Tool Interoperability in the Maude Formal Environment

Francisco Durán[1], Camilo Rocha[2], and José M. Álvarez[1]

[1] Universidad de Málaga, Spain
[2] University of Illinois at Urbana-Champaign, IL, USA

Abstract. We present the Maude Formal Environment (MFE), an executable formal specification in Maude within which a user can seamlessly interact with the Maude Termination Tool, the Maude Sufficient Completeness Checker, the Church-Rosser Checker, the Coherence Checker, and the Maude Inductive Theorem Prover. We explain the high-level design decisions behind MFE, give a summarized account of its main features, and illustrate with an example the interoperation of the tools available in its current release.

1 Introduction

Maude [1] is a reflective declarative language and system based on rewriting logic in which computation corresponds to efficient deduction by rewriting. Several tools for the formal analysis of Maude modules have been available for a number of years. They include an inductive theorem prover [4] for equational specifications, and an LTL model checker [10], a reachability analysis tool [1], and an invariant analyzer [13] for rewrite specifications. However, many verification techniques such as the ones implemented by the tools just mentioned assume that the input module satisfies certain properties. For example, any verification task with the LTL model checker on a rewrite specification $\mathcal{R} = (\Sigma, E \cup A, R)$ assumes that $\mathcal{R}$ satisfies the so-called *executability requirements*, namely, that the equations E specifying the functional part of $\mathcal{R}$ are both ground terminating and ground Church-Rosser modulo the structural axioms A, and that the rules R specifying the concurrent transitions of $\mathcal{R}$ are ground coherent w.r.t. E modulo A. If they do not hold, then any analysis performed by the LTL model checker on $\mathcal{R}$ can in general lead to unsound and incomplete results.

Maude has been successfully used as a *metatool* in the creation of tools for verifying properties of its modules [2]. In this sense, previous work presented in [3] describes the main features of several tools concerned with the analysis of either Maude modules or of extensions of Maude. However, these tools work in isolation, making it inconvenient to switch between their execution environments and difficult to exchange data between them. In response to these limitations, we present the Maude Formal Environment (MFE), an executable and highly extensible software infrastructure within which a user can interact with several tools to mechanically verify properties of Maude modules. In MFE, tools can interoperate to discharge proof obligations of different nature without switching between different execution environments. The integration of different tools inside MFE's common environment presents the user with a consistent user interface, a mechanism to keep track of pending proof obligations, and allows the execution of several instances of each tool, among other features.

The following tools are currently available as part of MFE: the Maude Termination Tool (MTT) can be used to prove termination of equational and rewrite specifications [5]; the Sufficient Completeness Checker (SCC) can be used to check completeness and freeness of equational specifications, and deadlock freedom of rewrite specifications [11,12]; the Church-Rosser Checker (CRC) can be used to check ground confluence and sort-decreasingness of equational specifications [7]; the Coherence Checker (ChC) can be used to check the ground coherence of rewrite specifications [8]; and the Inductive Theorem Prover (ITP) can be used to verify inductive properties of equational specifications [4,11].

Outline. In Section 2 we give a high-level overview of MFE's design features and we explain some of the commands available. In Section 3 we illustrate MFE's main functionality with a case study in which a user interacts with several of the tools in the environment. The executables, a white paper explaining MFE's design, examples, and preliminary documentation, are available at `http://maude.lcc.uma.es/MFE`. For further details on the tools available in MFE please check the given references.

2 MFE's Design and Main Features

MFE has been implemented as an extension of Full Maude [6], thus benefiting from its functionality and flexibility. Full Maude is an extension of Maude written in Maude itself that has become a common infrastructure on top of which tools can be built. In MFE, for instance, tools provided by Maude such as its LTL model checker and search command are available, as well as modules defined in Full Maude (including object-oriented ones) are directly amenable to formal verification.

MFE is highly extensible and amenable to tool interoperability given its modular design and the fact that it imposes no constraint on how each tool should model its particular domain or maintains its internal state. MFE is modeled in Maude as an interactive object-based system where tools are objects, the communication mechanism is message passing, and user interaction is available through Full Maude. Integration and interoperation of tools within MFE is module-centric, given that its main purpose is to support formal analysis of Maude modules.

The object-oriented model of MFE consists of three classes: the class `Proof` of proof objects which keep the state of specific proof requests, the class `Tool` of tool objects which manage proof objects, and a class `Controller` which inherits from the Full Maude's `DatabaseClass` and provides a centralized entry point for handling requests to the formal environment.

