An Integration of CafeOBJ into Full Maude

Adrián Riesco(✉)

Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain
ariesco@fdi.ucm.es

Abstract. We present in this paper an integration of CafeOBJ into Full Maude. We have developed a grammar to parse any CafeOBJ specification, an intermediate language to store it, and a translation from this representation into Maude specifications. This integration enhances CafeOBJ functionality in many ways: our intermediate representation has been developed mirroring Maude metalevel, and hence it allows CafeOBJ users to analyze, modify, and execute them; CafeOBJ specifications can use Maude commands, including the LTL model checker; other Full Maude tools can be straightforwardly combined with this extension; and we provide an alternative implementation for CafeOBJ that can be easily modified and extended. We present here the ideas for parsing and translating CafeOBJ specifications, and illustrate with examples the features listed above.

Keywords: CafeOBJ · Full Maude · Integration · Metalevel

1 Introduction

CafeOBJ [9] is a language for writing formal specifications of models for wide varieties of software and systems, and verifying properties of them. CafeOBJ implements equational logic by rewriting and can be used as a powerful interactive theorem proving system. Specifiers can write proof scores [10] also in CafeOBJ and perform proofs by executing these proof scores. CafeOBJ provides several features to ease the specification of systems. These features include a flexible mix-fix syntax, powerful and clear typing system with ordered sorts, parameterized modules and views for instantiating the parameters, module expressions, operators for defining terms, and equations for defining the (possibly conditional) equalities between terms and (possibly conditional) transitions for specifying how a system evolves, among others. However, only a subset of the CafeOBJ specifications, the equational part, is executable, where the operational semantics is given by a conditional order-sorted term rewriting system.

Maude modules are executable rewriting logic specifications. Maude functional modules [1, Chap. 4] are executable membership equational specifications

Research partially supported by Japanese project Kakenhi 23220002, MICINN Spanish project *StrongSoft* (TIN2012-39391-C04-04), and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC1465).

© Springer International Publishing Switzerland 2014
S. Escobar (Ed.): WRLA 2014, LNCS 8663, pp. 230–246, 2014.
DOI: 10.1007/978-3-319-12904-4_13

that allow the definition of sorts; subsort relations between sorts; operators for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative or commutative, for example; memberships asserting that a term has a sort; and equations identifying terms. Both memberships and equations can be conditional. Maude system modules [1, Chap. 6] are executable rewrite theories. A system module can contain all the declarations of a functional module and, in addition, declarations for rules and conditional rules. An important feature of rewriting logic is that it is reflective, that is, it can be faithfully interpreted in terms of itself. This feature is efficiently implemented in Maude by means of the `META-LEVEL` module [1, Chapt. 14], which allows us to use Maude modules and terms as usual data.

Full Maude [1, Part II] is an extension of Maude written in Maude itself. Full Maude provides an even more powerful module algebra than the one available in Core Maude, features for parsing and printing Maude modules, and an explicit module database. This database, combined with the meta-level features explained above, allows us to introduce, remove, modify, and analyze the modules introduced by the user. Moreover, it is also possible to change the syntax of existing features and add new kinds of modules and commands. Full Maude is built on top of the Loop Mode [1, Chapt. 17], which provides a mechanism to read the modules and commands introduced by the user enclosed in parentheses, and to show him the results generated by these commands. For these reasons, Full Maude has been traditionally used as a basis for further extensions, either for extra syntactic constructs, like the support for Real-Time modules [14], or for new commands, like the `narrowing` search currently available for symbolic execution [2, Chap. 16].

We present in this paper an extension of Full Maude to parse CafeOBJ modules. The advantages obtained by using this tool, publicly available at http://maude.sip.ucm.es/cafe/, are:

- Maude modules can be imported by CafeOBJ modules, and vice versa. The former is specially useful because Maude provides the predefined modules `SATISFACTION`, `LTL-SIMPLIFIER`, and `MODEL-CHECKER` [1, Chapt. 12], which allow the user to define and prove LTL properties on CafeOBJ specifications. We can also use the Loop Mode [1, Chapt. 17] to develop interactive tools. Moreover, we have defined an intermediate representation of CafeOBJ specifications that mirrors Maude metalevel, and have included functions to execute terms using these modules. That is, CafeOBJ modules can use CafeOBJ modules and terms as standard data, just as several Maude applications have been designed during the last years.
- Maude commands can be used on CafeOBJ specifications. This allows the user to use, among others, the `rew` command to apply transitions (and normalization via equations) to CafeOBJ terms (which cannot be done in the current release of CafeOBJ) [1, Chapt. 6]; the `search` command to perform searches to check invariants [1, Chapt. 12]; or the `narrowing` command for symbolic execution [2, Chapt. 16].

- It provides a new implementation of CafeOBJ. Our interface parses any CafeOBJ module and accepts `open-close` environments, required to execute proof scores. We also process behavioral specifications, although the current version of the tool does not distinguish between behavioral and non-behavioral statements in the translation.

