Separating Methodology and Specification Constructs

Dan Putnam
Gould Inc., Software Division
Urbana, Illinois 61801

This paper considers whether a specification language ought to enforce a particular modeling methodology or whether the language should allow flexibility in the choice of a model. It is suggested that modeling considerations ought to be kept out of a specification language, and that separate methodology enforcement tools be provided where necessary. A general scheme for doing this is outlined below. The scheme is illustrated with a description of how a state machine methodology is enforced in the VERUS™ design verification system.

Languages used in design verification typically allow the specification writer to describe a system as a collection of objects that stand in various relationships with one another. Some languages go further and enforce a particular modeling methodology, for example a state machine modeling methodology. These languages require that the specification writer declare the constructions of a specification according to their intended modeling roles. The language processor can then use the given constructions to formulate the theorems required by the particular methodology.

For example, a state machine specification in the manner of the Bell and LaPadula model will define allowable state transitions and correct system states. A language that explicitly enforces this kind of state machine methodology includes language constructs for expressing these concepts. Using these constructs, the language processor checks that the specification is a correctly formulated state machine specification and generates the appropriate theorems.

A drawback of this approach is that this kind of specification language is limited in the kind of models that it can treat. If the processor expects to see a list of state transition predicates and expects to generate theorems of the appropriate form, then the specification writer can only model state machines. A language specialized for writing state machine specifications may be inherently unable to express other kinds of models, for example, those dealing with concurrency.

It is suggested instead that a specification language be general enough to specify any sort of abstract system. The scheme outlined below provides a means of enforcing a modeling methodology outside of the basic specification language. The main advantage of this approach is that it allows the specification language to be used for more than one kind of modeling. The disadvantage is that the resulting language tends to be low-level since it does not impose a particular view of systems on the user. However, if separate methodology enforcement tools are provided, the user still has the benefit of an automated check on the formulation of a specification and its theorems.

One approach to separate methodology enforcement is to provide an additional file, outside the specification, that identifies the specification components and their roles. With these declarations, the methodology tool can inspect the actual specification to be sure that each of the components is present and properly formulated to play its given role. Further, the utility can check that all the theorems that need to be proved are present in the specification files and correctly proved.

The state machine methodology tool for the VERUS language works in just this way. For example, a VERUS state machine specification must contain a condition that defines the notion of a correct state. In the VERUS specification this condition appears only as a named predicate, no different than any other Boolean-valued expression. The special role of this condition is declared in the input file to the methodology tool. Thus, the methodology tool has the information needed to check that that the condition is properly formulated and used in the specification. With the other declarations in the input file, the checker can make a set of checks to assure that the specification correctly follows the state machine methodology. The specification fragments given below illustrate this approach.

The following text in VERUS defines an initial condition and a state requirement:

```
TYPE Time;
VAR t : Time;
CONST INIT : Time;

DEFINE InitialCond(t) : BOOLEAN
BY ...   (VERUS statements defining the initial condition)

DEFINE StateReq(t) : BOOLEAN
BY ...   (VERUS statements defining a correct state)
```

Later in the VERUS specification, the following theorem is stated and proved to show that the system begins in a correct state:

```
PROVE
   IF InitialCond(INIT)
   THEN StateReq(INIT);
{
   ...   (VERUS proof steps)
};
```

The following text is not expressed in VERUS, but in the input language to the specification checker. It simply identifies the VERUS predicates given above and declares their intended roles. The methodology tool checks that both terms are time-dependent Booleans and makes sure that the above theorem is present and correctly proved.

```
...

INITIAL CONDITION   InitialCond;
STATE REQUIREMENT   StateReq;
...
```

Other declarations, not shown here, identify the remaining state machine specification components, including the names of the state functions, state transition definitions, and axioms used in the specification. These declarations are used to make similar checks, ensuring that the entire specification correctly follows the state machine methodology.

VERUS is a trademark of Gould Inc.