Introducing CafeOBJ specifications in the Full Maude database. Integrating CafeOBJ with the Constructor-based Inductive Theorem Prover—User Guide

Adrián Riesco

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

August 2013
(Revised October 1, 2013)

Abstract

CafeOBJ and Maude are sister languages of the OBJ language, and two of the most advanced formal specification languages for systems verification. Although both of them have a similar syntax and semantics, different usages and applications have been developed for them. Hence, a tool combining both languages would be very useful for exploiting the specific features of each of them.

We present here how to load CafeOBJ specifications into the Full Maude. This is achieved by defining the CafeOBJ syntax and the rules for parsing these modules. Since this is the basis of all the applications built on top of Full Maude, an extra advantage of this process is that we can easily extend other tools to work with CafeOBJ specifications. We also present example of such an integration, namely the Maude Declarative Debugger and the Constructor-based Inductive Theorem Prover (CITP). While the Declarative Debugger works with its usual interface, we have extended the CITP to work specifically with CafeOBJ specifications, so the underlying Maude infrastructure remains hidden.

Contents

1 Introducing CafeOBJ modules into the Full Maude database

We describe in this section how to use our tool to introduce CafeOBJ specifications into the Full Maude database. We will use this tool as base to develop the tools in the next sections.

1.1 Configuring the tool

The first step for configuring the tool is downloading the `parser.jar`, `cafeOBJ2maude.maude`, and `cafe2maude` files, available at `http://maude.sip.ucm.es/cafe/`. Then, the user must edit (with any text editor) the `cafe2maude` file by fixing the values of the variables:

`MAUDE`, which keeps the path to the Maude binary (available at `http://maude.cs.uiuc.edu/`).

`CAFE2MAUDE`, which keeps the path of to the `cafeOBJ2maude.maude` file described above.

`PARSER`, which keeps the path of to the `parser.jar` file described above.

Moreover, the file `full-maude.maude` must be available either in the same directory as Maude or in the current directory.

1.2 Using the tool

To start the tool it is enough to execute the `cafe2maude` script configured in the previous section by:

```
$> ./cafe2maude file_1 ... file_n
```

where `file_1 ... file_n` refer to the CafeOBJ files we want to introduce. The `cafe2maude` script modifies the CafeOBJ files by using parser.jar" to add the features required by Full Maude, and then creates a temporal file with these modified files. This file is the one loaded by Full Maude. The particularities of this translation can be found in Appendix A, while Appendix B describes the transformations performed by the Java file, and how to use it in an isolated way.

The modules introduced in this way will behave as standard Maude modules, and hence they can:

- Be imported, that is, they can be imported by other CafeOBJ and *Maude* specifications, for example we can introduce the Maude module:

  ```
  (fmod EXT-TEST is
    pr TEST .

    sort SFoo .
    subsort Foo < SFoo .
  endfm)
  ```

  However, the importations are restricted to those allowed by Maude. See Section A for details.

- Be used as source or target module in a view. Any combination with Maude theories or modules is allowed. For example, we can define the following view using the `TEST` module:

  ```
  view Test from TRIV to TEST is
    sort Elt to Foo .
  endv
  ```

- Import Maude modules, that is, CafeOBJ specifications can import Maude modules. For example, we can

```
mod! TEST2 {
  pr(EXT-TEST)
  [Value]
}
```

- Show the original CafeOBJ modules. We can use the command `(original CafeOBJ module MODULE-NAME .)` to print the CafeOBJ specification initially introduced into Full Maude. For example, the original `TEST` module is shown as follows:

```
Maude> (original CafeOBJ module TEST .)

mod! TEST{
  [Foo]
  op a : -> Foo {constr}
  op b : -> Foo {constr}
  op `{_`,_`} : Foo Foo -> Bool {comm}
  eq [l1] : {a,a} = true .
  eq [l2] : {b,b} = true .
  eq [l3] : {a,b} = false {metadata "different"} .
}
```

- Show the transformed modules. The Maude command `(show module MODULE-NAME .)` allows the user to visualize all the modules thus far introduced. For example, the following command is used to show the module `TEST2`:

```
Maude> (show module TEST2 .)
fmod TEST2 is
  including BOOL .
  protecting EXT-TEST .
  sorts Value .
endfm
```

- Show the views. Similarly to the command above, the command `(show view VIEW-NAME .)` allows the user to print the views. For example, we can use it to show the `Test` view as follows:

```
Maude> (show view Test .)
view Test from TRIV to TEST is
  sort Elt to Foo .
endv
```

- Use all the evaluation Maude commands, including `red`, `rew`, and `search`. For example, we can reduce a term in `TEST` as follows:

```
Maude> (red in TEST : {a, b} .)
reduce in TEST : {a,b}
result Bool : false
```

  Note that, since the `red` command is available and importing Maude modules is possible, we can also use the Maude model checker [1, Chapter 13].