  Moreover, this new implementation is more powerful in the sense that any CafeOBJ programmer can add new syntax and commands. Although this extension would require modifying the Maude code used by the interface, it is so similar to CafeOBJ code that it can be easily understood. Actually, the code has been designed with this feature in mind, so the syntax and parsing modules are carefully distinguished and documented.

  As an example of the syntax that can be added to CafeOBJ specifications, our parser allows the user to use matching and rewrite conditions, as well as using the `nonexec` and `metadata` attributes in equations and transitions. Some of these features are available in the latest release of CafeOBJ, while others are only supported by our implementation.
- It allows an easy integration of CafeOBJ specifications with any tool implemented on top of Full Maude. We have currently integrated the Maude Declarative Debugger and Test-case Generator [16] and the Constructor-based Inductive Theorem Prover [11]. Our goal when integrating these tools was to provide a minimum framework where CafeOBJ functions can be tested, fixed when a wrong behavior is found, and proved correct with respect to some properties, once we have confidence in the soundness of the implementation. However, many other interesting tools can be integrated using our approach.
- Finally, we provide a script to connect CafeOBJ with Full Maude in a transparent way. We have implemented a Java class that transforms the source code to meet the format required by Full Maude, which includes enclosing the modules in parentheses, adding the ‘ to escape characters such as `[`, `]`, or `,`, and removing CafeOBJ comments, among others. In this way, it is not necessary to modify the original CafeOBJ specifications to use the interface.

The rest of the paper is organized as follows: Sect. 2 briefly introduces the related work, while Sect. 3 presents the basic notions used throughout the paper. Section 4 describes the parsing and translation process. Section 5 illustrates how to use the tool. Finally, Sect. 6 presents the concluding remarks and outlines some lines of future work.

2 Related Work

The most similar examples to the present work are Full Maude itself [1, Part II], Real-Time Maude [14], and the Maude Strategy Language [7]. The former defines a complete syntax for Maude, extends it with support for object-oriented modules, and provides commands to execute them. Similarly, Real-Time Maude defines real-time modules and timed commands to execute them, while the Strategy Language extends Maude modules with syntax for defining execution strategies, as well as rewrite commands using these strategies. Our work follows the

same steps: it requires to define the syntax of our modules and commands, parse them, translate them into Maude (in Full Maude this is only the case for object-oriented modules, since standard modules do not require translation), and execute the commands. Nonetheless, we take advantage of many features developed for Full Maude and reused later [5], which greatly ease the parsing task.

Besides these tools, Maude has been used as a semantical framework to specify the semantics of several languages, such as LOTOS [17], CCS [17], or C [8]. These researches, as well as several other efforts to describe a methodology to represent the semantics of programming languages in Maude, led to the *rewriting logic semantics project* [12], which presents a comprehensive compilation of these works.

Another translation from CafeOBJ to Maude can be found in [18]. There, the authors translate a subset of CafeOBJ specifications (more specifically, specifications of state machines standing for asynchronous distributed systems) into Maude to perform model checking. Although they follow an approach similar to the one in the current paper, it is focused in just one kind of specification, and hence it lacks scalability.

3 Preliminaries

We present in this section some basic notions required throughout the rest of the paper. First, we describe CafeOBJ and Maude by means of an example. Then, we give some details about the Maude metalevel and Full Maude.

3.1 CafeOBJ and Maude

CafeOBJ (on the lefthand side) can define modules with loose semantics by using the syntax `mod*`. For example, we can define a module `ELT` requiring the existence of a sort `Elt` and an element of this sort, called `mt`, which is a constructor. This kind of behavior is specified in Maude (on the righthand side) as a theory:

```
mod* ELT {
 [Elt]
 op mt : -> Elt {constr}
}
```

```
fth ELT is
 sort Elt .
 op mt : -> Elt [ctor] .
endfth
```

We can use this module to define a parameterized module with tight semantics, with syntax `mod!`. The module `LIST` below indicates that it receives a parameter `X` fulfilling the requirements stated by `ELT`. This module first defines the sort `List` for lists. Similarly, we define a parameterized system module `LIST` in Maude with syntax `mod`:

```
mod! LIST(X :: ELT) {
  [List]
```

```
mod LIST{X :: ELT} is
 sort List .
```

The constructors are defined, as shown above, with the keyword `op` and the `constr` attribute. In this case the constructors are `nil` for empty lists and the juxtaposition operator `__` for placing an element of sort `Elt` in front of a list. Note the different syntax for the sort `Elt`, qualified by the parameter `X`:

```
op nil : -> List {constr}
op __ : Elt.X List -> List {constr}
```

```
op nil : -> List [ctor] .
op __ : X$Elt List -> List [ctor] .
```