The `Controller` object defines the behavior of the environment and its tools with the user. The user interacts with the environment via commands which are encapsulated as messages in the object configuration. Each tool object and the controller object have a module defining the signature of the commands it can handle. The controller handles any command it can parse; since this object extends Full Maude, it handles its own commands and Full Maude ones. If the controller receives a command it cannot parse, then it delegates the message to the *active* tool (previously selected by the user). If the active tool can parse the delegated command, then it notifies the controller and handles the command. Otherwise, it will notify the failure to the controller, which in turn will output an error message of the failed command to the user.

Classes `Proof` and `Tool` define some basic functionality that can be inherited by any new tool. Class `Tool`, for example, defines a set of attributes that are convenient for supporting multiple instances of a tool and predefines some rewrite rules for managing the life cycle of proof objects. However, new tools can be added to MFE without inheriting from any of these two classes.

MFE provides the following user commands, in addition to the ones inherited from Full Maude:

`(select tool <tool-name> .)` sets `<tool-name>` as the *active* tool.
`(MFE help .)` shows MFE's help information.
`(show global state .)` shows the state of the environment.

Any command that cannot be parsed against MFE's grammar is delegated to the active tool. In this way, user interaction with any tools remains almost the same as before its integration in MFE. We refer the reader to MFE's documentation and to each tool's documentation for a detailed account of commands, restrictions, and additional examples.

In order to be flexible, MFE does not define any policy for naming tool commands. However, as a general guideline for using the environment, it is recomended that the tools provide at least the following commands, as it is the case with the tools available in MFE's current release:

`(<tool-name> help .)` shows the help information of tool `<tool-name>`.
`(show state .)` shows the state of the tool.

MFE's design supports non-trivial dependencies among tools and potential interaction complexity. For instance, a ground coherence check on a rewrite specification assumes the ground termination and ground Church-Rosser properties on its equational subspecification, and may produce a number of inductive proof obligations that could be discharged with help of ITP. Figure 1 depicts the tool-dependency graph for the current tools in MFE.

$$\begin{tikzcd}
\text{MTT} & & & \\
& \text{ChC} \arrow[ul] \arrow[dl] \arrow[r] & \text{ITP} \arrow[ull] \arrow[r] \arrow[dll, leftrightarrow] & \text{SCC} \arrow[ulll] \arrow[dlll, leftrightarrow] \\
\text{CRC} \arrow[uu] & & &
\end{tikzcd}$$

Fig. 1. Tool-dependency graph in MFE

A tool in MFE keeps track of both its pending and discharged proof obligations. It can submit proof obligations to other tools by means of a `submit` command and then be notified when these are discharged. When all proof obligations in the verification process of a module's property are discharged, the corresponding tool notifies the success result to the user or to the tool originating the verification task.

Of course, tools in general can impose constraints on its inputs. For instance, SCC does not support parametric modules but, nevertheless, proofs for such modules could be obtained by hand or by using a different tool. MFE offers a `trust` command for keeping track of proofs obtained outside MFE.

Finally, for tools which depend on external utilities not directly available from Maude such as MTT and SCC, we have extended the latest release of the Maude system with *built-in* operators associated with appropriate C++ code that interacts with the external tools. A similar extension was previously performed for the SCC [11].

3 Case Study: Ground Coherence of the Bakery Protocol

We explain how MFE can be used to prove the ground coherence of module `BAKERY`, yet another Maude specification of the Bakery Protocol. The Bakery Protocol is a classical solution by Lamport to the problem of achieving mutual exclusion between processes. The protocol is based on the procedure commonly used in bakeries where a customer is assigned a ticket number upon arrival. Here, a process is a term with sort `BProcess` built from operator `<_,_,_>` that takes a natural number with sort `Nat` as identifier, a constant term `sleep`, `wait`, or `crit` with sort `Mode` as current state, and a natural number as ticket number. A term with sort `BState` is a multiset of processes, where each process is a singleton multiset and union is denoted by juxtaposition. A term with sort `GBState` represents the system's state and it is built from operator `[[_]]` that takes a multiset of processes as argument.