2 Using the Constructor-based Inductive Theorem Prover

We present in this section how to use the Constructor-based Inductive Theorem Prover (CITP) with CafeOBJ specifications. A comprehensive explanation of the CITP can be found in `http://www.jaist.ac.jp/~danielmg/citp.html`.

The CITP is started by configuring and executing the `citp` script available `http://maude.sip.ucm.es/cafe/`, followed by the CafeOBJ modules to be used, as shown in the previous sections. Note that this script automatically applies the transformation described in Section 1, so the actual CafeOBJ files being used can be introduced. Although we could work with the transformed Maude versions obtained from the introduced modules, it is more convenient to allow CafeOBJ users to work with the tool in a transparent way. To allow that, the tool has been extended with two commands:

- `(cafeOBJ language .)`, which informs the tool that we want to work with CafeOBJ specifications, and hence no mentions to Maude code will be done.
- `(maude language .)`, which informs the tool that we want to forget about CafeOBJ syntax and use Maude syntax for commands and visualization.

By using the `(cafeOBJ language .)` option a number of commands in the CITP are modified from the interface point of view. We will consider in the following that this command has been used:

```
Maude> (cafeOBJ language .)
CafeOBJ selected as current specification language.
```

The revised commands are:

- `(goal ModuleName |- Equations/Rules/Memberships)`, which has been adapted to use CafeOBJ syntax and thus only accepts (possibly conditional) equations and transitions. Moreover, we can also use the standard syntactic sugar from CafeOBJ to avoid writing the variable sort once it has already been stated. For example, we can define a simple module `SPEC` (already modified by using the application shown in Section 1) for integers with the `0` constant, the successor and predecessor constructors, equations for representing the behavior of these operators combined, and a rule that decreases terms expressed by using successor:

  ```
  (mod! SPEC {
    [Zero < Int]
    op 0 : -> Zero {constr}
    ops s_ p_ : Int -> Int {constr}
    eq [eq1] : s p X:Int = X .
    eq [eq2] : p s X:Int = X .
    trans [tr1] : s(0) => 0 .
  })
  ```

  And now introduce a simple goal with two sentences. The first one will be an equation stating that the successor of any number is equal to applying the predecessor function once and the successor function twice to that number. The second one is a transition which indicates that we reach the integer `X` if we start from a term applying the successor function twice to it:

  ```
  Maude> (goal SPEC |- eq s(X:Int) = p(s(s(X))) ; trans s(s(X:Int)) => X ;)
  ============================ GOAL 1-1 ============================
  ```

```
< Module SPEC is concealed
...
end
,
  eq s X:Int
    = p s s X:Int .
  trans s s X:Int
    => X . >
unproved

INFO: an initial goal generated!
```

It is important to note that, since Maude does not allow importation of theories in protecting mode, this tool considers that the intended semantics is that of protecting. This is coherent with the approach presented in the previous section, where the importation modes were modified to including.

- `(set module on .)`, which asks the tool to show the current module in every proof step of the proof. If we activate this mode and then apply a step requiring the module to be shown, it will be displayed by using CafeOBJ syntax. For example, we can activate this mode

```
Maude> (set module on .)
INFO: Module will be displayed in goals
```

and introduce the goal shown above:

```
Maude> (goal SPEC |- eq s(X:Int) = p(s(s(X))) ; trans s(s(X:Int)) => X ;)
============================== GOAL 1-1 ==============================
<
mod! SPEC{
[Zero < Int]
op 0 : -> Zero {constr}
ops s_ p_ : Int -> Int {constr}
eq [eq1] : s p X:Int = X:Int .
eq [eq2] : p s X:Int = X:Int .
trans [tr1] : s 0 => 0 .
},
  eq s X:Int
    = p s s X:Int .
  trans s s X:Int
    => X:Int . >
unproved

INFO: an initial goal generated!
```

Note that now the original CafeOBJ module is displayed. Moreover, the prover introduces in some cases equations and rules to the module. These new statements will be also displayed following CafeOBJ syntax.

- `(discard critical pair .)`, which allows to discard critical pairs in equations and transitions. This command is dealt by using the commands `(equation .)`, to apply an equation, `(backward equation .)`, to apply an equation from right to left, `(transition .)`, to apply a transition, and `(backward transition .)`, to apply a transition from right to left.

