Maude2PVS

Sam Owre

owre@csl.sri.com

URL: http://www.csl.sri.com/~owre/

Computer Science Laboratory

SRI International

Menlo Park, CA

January 29, 2007

Protocols in Maude

- Maude provides an expressive language which is convenient for prototyping, search, and model checking
- This makes it quite good for prototyping protocols, for example, Strand Spaces
- There are extensions to Maude that provide some formal method support, but they are limited:
  - No quantifiers
  - No support for higher-order terms, in particular induction
  - No support for developing complex proofs

Maude and PVS

- There have been many suggestions in the past to integrate Maude and PVS:
  - Using Maude as a proof rewrite rule
  - Generating Maude executable specifications from the PVS ground evaluator
  - Translating Maude specifications to PVS
- Translating to PVS allows prototypes to be developed and tested in Maude, then translated to PVS for proof, both for specific protocols and for the meta-theory

Contents

Introduction to Maude

- Maude is based on *rewriting logic*
- Because of this, Maude may be used for programming, specification, and verification
- Maude is declarative, with both a mathematical and operational semantics

Maude Specifications

- *Modules* - these are the basic units of Maude specifications. There are two kinds:

  Functional modules - represent equational theories

  System modules - represent concurrent programs

- *Types* - the Maude type system is based on *order-sorted algebra*

  Sorts - the basic types

  Subsorts - subsets of Sorts

  Kinds - intuitively correspond to "error supertypes"; allow for partial operations

Maude Specifications (cont)

- *Operators* - each operation in Maude is declared with a *name*, *signature*, and optional set of *attributes*
- *Equations* - equational axioms, used for rewriting. May be *conditional*
- *Memberships* - state that a term has a given sort. May be conditional.
- *Rules* - used in system modules to specify state transformations

Example Maude Specification

```
fmod ATOM-SET is
 inc SUBST .
 pr NAT .

 sorts AtomSet .
 subsort Atom < AtomSet .

 var sb : Subst .
 var ams : AtomSet .
 var atm : Atom .
 vars tm0 tm1 : Message .
 var ktm : Key .

 op none : -> AtomSet .
 op __ :  AtomSet  AtomSet ->  AtomSet [ctor assoc comm id: none] .
```

Example Maude Specification (cont)

```
  eq atm atm = atm .
  op _[_] : AtomSet Subst -> AtomSet .
  eq (none).AtomSet [sb] = (none).AtomSet  .
  eq (atm ams)[sb] =  (atm[sb]) (ams[sb]) .

  op size : AtomSet -> Nat .
  eq size(none) = 0 .
  eq size(atm  ams) = s size(ams) .

  op member : Atom AtomSet -> Bool .
  eq member(atm, atm  ams) = true .
  eq member(atm, ams) = false [owise] .

  op atoms : Message -> AtomSet .
  eq atoms(atm) = atm .
  eq atoms((tm0, tm1)) = (atoms(tm0) atoms(tm1)) .
  eq atoms(tm0ktm) = (ktm atoms(tm0)) .

endfm
```

Introduction to PVS

- PVS is a comprehensive verification system with an expressive language, powerful theorem prover, Emacs-based user interface, and many other components

- The language is based on higher-order type theory, with support for functions, tuples, records, cotuples, predicate subtypes, dependent types, and inductive data types

- Typechecking is undecidable, and leads to proof obligations, called *Type correctness conditions* (TCCs)

PVS Specifications

- Specifications consist of a collection of *theories*, each of which primarily consists of types, constants, and formulas
- Theories my be parametrized with types or constants
- Theories may import other theories, providing instances for the parameters

Example PVS Theory

```
group[G: TYPE+]: THEORY
 BEGIN
  a, b, c: VAR G

  0: G
  +(a, b): G
  -(a): G

  ax1: AXIOM a + 0 = a
  ax2: AXIOM a + (b + c) = (a + b) + c
  ax3: AXIOM a + -a = 0
  inv_plus: LEMMA -a + a = 0
  zero_plus: LEMMA 0 + a = a
 END group
```

Overall Design of Maude2PVS

- Maude has very useful reflective capabilities
- Parsing a Maude specification from outside would be very difficult
- For these reasons, this tool is written in Maude

Translations

- Identifiers
- Sorts
- Modules
- Operators
- Equations
- Conditional Equations
- Operator Attributes
- Equation Attributes

Identifiers

- Maude has a very flexible syntax, allowing the user to declare prefix, infix, mixfix, and even "invisible" operators

- For example, list append is often declared in the form
  `__ : List List -> List`