We can also define functions for lists. For example, composition of lists is defined by distinguishing constructors on the first argument. Note that both CafeOBJ and Maude follow the same syntax, although CafeOBJ allows some extra syntactic sugar, including just-once on-the-fly declaration of variables:

```
var E : Elt.X      var L : List
op _@_ : List List -> List
eq [c1] : nil @ L = L .
eq [c2] : (E L) @ L':List =
          E (L @ L') .
```

```
var E : X$Elt .  var L : List .
op _@_ : List List -> List .
eq [c1] : nil @ L = L .
eq [c2] : (E L) @ L':List =
          E (L @ L':List) .
```

Similarly, we can define the `reverse` function. This function uses the constant `mt` from module `ELT` as the reverse of the empty list,[1] while the reverse for bigger lists is defined as usual by using the composition above:

```
op reverse : List -> List
eq [r1] : reverse(nil) = mt nil .
eq [r2] : reverse(E L) =
          reverse(L) @ (E nil) .
```

```
op reverse : List -> List .
eq [r1] : reverse(nil) = mt nil .
eq [r2] : reverse(E L) =
          reverse(L) @ (E nil) .
```

We can also define non-deterministic transitions. For example, we can combine two lists by using the commutative operator `mix` and two transitions to indicate that the next element is the first one of any of the lists (thanks to the matching modulo commutativity):

```
op mix : List List -> List {comm}
trans [m1] : mix(nil, L) => L .
trans [m2] : mix(E L, L')
          => E mix(L, L') .
}
```

```
op mix : List List -> List [comm] .
rl [m1] : mix(nil, L) => L .
rl [m2] : mix(E L, L')
       => E mix(L, L') .
endm
```

Finally, in CafeOBJ we can use an on-the-fly view to instantiate `LIST` with natural numbers:

```
mod! NAT-LIST {
  pr(LIST(view to NAT {sort Elt -> Nat, op mt -> 0}))
}
```

On the other hand, we need to define an explicit view in Maude, and then use this view to instantiate the module:

[1] This is a wrong definition that will be detected and fixed in Sect. 5.3.

```
view Nat from Elt to NAT is
 sort Elt to Nat .
 op mt to 0 .
endv

mod NAT-LIST is
 pr LIST{Nat} .
endm
```

3.2 Maude Metalevel and Full Maude

Exploiting the fact that rewriting logic is reflective [3], an important feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [1, Chapt. 14], a characteristic that allows many advanced metaprogramming and metalanguage applications. In this work, we take advantage of this feature to parse, store, transform, and execute CafeOBJ modules.

Full Maude [1, Part II] is an extension of Maude written in Maude itself. Full Maude is built on top of the `LOOP-MODE` module [1, Chapt. 17]. This module allows input/output interaction by means of the `[_,_,_]` operator, which builds terms of sort `System` and where the first argument corresponds to the input introduced by the user, which must be enclosed in parentheses to be recognized; the second one is a term of sort `State` that can be defined by the user for each application; and the third one the output shown to the user.

In Full Maude this `State` is defined by using a class `Database`, which has an attribute `db` standing for the Full Maude database. It also has attributes for the current `input`, the `output` not processed yet, and the `default` module. Essentially, the Loop Mode transforms the data introduced by the user into a list of quoted identifiers; this list is then meta-parsed by Full Maude by using the `GRAMMAR` module, which includes the syntax for modules and commands. If this parsing is successful, then the term thus obtained is placed in the `input` attribute. Different inputs are treated by using rules: modules and views are processed to check whether they fulfill the semantic constraints required by Maude, and then introduced into the database, while commands are executed by using this database. The results must be placed in the `output` attribute; a rule will move this data to the third component of the system.

Hence, our aims in this paper are to extend `GRAMMAR` to include CafeOBJ syntax, process the new terms obtained from the parsing, and define commands (and the appropriate rules) to deal with these new features.

4 Introducing CafeOBJ Modules into the Full Maude Database

We present in this section the basic ideas to introduce CafeOBJ modules into the Full Maude database. First, we describe how CafeOBJ modules are parsed. Then we show how the obtained modules can be translated into Maude and used by other tools implemented in Full Maude.

4.1 Parsing CafeOBJ Modules and Commands

As explained in the previous section, in order to parse CafeOBJ modules we have to define its syntax, which will be used by Full Maude to create a term that will be processed to obtain the actual module. We use the metarepresentation of this module to extend the `GRAMMAR` metamodule from Full Maude, providing the metamodule `CafeGRAMMAR`. It can be used to parse both Maude and CafeOBJ modules and commands.

