Program Verification: Lecture 14

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Verification of Functional Modules

We are now ready to consider a general methodology for verifying declarative programs. We will present the ideas in the context of verifying Maude functional modules, which are based on equational logic. The first key observation is that there are three viewpoints involved:

- the customer's viewpoint, expressed in the form of requirements that the desired software should satisfy;
- the implementor's viewpoint, whose job is to write a program meeting the customer's requirements; and
- the verifier's viewpoint, whose responsibility is to verify that the implementation does indeed meet the customer's requirements.

The Customer's Requirements and Specification

The customer's requirements may generally be informal. Furthermore, they may involve other concerns beyond correctness, such as user-friendliness, a good graphical user interface, performance requirements, requirements about the underlying hardware and systems software, interoperability requirements, and so on.

Program verification focuses primarily on correctness requirements, which are always important, but may be crucial for safety-critical applications, were incorrect software may cause loss of human lives and/or other important damages.

The Customer's Requirements and Specification (II)

To make possible the high assurance of correctness afforded by mathematical verification, such correctness requirements must be formalized, typically in the form of a logical theory T_{spec}, stating precisely the customer's (correctness) specification.

This capture of the informal correctness requirements into a formal specification can be done by the customer himself, or by an expert aiding the customer in this task. It is of course very important to make sure that the formal specification captures faithfully the informal requirements.

The Customer's Formal Specification

In the context of Maude functional modules, it is reasonable to assume that such a formal specification will take the form of a theory,

$$T_{spec} = (\Delta, E_0 \cup Q)$$

where:

- (Δ_0, E_0), with $\Delta_0 \subseteq \Delta$, is an equational theory, that could be called the framework theory, specifying things such as key data structures and functions, including auxiliary functions needed to state key properties, and
- Q is a collection of sentences in first-order logic, specifying the actual correctness properties that the software must satisfy within the (Δ, E_0) framework.

Customer Specification: A Sorting Example

We can illustrate these ideas with a simple example, namely a customer who wants a sorting program to sort lists of integers.

As already mentioned, the customer's requirements may involve other important considerations, such as reasonable efficiency; for example, that it returns answers in time at most quadratic on the size of the input list.

Informally, the correctness requirement seems both obvious and tautological, namely, the program should return the input list in sorted form.

Customer Specification: A Sorting Example (II)

However, in order to prove that a given implementation satisfies such a requirement, we need to capture such an informal requirement in a formalized way as a theory T_{spec}.

We must specify two things:

1. the data, namely lists of integers, and some auxiliary functions, and

2. the sorting function and its properties.

Customer Specification: A Sorting Example (III)

Specifying the properties of the sorting function is not entirely trivial:

- first of all, we need to make precise what we mean by a list being sorted;
- but it is not enough to just require that the result is sorted: a function returning always the empty list will satisfy such a requirement! The original list and the sorted list should have the same elements.

All this can be stated precisely in three equational theories:

Customer Specification: A Sorting Example (IV)

First, the data, say lists of numbers, is specified in a module such as the following INT-LIST module

```
fmod INT-LIST is protecting INT .
  sorts List .
  op nil : -> List [ctor] .
  op _:_ : Int List -> List [ctor] .
endfm
```

Customer Specification: A Sorting Example (V)

Then, a framework theory importing INT-LIST,

```
fmod FRAME-SORTING-REQUIREMENTS is protecting INT-LIST .
  sort Multiset .
  subsort Int < Multiset .
  op sorted : List -> Bool .
  op null : -> Multiset .
  op _ _ : Multiset Multiset -> Multiset [assoc comm id: null] .
  op mset : List -> Multiset .
  vars N M : Int .
  var L : List .
  eq sorted(nil) = true .
  eq sorted(N : nil) = true .
  ceq sorted(N : M : L) = sorted(M : L)  if (N <= M) = true .
  ceq sorted(N : M : L) = false  if N <= M = false .
  eq mset(nil) = null .
  eq mset(N : L) = N mset(L) .
endfm
```

Customer Specification: A Sorting Example (VI)

Finally, a functional theory specifying two key requirements for the `sort` function:

```
fth SORTING-REQUIREMENTS is
  protecting FRAME-SORTING-REQUIREMENTS .
  op sort : List -> List .
  var L : List .
  eq sorted(sort(L)) = true .
  eq mset(sort(L)) = mset(L) .
endfth
```

Customer Specification: A Sorting Example (VII)

This is an instance of our general methodology, where the correctness specification has the form, $T_{spec} = (\Delta, E_0 \cup Q)$. Here, the framework theory (Δ_0, E_0) is the module `FRAME-SORTING-REQUIREMENTS`, and the theory T_{spec} itself is `SORTING-REQUIREMENTS`.