- Then `L1` appended to `L2` is written `L1 L2` or `__(L1, L2)`

- Fortunately, the latter form is what is found at the meta (reflective) level

Mapping Identifiers

- PVS has more restricted identifiers, as well as keywords - similar to conventional programming languages
- Maude2PVS maps identifiers in stages:
  1. look up the identifier in a user-provided identifier map
  2. otherwise check if it is a valid PVS id:
     - if it is, then check if it is a PVS keyword and name it apart by appending '_'
     - if not, translate '-' and ''' to '_' in the identifier
- The result still may not parse in PVS, but it should be easy to determine identifiers that should be added to the map

Types

- Sorts and subsorts are very similar to the PVS notion of type and subtype

- But there are some subtle differences:

  - PVS subtypes have associated predicates - operators applied to terms not known to be of the associated subtype lead to proof obligations

  - Maude does not enforce subsorts on operators

  - Sorts form a lattice, as in PVS - however, unlike PVS, initially disjoint sorts may later be connected

Translating Types

- We translate Maude *kinds* into uninterpreted (nonempty) PVS types
- Sorts are mapped to (nonempty) uninterpreted subtypes
- Example:

```
sorts Name Key Nonce Text Atom Message .
subsorts Name Key Nonce Text < Atom < Message .
```

- Maps to:

```
Message: TYPE+
Atom: TYPE+ FROM Message
Atom?(x: Message): MACRO bool = Atom_pred(x)
Key: TYPE+ FROM Atom
Key?(x: Message): MACRO bool = Atom?(x) and Key_pred(x)
...
```

Modules

- Maude functional modules are translated to PVS theories

- Because newly loaded Maude modules may connect previously disjoint sorts, the translation should only be done after all Maude modules have been loaded

- Not even the Maude *prelude* may be preprocessed, as the type lattice may change as new modules are loaded

Operators

- Operators are mapped to PVS constants
- The signature is *lifted* to the kind level
- This is what Maude does, as experiments show
- Equations do respect (sub)sorts

Equations

- Equations are mapped to PVS axioms:

```
eq lookup ((av <- atm) sb, av) = (av <- atm)  .
```

- Maps to:

```
eq10: AXIOM FORALL (sb: Subst, av: Atom, atm: Atom):
  lookup(append(assign(av, atm), sb), av)
    = assign(av, atm)
```

- Conditional equations are simply mapped to a PVS `WHEN` expression

Operator Attributes

- There are a number of attributes associated with Maude operator declarations:

  Current: assoc, comm, idem, id, left id, right id

  Future: ditto, iter, ctor, metadata

  Ignored: poly, obj, msg, memo, strat, special, format, frozen, prec, gather, config

- The currently supported attributes lead to straightforward PVS axioms:

```
op __ : Subst Subst -> Subst [ctor assoc comm id: none] .
```

- Maps to:

```
append_assoc: AXIOM associative?(append)
append_comm: AXIOM commutative?(append)
append_id: AXIOM identity?(append)(none)
```

Equation Attributes

- Equation attributes include `nonexec`, `otherwise`, `metadata`, and `label`
- `otherwise` (`owise`) is translated to a conditional equation in PVS
- Example:

```
eq member(atm, atm  ams) = true .
eq member(atm, ams) = false [owise] .
```

- translates to:

```
eq6: FORALL (atm: Atom, ams: AtomSet):
       member(atm, append(atm, ams)) = true
eq7: FORALL (atm: Atom, ams: AtomSet):
       (NOT EXISTS (atm1: Atom, ams1: AtomSet):
              member(atm, ams) = member(atm1, append(atm1, ams1)))
         IMPLIES member(atm, ams) = false
```

Translation and Proof Obligations

- The translation not only allows reasoning about the Maude specification, but should generate various proof obligations

- For example, modules may be imported using `including`, `protecting`, or `extending`

- Each entails constraints that are up to the user to prove

- Details have not been worked out, but the translation should be able to generate these obligations

Difficulties in Using the Translation

- In general, the generated theories will be difficult to use and reason about directly in PVS
- Axioms generated from `owise` equations will be especially difficult to use as they involve existential conditions that must be checked
- The translation is fairly direct, but makes little use of some advanced features of PVS: abstract datatypes, (recursive) definitions, dependent types, judgements, etc.

Theory Interpretations

- The solution is to develop a PVS specification separately, making use of all the features of PVS
- Then show that specification is a *theory interpretation* of the Maude specification
- Under the interpretation, axioms are mapped to proof obligations
- Discharging these guarantees that the interpretation is sound
- Of course, this does **not** say anything about the Maude2PVS translation, which must be verified by hand

Current Status

- Muade2PVS is currently able to translate part of the Strand Space specification developed by Carolyn Talcott
- This is driving the development, giving priority to the Maude constructs actually used
- This includes the Identifier translations, operators, sorts, and (conditional) equations
- Currently working on `owise` equations

Future Work

- System modules
- Getting the generated string into a PVS file - probably using the `LOOP-MODE` module
- Extending the attribute list to include PVS specific annotations:
  - Mapping sorts to existing PVS types and datatypes
  - Mapping to an existing operator rather than creating a new one