Basically, the syntax follows the CafeOBJ grammar in [13], although we have extended it with some features that will be available in the next release of CafeOBJ, such as the `nonexec` attribute or matching conditions. Following the standard approach, we define a sort for each syntactic category in the grammar, and operator declarations for each production rule. In this way, we specify a module `CafeMETA-SIGN` where this information is contained. For example, the sort `@CafeTransDecl@`[2] stands for the definition of transitions in CafeOBJ syntax:

```
op trans_=>_. : @CafeBubble@ @CafeBubble@ -> @CafeEqDecl@ [ctor] .
op ctrans_=>_if_. : @CafeBubble@ @CafeBubble@ @CafeBubble@
                    -> @CafeTransDecl@ [ctor] .
op ctrns_=>_if_. : @CafeBubble@ @CafeBubble@ @CafeBubble@
                   -> @CafeTransDecl@ [ctor] .
```

Note that we use a special sort `@CafeBubble@` to encapsulate terms that can take any form. Basically, a bubble is any list of quoted identifiers, which must be later parsed to obtain a valid term in the current module.

These declarations, as well as the rest of declarations for the statements available in a CafeOBJ module, are defined as a subsort of a `@CafeDeclList@`, which are composed by means of a juxtaposition operator:

```
subsorts @CafeImportDecl@ ... @CafeTransDecl@ < @CafeDeclList@ .
op __ : @CafeDeclList@ @CafeDeclList@ -> @CafeDeclList@ [assoc] .
```

For example, the transition `m1` from Sect. 3 would be parsed as:

```
'trans_=>_.['CafeBubble['__[''[.Qid, ''m1.Qid, ''], ':, ''mix.Qid,
    ''(.Qid, ''nil.Qid, '',, ''L.Qid, '').Qid]], 'CafeBubble[''L.Qid]]
```

Note that the label is included in the bubble for the lefthand side; it must be extracted before processing this side (analogously, attributes might appear in the bubble for the righthand side). This term must be now parsed again in order to check whether it fulfills the semantics constraints, e.g., the terms only use variables previously defined, they are bound either in the lefthand side or in a matching condition, and terms are built using existing operators. This second phase returns, when the module is correct, a term of sort `CafeModule`:

```
op mod*_{__[_]____} : CafeHeader CafeImportList HiddenSortDecl SortSet
                      CafeSubsortDeclSet CafeOpDeclSet CafeEqSet
                      CafeTransSet -> CafeModule [ctor] .
```

[2] We follow the Full Maude convention and enclose sorts for parsing in `@`.

Our definition of CafeOBJ modules uses the sorts `Qid`, `Term`, and `Condition` from Maude metalevel to define the sorts used here. For example, transitions are declared as follows:

```
op trans_=>_{_}. : Term Term CafeAttrSet -> CafeTrans [ctor] .
op ctrans_=>_if_{_}. : Term Term Condition CafeAttrSet -> CafeTrans [ctor] .
```

In this way, the transition `m1` is represented as:

```
trans 'mix['nil.List, 'L:List] => 'L:List {label:('m1)} .
```

Once the final module has been obtained, it is stored in a database, which is just a partial function from quoted identifiers (of sort `Qid`) to `CafeModule`. This modules can be retrieved, modified, executed, and stored again, as we will see in Sect. 5. Note that the current version of the tool does not support metasyntax for views; they are just introduced as Maude views.

Regarding commands, we provide the syntax for `open...close` environments, which combine operator declarations (mainly constants) and equation definitions with `red` commands to define proof scores [10], and specific commands for dealing with CafeOBJ modules. In this case we create an on-the-fly module where the reductions take place.

4.2 Translating the Modules

Taking advantage of the similarities between the syntax and the semantics of CafeOBJ and Maude, most of the transformations performed by our tool are straightforward. Both languages have modules with loose semantics (called *theories* in Maude), modules with tight semantics, parameterized modules, views to instantiate these modules, equations, and transitions (*rules* in Maude) as main features. From the Maude point of view there are some features that cannot be translated into CafeOBJ, being the main one the membership axioms stating the members of a sort, because Maude implements membership equational logic while the CafeOBJ type system is based on order sorted algebra. However, the differences in this case are not important because we are interested in the translation from CafeOBJ to Maude.

There are two important features in CafeOBJ that cannot be translated into Maude. Both of them are related to the importation of modules with loose semantics: (i) these modules can be imported by any module, while in Maude they can only be imported by other theories, and (ii) these modules can be imported in any mode (being the modes `protecting`, indicating that no junk and no confusion is added to the sorts; `extending`, denoting that no confusion is allowed; and `including`, indicating that there are no restrictions, see [1, Chapt. 8] for details), while Maude theories can only be imported in `including` mode. We have dealt with these restrictions in a conservative way. First, we translate these modules, that should be Maude theories, as modules (i.e., they have tight semantics), and a warning message is shown. This change is harmless if our aim is to execute them or to use any of the tools currently integrated (the declarative debugger

and the CITP), but has two disadvantages: (a) it might fail later, if this module is used as the target of a view, and (b) other tools, not integrated yet, might distinguish between the different kinds of modules. Similarly, we always translate the importation modes for these modules as `including`, which is also fine in our case (the tools integrated thus far use flattened modules) but might produce problems with other tools. The user can force the tool to translate the modules without modifications with the `(strict translation on.)` command.