The concurrent behavior of the Bakery Protocol is modeled in the `BAKERY` system module as follows.

```
(mod BAKERY is
  protecting MNAT .

  sorts Id Mode BProcess BState GBState .
  ops sleep wait crit : -> Mode [ctor] .
  op <_`,_`,_> : MNat Mode MNat -> BProcess [ctor] .
  subsort BProcess < BState .
  op __ : BState BState -> BState [ctor assoc comm id: none] .
  op none : -> BState [ctor] .
  op `[`[_`]`] : BState -> GBState [ctor] .

  var P : Mode . vars I N M : MNat . var BSt : BState .

  ---- max of the numbers assigned to processes (0 if none)
  op maxNumber : BState -> MNat .
  op maxNumber : BState MNat -> MNat .
  eq maxNumber(< I, P, N > BSt) = max(N, maxNumber(BSt)) .
  eq maxNumber(none) = 0 .

  ---- min. of the nonzero numbers assigned to processes (0 if none)
  op minNzNumber : BState -> MNat .
  op minNzNumber : BState MNat -> MNat .
  eq minNzNumber(< I, P, 0 > BSt) = minNzNumber(BSt) .
  eq minNzNumber(< I, P, s N > BSt) = minNzNumber(BSt, s N) .
  eq minNzNumber(none) = 0 .
  eq minNzNumber(< I, P, 0 > BSt, M) = minNzNumber(BSt, M) .
  eq minNzNumber(< I, P, s N > BSt, M) = minNzNumber(BSt, min(M, s N)) .
  eq minNzNumber(none, M) = M .

  rl [s2w] : [[ < I, sleep, 0 > BSt ]]
    => [[ < I, wait, s maxNumber(BSt) > BSt ]] .
 crl [w2c] : [[ < I, wait, N > BSt ]]
    => [[ < I, crit, N > BSt ]]
    if N = minNzNumber(< I, wait, N > BSt) .
  rl [c2s] : [[ < I, crit, N > BSt ]]
    => [[ < I, sleep, 0 > BSt ]] .
endm)
```

Operators `maxNumber` and `minNzNumber` operate on terms with sort `BState` and compute, respectively, the maximum and minimum ticket numbers in a multiset of processes, or `0` if none.

In MFE, `BAKERY`'s executability properties can be proved in different order. For instance, we first activate CRC and use its `check Church-Rosser` command to first verify `BAKERY`'s equational subspecification Church-Rosser.

```
Maude> (select tool CRC .)
The CRC has been set as current tool.
```

```
Maude> (check Church-Rosser BAKERY .)
Church-Rosser check for BAKERY
   All critical pairs have been joined.
   The specification is locally-confluent.
   The module is sort-decreasing.
```

All critical pairs are joined and consequently the specification is locally confluent. Notice also that CRC proves the equations sort-decreasing. Hence, a proof of termination would imply the ground Church-Rosser property of `BAKERY`'s equational part.

In this case, termination of `BAKERY`'s equational part is the only pending proof obligation. We ask the CRC to submit this proof obligation and then activate MTT.

```
Maude> (submit .)
The termination goal for the functional part of BAKERY has been submitted to MTT.
Warning: A proof of the termination of functional part of module BAKERY has not
     ↪been found.
```

MTT is not able to find a proof automatically, and we need to interact with it to go on.

```
Maude> (select tool MTT .)
The MTT has been set as current tool.
```

A proof of the termination of this specification can be found in [9]. Instead of completing the proof here, let us rely on this reference and use the `trust` command to tell the tool that it can assume the existence of a proof of the termination of the current module and proceed.[1]

```
Maude> (trust .)
The functional part of module BAKERY is assumed terminating.
Success: The module is therefore Church-Rosser.
```

Upon notification from MTT that a termination proof has been found, CRC notifies the user that `BAKERY`'s functional part is Church-Rosser. Then, it only remains to prove

[1] If you are running a version of Maude that does not include the hooks to the external termination tools, you would get a message indicating that the tool cannot be used to prove the termination of the current module. You will still be able to use the `trust` command.