- The rest of the rules, like the ones for applying rules or instantiating lemmas, have been internally modified to deal with CafeOBJ statements instead of Maude ones. However, its appearance is only modified is the command above is activated.
- It is worth noticing that the commands are also extended with error rules dealing with specific errors due to the interleaving of CafeOBJ and Maude specifications. For instance, if the CafeOBJ mode is active the user cannot introduce goals on Maude modules:

```
Maude> (goal MAUDE-MODULE |- eq 0 = 0 ;)
WARNING: the selected module is not a CafeOBJ specification.
```

References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[2] F. Durán, C. Rocha, and J. M. Álvarez. Towards a Maude formal environment. In G. Agha, J. Meseguer, and O. Danvy, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 329–351. Springer, 2011.

[3] K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.

A Translation

We present here the details and limitations of the translation from CafeOBJ specifications to Maude modules performed to introduce the former into the Full Maude database.

A.1 Built-in modules

CafeOBJ predefined modules are automatically mapped to the predefined Maude modules with the same name. As said in the introduction, one of the benefits of CafeOBJ and Maude being sister languages is that their share part of their syntax. This is quite convenient in this case, since it allows us to reuse the Maude code. In other case, specific modules defining the behavior of CafeOBJ modules should be used.

Note that some tools, as the Maude Formal Environment [2], does not allow the use of predefined modules, so the user cannot import them if he want to use those tools.

A.2 Types of modules

CafeOBJ allows the definition of two different kinds of module: modules with tight semantics, introduced by the keyword `mod!`, and modules with loose semantics, introduced by `mod*`. We translate the former to Maude modules (which will be functional modules if not rules are specified, and system modules otherwise) and the latter to Maude theories (which will be again translated as functional or system theories depending on the existence of rules). CafeOBJ views will be always translated as Maude views.

A.3 Parameterized modules and views

As explained above, modules defined with syntax `mod*` have loose semantics and are thus translated as Maude theories. Since the current version of Maude does not support parameterized theories, one of the main limitations of the translation is that *modules with loose semantics cannot be parameterized*. Regarding views, CafeOBJ syntax is richer than the one in Maude:

- Views can be declared "on the fly" when instantiating a parameterized module.

- The views assigned to each parameter during the instantiation can be identified by the parameter name, instead of introducing the views in the order given by the parameterized module.
- Moreover, in CafeOBJ the syntax `SortId.ParamId` is used for qualifying sorts, while in Maude the syntax `ParamId$SortId` is used.

Although the translation of each of these features is intuitive, is worth presenting a a small example. We start by translating the module `M1`, which requires the existence of a sort `A`:

```
(mod* M1 {
  [A]
})
```

This module is translated into a Maude theory as follows:

```
(fth M1 is
  sort A .
endfth)
```

Now, we define a parameterized CafeOBJ module `M2` with tight semantics, that receives two parameters fulfilling the theory `M1`, `X` and `Y`, and states that the sort `A` from `X` is a subsort of a new sort `B` declared in this module:

```
(mod! M2(X :: M1, Y :: M1){
  [A.X < B]
})
```

This module is translated into Maude as follows:

```
(fmod M2{X :: M1, Y :: M1} is
  sort B .
  subsort X$A < B .
endfm)
```

Now we define another module with tight semantics `M3` that will be used to declare a view `V` from `M1`, mapping the sort `A` from `M1` to the sort `C` declared in `M3`:

```
(mod! M3 {
  [C]
})

(view V from M1 to M3 {
  sort A -> C
})
```

These modules are translated into Maude in a straightforward way:

```
(fmod M3 is
  sort C .
endfm)

(view V from M1 to M3 is
  sort A to C .
endv)
```

It is important to note that the target module in a view *cannot be a theory*. This is due to the fact that in Maude this kind of view is still not instantiated and hence it requires the module where the instantiation takes place to be also parameterized and use some of these parameters in the instantiation, which is not required in CafeOBJ and would lead to illegal specifications. We explain at the end of the section a way to overcome this problem.