There are also some other complex features that require a non-straightforward translation. More specifically, the CafeOBJ syntax for views is much more flexible than the one used by Maude: they can be defined on-the-fly and can be used in an order different from the one specified in the parameterized module by using the parameter name. The former is solved by creating explicit views with fresh view identifiers, while the latter requires to manipulate the parameterized module from the database to reorder the views.

Basically, our implementation defines a function `cafe2maude`, which takes a `CafeModule` and returns a Maude `Module`:

```
op cafe2maude : CafeModule -> Module .
```

It uses auxiliary functions to translate each element in a CafeOBJ module. For instance, transitions are translated into rules as follows:

```
op cafe2maude : CafeTrans -> Rule .
eq cafe2maude(trans T => T' {AtS} .) = rl T => T' [cafe2maude*(AtS)] . .
eq cafe2maude(ctrans T => T' if C {AtS} .) = crl T => T'
                                              if C [cafe2maude*(AtS)] . .
```

where `cafe2maude*` is an auxiliary function that translates the attributes.

As explained in Sect. 3.2, the connection between the Loop Mode and the behavior of the tool is implemented by rules. We have defined a new class `CafeDatabase`, subclass of `Database`, to take care of the translation and the new commands:

```
sort CafeDatabaseClass .
subsort CafeDatabaseClass < DatabaseClass .
op CafeDatabase : -> CafeDatabaseClass [ctor] .
```

This class defines two new attributes: `strict`, which indicates whether the translation is strict or not, and `cafeDB`, which contains the CafeOBJ database:

```
op strict :_ : Bool -> Attribute [ctor] .
op cafeDB :_ : CafeDB -> Attribute [ctor] .
```

4.3 Combining CafeOBJ and Other Full Maude Tools

Using the modules described in the previous sections, it is easy to modify any tool built in Full Maude for Maude specifications and make it work with CafeOBJ modules, given that they follow two standard principles:[3]

[3] Note that these changes will allow us to execute the tools. However, some theoretical considerations may be required to prove that this execution is correct.

– They use a module extending `GRAMMAR` to parse their modules/commands. In this case, it is enough to extend `CafeGRAMMAR` instead, and CafeOBJ modules will be parsed.
– They define a subclass of `Database` to process their modules/commands. We have to modify this definition to extend `CafeDatabase`. It is also required to initialize the attributes `strict` and `cafeDB`, so they can be used later.

To test the benefits of this approach we have already worked with the Maude declarative debugger and test-case generator [16] and the Constructor-based Inductive Theorem Prover (CITP) [11]. Note that a potential problem of any integration is that the output provided by the tool refers to the transformed Maude code. Although this might be fine in some cases (e.g. the debugger refers to the label of the wrong statement, so it is safe to use it, see Sect. 5.3 for details), in some others it is interesting to refer to the original CafeOBJ module or just use commands which are specifically defined for CafeOBJ users. In this case, some extra changes are required, as shown in the next section for the CITP.

5 Connecting CafeOBJ and Maude

We present in this section how to use the most important features of our implementation. We first show how to use the metalevel representation of CafeOBJ. Then, we describe the basic commands provided in the interface and how to use the Maude Declarative Debugger and the Constructor-based Inductive Theorem Prover. All the modules, scripts, and examples shown here are available at http://maude.sip.ucm.es/cafe/.

5.1 Metaprogramming in CafeOBJ

We provide in the `META-CAFE-SYNTAX` module the syntax for CafeOBJ modules. It follows the syntax in the predefined module `META-LEVEL` for Maude modules, but uses specific syntax to follow CafeOBJ conventions. These modules are retrieved from and inserted into the database with the functions `getTopModule` and `setTopModule`. Note that, since these modules are stored in a specific attribute of the `CafeDatabase` class, specifications using the database are not completely transparent from Maude syntax:

```
op getTopModule : CafeDB Qid ~> CafeModule .
op setTopModule : CafeDB Qid CafeModule -> CafeDB .
```

Finally, these modules can be modified and executed by using the functions in `CAFE-META-LEVEL`. It includes functions for accessing the different components of a module, update them, and for executing terms in a given module. The current version of the tool provides the functions `metaReduce`, for applying equations until a normal form is reached; `metaRewrite`, for applying transitions given a bound in the number of transitions applied; and `metaFrewrite`, for fair application of transitions given a bound in the number of transitions applied and the maximum number of rewrites at each entitled position on each traversal of a subject term (see [1, Chapt. 14] for details):

```
op metaReduce : Qid Term CafeDB Database -> ResultPair .
op metaRewrite : Qid Term Bound CafeDB Database -> ResultPair .
op metaFrewrite : Qid Term Bound Nat CafeDB Database -> ResultPair .
```

Note that these functions require the Maude database, since they might import some Maude modules. They are implemented by building the corresponding flat Maude module and then using the appropriate built-in Maude functions.