BAKERY ground coherent. We set ChC as the active tool in MFE and issue the corresponding checking command.[2]

```
Maude> (select tool ChC .)
The ChC has been set as current tool.
```

```
Maude> (check ground coherence BAKERY .)
Ground coherence checking of BAKERY
   All critical pairs have been rewritten and no rewrite with rules can happen at
       ↪ non−overlapping positions of equations left−hand sides.
   The sufficient−completeness, termination and Church−Rosser properties must
       ↪still be checked.
```

The decision procedure implemented by ChC discharges all critical pairs between equations and rules. However, this procedure requires BAKERY's functional part to be sufficiently complete, ground terminating, and ground Church-Rosser. Since the termination and Church-Rosser properties have previously been proved, ChC is notified that such proofs have been found when submitting the proof obligations.

```
Maude> (submit .)
The Church−Rosser goal for BAKERY has been submitted to CRC.
The Sufficient−Completeness goal for BAKERY has been submitted to SCC.
The termination goal for the functional part of BAKERY has been submitted to MTT.
Success: The equational theory of BAKERY does not have counterexamples for
    ↪sufficient completeness.
   However,this is under the assumption that it is ground weakly−normalizing and
       ↪ground sort−decreasing.
The functional part of module BAKERY has been checked terminating.
The module BAKERY has been checked Church−Rosser.
```

Although we already have all the pieces, we still need to select the SCC tool to complete its proof.

```
Maude> (select tool SCC .)
The SCC has been set as current tool.
```

```
Maude> (submit .)
The sort−decreasingness goal for BAKERY has been submitted to CRC.
The termination goal for the functional part of BAKERY has been submitted to MTT.
Church−Rosser check for BAKERY
   The module is sort−decreasing.
The module BAKERY has been checked sufficiently−complete.
Success: The module BAKERY is ground−coherent.
```

Thus, as desired, we conclude that system module BAKERY is (ground) coherent.

4 Future Work

More tools such as Maude's LTL and LTLR Model Checkers, Maude's Invariant Analyzer, and Real-Time Maude could be integrated in MFE. This will result in a more comprehensive environment with more features and broader applications. One could also think of MFE automatically generating the proof obligations associated to the semantics of protected and extended modules, and to that of parameterized modules.

[2] Notice that for the proof of ground coherence, assuming sufficient completeness, defined operators can be regarded as frozen (see [8]).

More ambitiously, a graphical user interface and support for distributed interoperation will enhance the user experience within MFE.

Acknowledgments. We thank the anonymous referees for their comments, and for taking the time to evaluate the manuscript and the tool. The first author was partially supported by projects TIN2008-03107 and P07-TIC-03184. The second author has been partially supported by NSF grants CNS 07-16638 and CCF 09-05584.

References

1. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Bevilacqua, V., Talcott, C.: All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
2. Clavel, M., Durán, F., Eker, S., Meseguer, J., Stehr, M.O.: Maude as a formal meta-tool. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1684–1703. Springer, Heidelberg (1999)
3. Clavel, M., Durán, F., Hendrix, J., Lucas, S., Meseguer, J., Ölveczky, P.: The Maude formal tool environment. In: Mossakowski, T., Montanari, U., Haveraaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 173–178. Springer, Heidelberg (2007)
4. Clavel, M., Palomino, M., Riesco, A.: Introducing the ITP tool: a tutorial. Journal of Universal Computer Science 12(11), 1618–1650 (2006)
5. Durán, F., Lucas, S., Meseguer, J.: MTT: The Maude termination tool (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 313–319. Springer, Heidelberg (2008)
6. Durán, F., Meseguer, J.: Maude's module algebra. Science of Computer Programming 66(2), 125–153 (2007)
7. Durán, F., Meseguer, J.: A church-rosser checker tool for conditional order-sorted equational maude specifications. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 69–85. Springer, Heidelberg (2010)
8. Durán, F., Meseguer, J.: A maude coherence checker tool for conditional order-sorted rewrite theories. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 86–103. Springer, Heidelberg (2010)
9. Durán, F., Meseguer, J.: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. Journal of Logic and Algebraic Programming Journal of Logic and Algebraic Programming (2011) (accepted for publication)
10. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Gaducci, F., Montanari, U. (eds.) Proceedings of 4th International Workshop on Rewriting Logic and its Applications (WRLA 2002). Electronic Notes in Theoretical Computer Science, vol. 71 (2002)
11. Hendrix, J.: Decision Procedures for Equationally Based Reasoning. Ph.D. thesis. University of Illinois at Urbana-Champaign (2008)
12. Rocha, C., Meseguer, J.: Constructors, sufficient completeness and deadlock freedom of rewrite theories. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 594–609. Springer, Heidelberg (2010)
13. Rocha, C., Meseguer, J.: Proving safety properties of rewrite theories. Tech. rep. University of Illinois at Urbana-Champaign (2010), `http://hdl.handle.net/2142/17407`