Finally, we define a module `M4` that instantiates `M2` by using the view `V` with the parameter `Y` and an on-the-fly view to instantiate `X`:

```
(mod! M4 {
  pr(M2(Y <= V, X <= view to M3 {sort A -> C}))
})
```

The translation into Maude first requires the creation of a new view `OTF-VIEW0`, where the prefix `OTF` stands for *on the fly* and the index is generated for each new view. Once this view has been introduced in the Maude database, the views in the instantiation are sorted to match the parameters in `M2`, and then the importation is accepted:

```
(view OTF-VIEW0 is
  sort A to C .
endv)

(fmod M4 is
  pr M1{OTF-VIEW0, V} .
endfm)
```

It is worth noting that, to allow a more flexible translation, we introduce an equivalent Maude module for each theory introduced into the database. If it is required to use this module instead of the theory to obtain a valid Maude specification, the system will use it and show a warning message to the user. This feature can be turned off with the command:

```
Maude> (strict translation on .)
The modules will be introduced as originally written.
```

and used again by typing:

```
Maude> (strict translation off .)
The translation will adapt CafeOBJ specifications to meet Maude requirements when
possible.
```

A.4 Importations

Maude importation features present some limitations:

- Theories can only be imported by other theories, and only in `including` mode.
- Modules can be imported by theories. Note that this importation indicates that, if the theory is used as the source of a view the target module must also import the module.
- Modules cannot import theories, only modules.

Some of the specifications thus obtained are not valid Maude specifications from the executability point of view, and hence commands like `red`, `rew`, or `search` cannot be used on them. Under these conditions, the importations allowed by our translation are, in addition to all the already available in Maude:

- We allow the importation of theories by other theories in any mode, since they will be just introduced into the theory assuming `including` mode.

- Modules can be imported by theories, but the conditions presented above must hold.
- importing a `mod*` within a `mod!` is allowed, and the tools will use the equations and rules in these theories in the standard way.

As explained above, we can use the commands `(strict translation on .)` and `(strict translation off .)` to force the system to accept the exact CafeOBJ specification.

A.5 Operators

Both CafeOBJ operators and behavioral operators are straightforwardly mapped to Maude operators. In the case of the CafeOBJ operator attributes `l-assoc` and `r-assoc` attributes, which are not available in Maude, they are translated by using `gather`.

A.6 Equations

CafeOBJ equations are directly translated into Maude equations.

A.7 Rules

`trans` and `btrans` (as well as `ctrans` and `cbtrans`) declarations are straightforwardly translated into Maude rules.

B Transforming CafeOBJ files to meet Full Maude requirements

Hence, in order to make CafeOBJ specifications [3] compatible with Full Maude it is necessary to perform some modifications. Namely, we have to:

- Enclose every module into parentheses.
- Introduce the symbol ‘ before the scape characters, such as parentheses, commas, or curly braces, when defining operators.
- Transform CafeOBJ comments into Maude comments.

Moreover, we consider that the `metadata` attribute might be quite useful in some cases, and it is used by several Maude tools, like the prover in Section 2. Thus, it is possible to write metadata information following equations and transitions as CafeOBJ comments (hence being valid CafeOBJ specifications). These comments must be introduced later into the statements to work.

We have implemented a Java application that, given a CafeOBJ specification (possibly with commented metadata annotations), generates a new file performing the modifications explained above. In this way, if we have a file `test.cafe` with the following CafeOBJ specification

```
mod! TEST {
 [Foo]
 op a : -> Foo
 op b : -> Foo
 op {_,_} : Foo Foo -> Bool {comm}
 --> We just check whether the values are equal
 eq [l1] : {a, a} = true .
 eq [l2] : {b, b} = true .
 eq [l3] : {a, b} = false . **> {metadata "different"}
}
```

We can download the application `parser.jar`, available at `http://maude.sip.ucm.es/cafe/`, and use the command

```
$ java -jar parser.jar test.cafe test-trans.cafe
```

where `test-trans.cafe` is the name of the target file to be created, we obtain the following transformed module:

```
(mod! TEST {
 [Foo]
 op a : -> Foo
 op b : -> Foo
 op `{_`,_`} : Foo Foo -> Bool {comm}
 --- We just check whether the values are equal
 eq [l1] : {a, a} = true .
 eq [l2] : {b, b} = true .
 eq [l3] : {a, b} = false {metadata "different"} .
}
)
```

Note that the module is now enclosed in parentheses, the scape characters in the third operator declaration are now preceded by `, while the rest of scape characters remain unchanged, the comment has been transformed into a Maude comment and the metadata attribute has been introduced into the equation.

Once the files we want to load have been transformed with this application, we have to load Full Maude and `cafeOBJ2maude.maude`, available at `http://maude.sip.ucm.es/cafe/`:

```
Maude> loop cafe2maude-init .

     CafeOBJ2Maude 1.0 started.
 CafeOBJ specifications can be introduced now into the Full Maude database.
```

This command starts an input/output loop where the modules previously obtained can be introduced. Now we can use the standard Maude commands to load modules:

```
Maude> in test-trans.cafe
Introduced Tight Cafe Module: TEST
```