For example, we could define a function `getCommOps` extracting the commutative operators from a CafeOBJ module by using an auxiliary function `filterCommOps` that keeps the commutative operators from a set:

```
op getCommOps : CafeModule -> CafeOpDeclSet
eq getCommOps(CM) = filterCommOps(getOps(CM)) .
op filterCommOps : CafeOpDeclSet -> CafeOpDeclSet
eq filterCommOps(none) = none .
eq filterCommOps(COD CODS) = if isComm?(COD) then COD
                             else none fi filterCommOps(CODS) .
```

where `isComm?` is an auxiliary function that checks whether an operator is commutative. Note that we allow operators with both the `op` definition and the `pred` keyword. This function uses another auxiliary function `containsComm?` which just traverses the attributes looking for `comm`:

```
pred isComm? : CafeOpDecl
eq isComm?(op Q : TyL -> Ty {AtS}) = containsComm?(AtS) .
eq isComm?(pred Q : TyL {AtS}) = containsComm?(AtS) .
pred containsComm? : CafeAttrSet
eq containsComm?(none) = false .
eq containsComm?(A AtS) = A == comm or containsComm?(AtS) .
```

5.2 Basic Commands

Once the files in the webpage have been downloaded and the paths have been configured, and assuming the modules above are saved in a file called `wrla.cafe`, we can start the tool by typing:

```
$ ./cafe2maude wrla.cafe
```

The `cafe2maude` script creates a temporary file generated by a Java application. This file contains the original CafeOBJ modules modified in order to be accepted by Full Maude (e.g. adding the parentheses enclosing modules and views, removing CafeOBJ comments, and adding the ‘ character to the escape characters such as { or }). Once the script is executed, the modules are introduced into the Full Maude database and we can use any Maude command on them. For example, the `rew` command uses transitions to evaluate terms. Note that this command, as well as the one below, is not available in CafeOBJ:

```
Maude> (rew mix(1 3 nil, 2 4 nil) .)
result List : 1 2 3 4 nil
```

We can also use symbolic search to start with terms with variables and look for substitutions that fulfill the conditions imposed by the search. For example, we can look for the term required in the `mix` operator to obtain the result from the `rew` command:

```
Maude> (search [1] mix(L:List, 2 4 nil) ~>! 1 2 3 4 nil .)
Solution 1
L:List --> 1 3 nil
No more solutions.
```

where the `!` option indicates that we are looking for *final* terms and `>!` distinguishes the symbolic search from the standard one, performed with `=>!`. In this case we obtain the substitution `L:List --> 1 3 nil`, indicating that we needed this list to obtain the result.

Besides using Maude commands, we can also work with CafeOBJ specifications. For example, we can see the original module and execute proof scores. Basically, proof scores are scripts defining an inductive proof, where constants can be declared by means of operators and hypothesis by using equations. The base and the inductive steps are proved by using the `red` command. For example, we can prove the associativity of the `_+_` function as follows:

```
open NAT + BOOL
 ops i j k : -> Nat
 red (0 + j) + k == 0 + (j + k) .        -- base step
 eq (i + j) + k = i + (j + k) .          -- induction hypothesis
 red (s(i) + j) + k == s(i) + (j + k) . -- inductive step
close
```

Once we load the file with this `open-close` environment, Maude executes the `red` commands and provides the following result:

```
Processing open-close environment:
reduce(0 + j)+ k == 0 + j + k .
Result: true : Bool
reduce(s i + j)+ k == s i + j + k .
Result: true : Bool
```

5.3 Using the Declarative Debugger and Test-Case Generator

To start this tool it is enough to download the script `cdd`, configure the paths, and execute it with the files we want to test and debug. Then, we can use all the commands described in http://maude.sip.ucm.es/debugging/ to test and debug our CafeOBJ modules. For example, we can test the `reverse` function by using the so called *function coverage* criterium, which generates ground test cases that must use all the equations defined for `reverse` (`r1` and `r2`) in all the calls (the single call to this function is located in `r2`). This is done by using:

```
Maude> (function coverage .)
Function Coverage selected
Maude> (test in NAT-LIST : reverse .)
1 test cases have to be checked by the user:
  1. The term reverse(0 0 nil) has been reduced to 0 0 0 nil
All calls were covered.
```

That is, the call `reverse(0 0 nil)` uses both `r1` and `r2` for the recursive call (`r2` for the first call and `r1` for the second one). Note that the result of this call is unexpected, because it should also be `0 0 nil`. Hence, this function is buggy and must be debugged. We can do it by typing:

```
Maude> (invoke debugger with user test case 1 .)
Declarative debugging of wrong answers started.
```

This command starts the declarative debugger. Declarative debuggers find bugs in programs by asking questions to the user, that must answer `yes` or `no` (check the webpage above for more possible answers) until the bug is found. Hence, the debugger presents the following question:

```
Is this reduction (associated with the equation r2) correct?
reverse(0 nil) -> 0 0 nil
Maude> (no .)
```

