Object-based Systems and Meta-interpreters

Maude supports formally specifying concurrent systems in an object-based style where concurrent object systems are modeled by a set of *objects* that can interact with each other by sending and receiving *messages*. Moreover, meta-interpreters are a new feature in Maude that are external objects representing external entities with independent Maude interpreters, which have their own module and view databases and maintain their own states. Each meta-interpreter is run as a separate process that can send and receive messages to/from other objects from which each job can be handled by a meta-interpreter independently. Therefore, we use object-based programming and meta-interpreters in Maude to build the parallel version of Maude-NPA in the paper.

1 OBJECT-BASED SYSTEMS

We can formally specify concurrent systems as object-based systems using the module `CONFIGURATION` in Maude. The state of an object-based system is called a *configuration* that is a multiset of objects and messages whose sort is `Configuration`. Configurations are denoted by an empty syntax (`none`) and a juxtaposition operator (`__`), which is declared associative and commutative and has `none` as its identity. For object syntax, there are four sorts introduced: `Oid` (object identifiers), `Cid` (class identifiers), `Attribute` (an attribute of an object), and `AttributeSet` (multisets of attributes). In this syntax, an *object* is a term of the sort `Object` with the following form:

< O : C | att_1, ..., att_n >

where `O` is the object's identifier of the sort `Oid`, `C` is the object's class identifier of the sort `Cid`, and $att_1, \ldots, att_n$ are the object's attributes of the sort `AttributeSet`. A *message* is a term of the sort `Msg` where the declaration defines the syntax of the message $m(v_1, \ldots, v_n)$ and the sorts $s_1, \ldots, s_n$ of its parameters $v_1, \ldots, v_n$ as follows:

op m : s_1 ... s_n -> Msg [ctor] .

Although messages do not have a fixed syntactic form, we follow a convention that the first and second arguments of a message are the identifiers of its destination and source objects, respectively. For example, let us specify a client-server communication in which there are several clients and servers, and the status of each server or client is either $idle$ or $busy$. Each server can have many clients but each client can communicate with only one server. If a client C is $idle$, the client sends a request `N`, a natural number, to a server S and becomes $busy$ (see the rewrite rule `req` below). If the server S is $idle$, then S receives the request (message), becomes $busy$, and returns `N + 1` to C as a message (the rewrite rule `repl`), meaning that the server increments `N`. Suppose that only the server knows how to increment a natural number. A $busy$ client can receive an answer and become $idle$ (the rewrite rule `recv`), and a $busy$ server can become $idle$ at any time (the rewrite rule `idle`). The system can be specified by the following system module:

```
mod CLIENT-SERVER is
  protecting NAT .
  including CONFIGURATION .

  sorts Status .

  ops Client Server : -> Cid [ctor] .
  ops idle busy : -> Status [ctor] .
  op status :_ : Status -> Attribute [ctor] .
  op val :_ : Nat -> Attribute [ctor] .
  op to :_ : Oid -> Attribute [ctor] .
  op m : Oid Oid Nat -> Msg [ctor] .

  vars N N' : Nat .
  vars C S : Oid .

  rl [req]: < C : Client | status : idle, val : N,
    to : S >
  => < C : Client | status : busy, val : N,
    to : S > m(S, C, N) .
  rl [repl]: < S : Server | status : idle >
    m(S, C, N)
  => < S : Server | status : busy >
    m(C, S, (N + 1)) .
  rl [recv]: < C : Client | status : busy, val : N,
    to : S > m(C, S, N')
  => < C : Client | status : idle, val : N',
    to : S > .
  rl [idle]: < C : Server | status : busy >
  => < C : Server | status : idle > .
endm
```

where the natural number that needs to be incremented by a server is stored in the `val` attribute of a client.

Let us suppose that there is a server `s` communicating with two clients `c1` and `c2` in the client-server system. Initially, the status of each `s`, `c1`, and `c2` is $idle$, the values of the `val` attributes of `c1` and `c2` are 3 and 4, respectively, and the value of the `to` attribute of each `c1` and `c2` is `s`. The initial state (referred to as `init`) and the object identifiers (`s`, `c1`, and `c2`) are defined in the following system module:

```
mod CLIENT-SERVER-TEST is
  extending CLIENT-SERVER .

  ops s c1 c2 : -> Oid .
  op init : -> Configuration .

  eq init = < s : Server | status : idle >
  < c1 : Client | status : idle, val : 3, to : s >
  < c2 : Client | status : idle, val : 4, to : s > .
endm
```

Given `init`, the rewrite rule `req` can be applied to the term expressing it at two positions, meaning that there are at least two execution traces that start with `init`, making many possible traces in a concurrent system.