T_{spec} is a Maude theory (see Section 8.3.1 of "All About Maude") introduced with the keywords `fth`T_{spec}`endth`. This means that it has a loose semantics; that is, we do not require its models to be initial. However, because of the keyword `protecting FRAME-SORTING-REQUIREMENTS`, the functional submodule `FRAME-SORTING-REQUIREMENTS` is imported with its initial semantics.

Customer Specification: A Sorting Example (VIII)

Mathematically, what this means is that, for $\mathcal{A}$ to be an acceptable model of T_{spec}, besides having to satisfy the axioms in T_{spec}, the data types of lists, multisets, integers, and of booleans, as well as all the functions defined on them by initial algebra semantics (with keywords `fmodFRAME-SORTING-REQUIREMENTSendfm`) must be respected. In short, we must have an isomorphism,

$$\mathcal{A}|_{\Sigma_{\texttt{FRAME-SORTING-REQUIREMENTS}}} \cong \mathcal{T}_{\texttt{FRAME-SORTING-REQUIREMENTS}}.$$

The Implementation: Insert-Sort

One possible Maude implementation is insert-sort,

```
fmod INSERT-SORT is
  protecting INT-LIST .
  op ins : Int List -> List .
  op sort : List -> List .
  vars N M : Int .
  var L : List .
  eq ins(N, nil) = N : nil .
  ceq ins(N, M : L) = N : M : L if N <= M = true .
  ceq ins(N, M : L) = M : ins(N, L) if N <= M = false .
  eq sort(nil) = nil .
  eq sort(N : L) = ins(N, sort(L)) .
endfm
```

The Implementation: Insert-Sort (II)

The module T_{imp} in our general methodology, becomes in this case the Maude module `INSERT-SORT`.

Note that the auxiliary function `ins` defined in `INSERT-SORT` for implementation purposes is completely different from the auxiliary functions, `sorted`, `_ _`, and `mset` defined in `FRAME-SORTING-REQUIREMENTS` for specification purposes, to formally capture the customer's correctness requirements.

Therefore, the signatures of T_{spec} and T_{imp} do not necessarily coincide. However, for verification purposes, they are both included as subsignatures in T_{ver}.

The Theory T_{ver}

For verification purposes we typically need to use the auxiliary functions defined in both the framework theory (Δ_0, E_0) and in $T_{imp} = (\Sigma, E)$. This means that the theory $T_{ver} = (\Sigma', E')$ will typically have theory inclusions,

- $(\flat)$ $(\Delta_0, E_0) \subseteq (\Sigma', E')$, and
- $(\dagger)$ $(\Sigma, E) \subseteq (\Sigma', E')$.

The main goal of the verification effort is then to establish,

$$T_{ver} \vdash_{ind} Q.$$

But for this inductive inference to be applicable to the implementation theory $T_{imp} = (\Sigma, E)$, we need to require that $(\dagger)$ is a protecting inclusion, so that we have an isomorphism, $\mathcal{T}_{\Sigma'/E'}|_{\Sigma} \cong \mathcal{T}_{\Sigma/E}$.

The Theory T_{ver} and its Verification

In this example, the theory T_{ver} must contain both the framework theory and the implementation theory:

```
fmod INSERT-SORT-VERIFICATION is
protecting INSERT-SORT .
protecting FRAME-SORTING-REQUIREMENTS .
endfm
```

We can then give this theory to the ITP and prove in it as theorems the two equations in T_{spec}, namely,

```
  eq sorted(sort(L)) = true .
  eq mset(sort(L)) = mset(L) .
```