This result is erroneous for the same reasons explained above, so the user answers `no` and the debugging session continues with the following questions:

```
Is this reduction (associated with the equation com2) correct?
(0 nil) @ 0 nil -> 0 0 nil
Maude> (yes .)
Is this reduction (associated with the equation r1) correct?
reverse(nil) -> 0 nil
Maude> (no .)
```

We answer `yes` for a correct composition but `no` for another application of `reverse`. With this information the debugger is able to find the bug:

```
The buggy node is: reverse(nil) -> 0 nil
with the associated equation: r1
```

In fact, the equation `r1` should return just `nil`. The questions asked during the session correspond to the nodes of a tree representing the wrong computation. This tree, which might be useful to the user to check the relations between the calls, can also be shown.

Finally, it is also possible to use a property and a correct module to test the functions. For example, we can define in the module `PROP-LIST` the property `prop` stating that applying `reverse` twice returns the same list, while in `CORRECT-PROP-LIST` we state that this property must be always `true`:

```
mod! PROP-LIST {
  pr(NAT-LIST)
  op prop : List -> Bool
  eq [p1] : prop(L:List) =
      reverse(reverse(L:List)) == L:List .
}
```

```
mod!CORRECT-PROP-LIST {
  pr(NAT-LIST)
  op prop : List -> Bool .
  eq prop(L) = true .
}
```

Now we can set the correct module and generate test cases:

```
Maude> (correct test module CORRECT-PROP-LIST .)
CORRECT-PROP-LIST selected as correct module for testing.
Maude> (test in PROP-LIST : prop .)
10 test cases are incorrect with respect to the correct module.
```

Once the test cases have been generated, they can be displayed and debugged as shown above.

5.4 Using the Constructor-Based Inductive Theorem Prover

We have extended the CITP to work with CafeOBJ-like commands, hence obtaining a tool fully customized for CafeOBJ. This has been done by adding an extra attribute `language` to the tool, which allows us to distinguish between interfaces, while the underlying modules dealing with proofs are left unmodified.

The CITP allows the user to prove properties on CafeOBJ specifications. It is started by the `citp` script. Since we want to prove properties on CafeOBJ specifications, we have to indicate it with a specific command, which sets the `language` attribute explained above to `cafeOBJ`, hence modifying the syntax and the display options to work with CafeOBJ specifications:

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

Now we can introduce goals, which are depicted as equations or transitions. For example, we can prove the associativity of list composition, using on-the-fly declaration of variables from CafeOBJ, by typing:

```
Maude> (goal NAT-LIST |- eq L1:List @ (L2:List @ L3:List) =
                           (L1 @ L2) @ L3 ;)
============================ GOAL 1-1 ============================
< Module NAT-LIST is concealed ... end,
 eq L1:List @(L2:List @ L3:List) = (L1:List @ L2:List)@ L3:List . >
unproved
INFO: an initial goal generated!
```

This goal can be easily proved by using induction on `L1` and then applying the default tactic with the `auto` command:

```
Maude> (set ind on L1:List .)
INFO: Induction will be conducted on L1:List
Maude> (auto .)
INFO: Goal 1-1 was successfully proved by applying tactic: SI CA CS TC IP
INFO: PROOF COMPLETED
```

It is also possible to state goals involving transitions. For example, we can define the following trivial goal, which just uses the commutativity attribute:

```
Maude> (goal NAT-LIST |- trans mix(L:List, nil) => L ;)
============================ GOAL 1-1 ============================
< Module NAT-LIST is concealed ... end,
 trans mix(L:List,nil) => L:List . >
unproved
INFO: an initial goal generated!
```

Note that CafeOBJ syntax is used for both the goal and the displayed information. This simple goal can be discarded by just using `auto`:

```
Maude> (auto .)
INFO: Goal 1-1 was successfully proved by applying tactic: SI CA CS TC IP
INFO: PROOF COMPLETED
```

Much more information on the CITP, including several other commands, all of them now customized for CafeOBJ specifications, is described at http://www.jaist.ac.jp/~danielmg/citp.html.

6 Concluding Remarks and Ongoing Work

We have presented in this paper a tool to introduce CafeOBJ specifications into the Full Maude database. This tool allows us to use Maude modules and commands with CafeOBJ specifications, provides an implementation of a CafeOBJ metalevel, and eases the task of connecting CafeOBJ specifications with tools implemented on top of Full Maude. Using this feature we provide an environment where CafeOBJ specifications can be tested, debugged, and proved correct by integrating the Maude Declarative Debugger and Test-case Generator and the Constructor-based Inductive Theorem Prover.

We want to improve the implementation of the metalevel in two different ways: first, we want to define the syntax for representing views, in such a way that they can also be analyzed and modified. On the other hand, we are interested in defining more execution commands: currently only `metaReduce`, `metaRewrite`, and `metaFrewrite` are available, but several others can be implemented using our translation for CafeOBJ specifications and the built-in commands in Maude metalevel. Another interesting topic would be distinguish between behavioral and non-behavioral specifications when translating and executing the modules.