2 META-INTERPRETERS

We can work with meta-interpreters using the module `META-INTERPRETER` in Maude that contains several sorts, constructors, a built-in object identifier `interpreterManager`, and a collection of command and response messages. There are some key messages as follows:

- The `createInterpreter` message is sent to the object `interpreterManager` to request creating a new instance of meta-interpreters; and the `createdInterpreter` message is sent back from the object with a meta-interpreter identifier created if successful. We can communicate with the meta-interpreter instance using this identifier.
- The `insertModule` and `insertView` messages are sent to a meta-interpreter instance to request loading a module and a view into the meta-interpreter, respectively; the `insertedModule` and `insertedView` messages are sent back from the meta-interpreter if the module and the view are loaded successfully, respectively.
- The `reduceTerm` message is sent to a meta-interpreter instance to request simplifying (or reducing) a term under a module loaded into the meta-interpreter before; and the `reducedTerm` message is sent back from the meta-interpreter along with the result of the simplification when it is complete.
- The `quit` message is sent to a meta-interpreter instance to request stopping the meta-interpreter; and the `bye` message is sent back as soon as the meta-interpreter is closed successfully.

Let us specify a system that has a server and the server is requested to increment a natural number. However, the increment of the natural number is not carried out by the server but by a meta-interpreter independently. For the sake of simplicity, we ignore handling errors. The system can be specified by the following system module:

```
mod SERVER is
  extending META-INTERPRETER .

  op Server : -> Cid .
  op aServer : -> Oid .
  op val :_ : Nat -> Attribute .

  vars O O' MI : Oid .
  var AS : AttributeSet .
  var T : Term .
  var RT : Type .
  vars N C : Nat .

  rl [loadMod]: < O : Server | AS >
    createdInterpreter(O, O', MI)
  => < O : Server | AS >
    insertModule(MI, O, upModule('NAT, true)) .
  crl [redTerm]: < O : Server | val : N, AS >
    insertedModule(O, MI)
  => < O : Server | val : N, AS >
    reduceTerm(MI, O, 'NAT, T)
  if T := '_+_[upTerm(N), upTerm(1)] .
  rl [quit]: < O : Server | val : N, AS >
    reducedTerm(O, MI, C, T, RT)
  => < O : Server | val : downTerm(T, 0), AS >
    quit(MI, O) .
  rl [bye]: < O : Server | AS > bye(O, MI)
  => < O : Server | AS > .
endm
```

where the four rewrite rules describe how the system interacts with meta-interpreters in regards to some messages exchanged to increment the natural number stored in the `val` attribute of a server. `upTerm` and `downTerm` functions [1] are used to move between the object and meta representation of a term, respectively. For example, the meta representation of 3, a natural number, is `'s_^3['0.Zero]` by using `upTerm`, while the object representation of `'s_^3['0.Zero]` is 3 by using `downTerm`. Modules and terms are first meta-presented by using the `upTerm` function and then sent to meta-interpreters as in the rewrite rules `loadMod` and `redTerm`, where `'_+_[upTerm(N), upTerm(1)]` is constructing the meta representation of the term (i.e., `N + 1`) that we want the meta-interpreter to reduce. We do not use `upTerm(N + 1)` to construct the meta representation of `N + 1` because `upTerm` will try to reduce it before sending it to the meta-interpreter, while we want the reduction done by the meta-interpreter. Terms returned by meta-interpreters are meta-representations that should be converted into object representations by using the `downTerm` function before being used as in the rewrite rule `quit`.

For example, we can request the system to increment a natural number, namely 3, stored in the `val` attribute of a server and get its result by the following command:

```
erewrite in SERVER :
  <> < aServer : Server | val : 3 >
  createInterpreter(interpreterManager, aServer,
      none) .
result Configuration:
  <> < aServer : Server | val : 4 >
```

where the `<>` symbol means communicating with external objects, namely meta-interpreters, `aServer` is an object identifier of the class `Server`, and `none` is an empty set of options. Rewriting with external objects is conducted by the external rewrite command **`erewrite`** in Maude.

REFERENCES

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, "Reflection, metalevel computation, and strategies," in *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science, M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, Eds. Springer, 2007, vol. 4350, pp. 419–458.