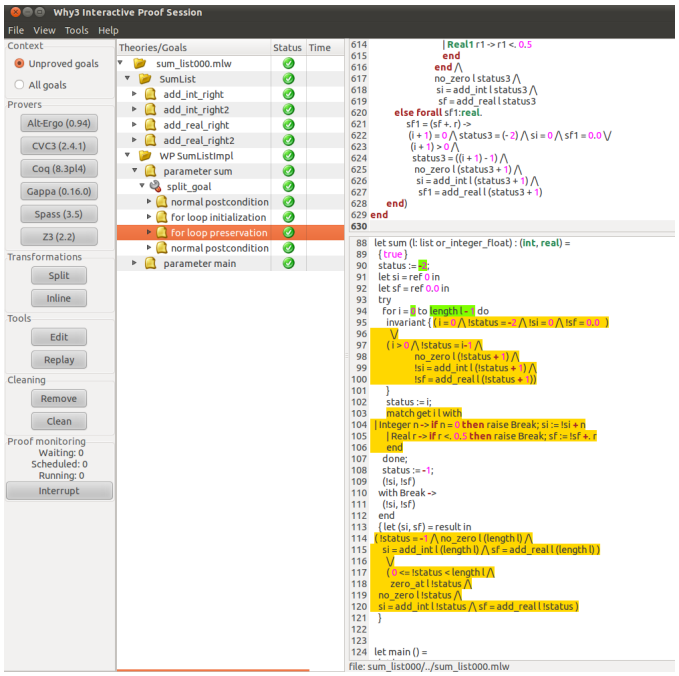

Figure 3: Verification of Example Program

The first and the last goal are about proving the correctness of postconditions. The first goal is to prove the postcondition when the loop body is not executed, while the last goal is to prove the postcondition when the loop body is executed, which requires invariant-based reasoning.

The second goal is to prove that the loop invariant holds at the start of the execution of the loop. The third goal is to prove that the loop invariant is preserved by the execution of the loop and must also hold when the loop terminates.

To run through the prove of these generated verification conditions, it was required to add some lemmas manually (at Why3 level) because the function definitions generated by the translator from the corresponding numeric quantifiers (**add**) appearing in the *MiniMaple* procedure specification were not adequate for this proof. These lemmas introduce the facts that the translated addition functions (*add_int* and *add_real* over lists) correctly handle the integers and reals in the list; e.g. the lemma of integers for integer-addition function *add_int* is as follows:

$$
\begin{aligned}
&\mathit{lemma}\ \mathit{add\_int\_right}: \\
&\quad \forall\ e : \mathit{list}\ \mathit{or\_integer\_float}, j : \mathit{int}. \\
&\quad 0 \le j < \mathit{length}\ e \rightarrow \forall n : \mathit{int}. \\
&\quad \mathit{nth}\ j\ e = \mathit{Some}\ (\mathit{Integer}\ n) \\
&\qquad\qquad \rightarrow \mathit{add\_int}\ e\ (j+1) = \mathit{add\_int}\ e\ j + n
\end{aligned}
$$

By the introduction of these lemmas, all of the verification conditions could be proved with the automatic decision procedures Alt-Ergo and Z3. On the other hand, we had to prove these lemmas manually by induction using the

interactive theorem prover Coq. In the future, we will generate these lemmas automatically as axioms: then the proof of the aforementioned generated verification conditions is automatic.

Using our verification framework we have verified a substantial part of our test package *DifferenceDifferential* that mainly involves the low-level (called by high-level) procedures with concrete data type based specifications. In fact, the verification required some pre-processing where we analyzed and partially revised the corresponding parts of the package such that the formalism was adequate enough to perform the proofs for the correctness of the generated verification conditions.

4 Conclusions and Future Work

Currently we are in the process of verifying some higher-level procedures (which call low-level procedures) of the Maple package *DifferenceDifferential* which involve abstract data type-based specifications. To verify these procedures, we have, based on a simpler example, experimented with adequate formulations and proof strategies for the verification of such specifications. The challenge is to prove the correctness of the behavior of higher-level procedures, which have

- on the one hand an implementation based on concrete data types (e.g. the difference-differential operator is implemented as a list of its terms as tuples) and
- on the other hand are specified by abstract data types (e.g. the difference-differential operator is specified by an abstract data type "addo" with corresponding operations and mathematical properties).