We are currently working to extend our framework with the Maude Formal Environment (MFE) [6]. This environment allows to check properties such as termination, confluence, and coherence on Maude specifications. It also includes the Inductive Theorem Prover [4], a tool to prove inductive properties on equational Maude specifications. Integrating this environment with CafeOBJ specifications would allow us to check that the executability requirements hold.

We are also interested in integrating Real-Time Maude [14] in our framework. This integration would be specially interesting for CafeOBJ users, since several protocols, such as [15], has already been specified in CafeOBJ. However, this integration is not straightforward, since it requires to extend the syntax of CafeOBJ specifications with timed transitions, as originally implemented for Maude.

Besides connecting more tools, we are also interested in extending the commands for CafeOBJ. More specifically, we are interested in the `t1 =(m,n)=> t2` predicate, which indicates that the term `t2` is reachable from `t1`, with `m` the number of searched terms and `n` the depth of the search (both numbers can be set to `*` to indicate that it is unbounded). This predicate, that is not documented and allows several extra conditions to constrain the states, is similar to the `search` command in Maude. It is interesting to implement this predicate, since it would increment the amount of CafeOBJ commands supported by our interface while providing a documented version in terms of Maude.

References

1. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: A hierarchy of data types: from trees to sets. In: Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.) All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350, pp. 119–129. Springer, Heidelberg (2007)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. Talcott, C.: Maude Manual (Version 2.6), January 2011. http://maude.cs.uiuc.edu/maude2-manual
3. Clavel, M., Meseguer, J., Palomino, M.: Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. Theor. Comput. Sci. **373**(1–2), 70–91 (2007)
4. Clavel, M., Palomino, M., Riesco, A.: Introducing the ITP tool: a tutorial. J. Univ. Comput. Sci. **12**(11), 1618–1650 (2006). Programming and Languages. Special Issue with Extended Versions of Selected Papers from PROLE 2005: The 5th Spanish Conference on Programming and Languages
5. Durán, F., Ölveczky, P.C.: A guide to extending full maude illustrated with the implementation of real-time Maude. In: Roşu, G. (ed), Proceedings of the 7th International Workshop on Rewriting Logic and its Applications, WRLA 2008, vol. 238(3), Electronic Notes in Theoretical Computer Science, pp. 83–102. Elsevier (2009)
6. Durán, F., Rocha, C., Álvarez, J.M.: Towards a Maude formal environment. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 329–351. Springer, Heidelberg (2011)

7. Eker, S., Martí-Oliet, N., Meseguer, J., Verdejo, A.: Deduction, strategies, and rewriting. In: Archer, M., de la Tour, T.B., Muñoz, C.A. (eds.) Proceedings of the 6th International Workshop on Strategies in Automated Deduction (STRATEGIES 2006), vol. 174, Electronic Notes in Theoretical Computer Science, pp. 3–25. Elsevier (2007)
8. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: Proceedings of the 39th Symposium on Principles of Programming Languages, POPL 2012, pp. 533–544. ACM (2012)
9. Futatsugi, K., Diaconescu, R.: CafeOBJ Report. World Scientific, AMAST Series (1998)
10. Futatsugi, K., Gâinâ, D., Ogata, K.: Principles of proof scores in CafeOBJ. Theor. Comput. Sci. **464**, 90–112 (2012)
11. Găină, D., Zhang, M., Chiba, Y., Arimoto, Y.: Constructor-based inductive theorem prover. In: Heckel, R., Milius, S. (eds.) CALCO 2013. LNCS, vol. 8089, pp. 328–333. Springer, Heidelberg (2013)
12. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theor. Comput. Sci. **373**(3), 213–237 (2007)
13. Nakagawa, A.T., Sawada, T., Futatsugi, K.: CafeOBJ User's Manual (version 1.4.8), July 2010. http://www.comp.dit.ie/pbrowne/compfund2/manual.pdf
14. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of real-time Maude. High. Order Symbolic Comput. **20**, 161–196 (2007)
15. Ouranos, I., Ogata, K., Stefaneas, P.: Formal analysis of TESLA protocol in the timed OTS/CafeOBJ method. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part II. LNCS, vol. 7610, pp. 126–142. Springer, Heidelberg (2012)
16. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: Declarative debugging of rewriting logic specifications. J. Logic Algebraic Program. **81**(7–8), 851–897 (2012)
17. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. J. Logic Algebraic Program. **67**, 226–293 (2006)
18. Zhang, M., Ogata, K.: Modular implementation of a translator from behavioral specifications to rewrite theory specifications. In: Choi, B. (ed.) Proceedings of the 9th International Conference on Quality Software, QSIC 2009, pp. 406–411. IEEE Computer Society (2009)