To address this challenge, we have defined a general logical formulation for any such formally specified procedure as follows:

1. An *abstract model* defines an abstract (data type-based) object A and specifies its associated operations and properties.
2. A concrete type *representation* C is defined, which is used as the underlying implementation of the abstract object A.
3. A mapping function "$abstract : C \to A$" maps a concrete representation to the corresponding abstract type.
4. A concretization predicate "$concrete \subseteq C \times A$" defines a relationship between a concrete type representation C and its corresponding abstract type representation A.
5. An invariant predicate "$invariant \subseteq C$" holds only for those elements of C that can be created by the corresponding constructor operations.
6. A *lemma* states that for all the valid representations c of concrete type C, if we abstract the concrete representation c to a, then the concretization relation holds between c and a. The lemma can be formulated as

$$\forall c : C, a : A, invariant(c) \Rightarrow (a = abstract(c) \Leftrightarrow concrete(a, c))$$

This lemma makes the subsequent proof easier.

7. The specification of an implementation based on concrete type C employs the abstract type A by applying the map function *abstract* to the concrete argument/result values.

The translation of *MiniMaple* to Why3ML must be sound w.r.t. the formal semantics of *MiniMaple* and Why3ML; we can formulate a corresponding soundness statement for the translation of expressions as follows:

$$\forall\ E \in Expression_m, s_1, s_2 \in State_m, e \in Env_m,\ v \in Value_m, v_1 \in Value_w, \ldots :$$
$$[\![E]\!](e)(s_1, s_2, v) \Leftrightarrow \langle\ \mathrm{T}[\![s_1]\!](e),\ \mathrm{T}[\![E]\!](e, \ldots)\ \rangle_w \Downarrow \langle\ \mathrm{T}[\![s_2]\!](e),\ v_1\ \rangle_w \Rightarrow v \equiv v_1$$

This statement means that if the evaluation of a *MiniMaple* expression E in a given environment e and a pre-state s_1 yields a value v and results in a post-state s_2 then, it is equivalent to the evaluation of a translated Why3 expression $\mathrm{T}[\![E]\!](e, \ldots)$ in a translated pre-state $\mathrm{T}[\![s_1]\!](e)$ that yields a value v_1 and results in a translated post-state $\mathrm{T}[\![s_2]\!](e)$ and the result values v and v_1 are semantically equivalent. The proof of this statement is planned as a next goal.

Acknowledgment

The author cordially thanks Wolfgang Schreiner for his valuable and constructive comments and suggestions throughout this work.

5 References

[1] Mike Barnett, Boryuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

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

[3] Christian Dönch. Bivariate Difference-Differential Dimension Polynomials and Their Computation in Maple. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University (JKU), Linz, 2009.

[4] Jean-Christophe Filliâtre. Verifying Two Lines of C with Why3: an Exercise in Program Verification. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, Philadelphia, USA, January 2012.

[5] Jean-Christophe Filliâtre . Why: an Intermediate Language for Program Verification. A Tutorial Lecture at Summer School, 2007. https://www.lri.fr/~filliatr/types-summer-school-2007/notes.pdf.

[6] Jean-Christophe Filliâtre. One Logic To Use Them All. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Lake Placid, USA, June 2013. Springer.

[7] Muhammad Taimoor Khan. Translation of *MiniMaple* to Why3ML. DK Technical Report 2013-02, Doktoratskolleg, Linz, February 2013.

[8] Muhammad Taimoor Khan and Wolfgang Schreiner. On the Formal Specification of Maple Programs. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics*, volume 7362 of *LNCS*, pages 443–447. Springer, 2012.

[9] Muhammad Taimoor Khan and Wolfgang Schreiner. Towards the Formal Specification and Verification of Maple Programs. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics*, volume 7362 of *LNCS*, pages 231–247. Springer, 2012.

[10] Wolfgang Schreiner. Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs. In Pedro Quaresma and Ralph-Johan Back, editors, *Proceedings First Workshop on CTP Components for Educational Software (THedu'11)*, number 79 in Electronic Proceedings in Theoretical Computer Science (EPTCS), pages 124–142, Wroclaw, Poland, July 31, 2011, February 2012.