Towards a 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. Maude is a declarative and reflective language based on rewriting logic in which computation corresponds to efficient deduction by rewriting. Because of its reflective capabilities, Maude has been useful as a *metatool* in the development of formal analysis tools for checking specific properties of Maude specifications. This includes tools for checking termination, confluence, and inductive properties of rewrite theories. Nevertheless, most of these tools have been designed to work in isolation, making it difficult, for instance, to exchange data between them and inconvenient to switch between their environments. This paper presents the Maude Formal Environment (MFE), an executable formal specification in Maude within which a user can interact with tools to mechanically verify properties of Maude specifications. One important aspect of this work is that the MFE has been designed to be easily extended with tools having highly heterogeneous designs whilst creating synergy among them. As a proof of concept, we report on the integration of five commonly used formal analysis tools for Maude specifications into MFE and illustrate their interoperability with an example.

1 Introduction

There is a great deal of interest today in developing multipurpose environments that combine declarative programming with specification languages and useful formal analysis tools (see, e.g., [**?**,**?**,**?**,**?**,**?**]). Maude [**?**,**?**] is a reflective declarative language and system based on rewriting logic in which computation corresponds to efficient deduction by rewriting. Maude has been successfully used as a *metatool* in the creation of tools for verifying properties of Maude specifications [**?**,**?**]. Nevertheless, these tools work in isolation, making it inconvenient to switch between their environments and difficult to exchange data between them. In this sense, despite its title, previous work presented in [**?**] does not conform to the notion of formal tool environment discussed here. 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 specifications. In MFE, tool interoperability allows for discharging proof obligations of different nature without switching between different tool 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 purpose of MFE is to support interactive formal analysis of Maude specifications, and therefore the integration of tools within MFE revolves around Maude modules. MFE is naturally modeled in Maude as an object-based system in which the tools are objects and their communication mechanism is message passing. User interaction is available through Full Maude [**?**,**?**], an extension of Maude that has become a common base on top of which tools can be built, offering a modular design for easily integrating other tools written in Maude (see, e.g., [**?**] for a guide on how to extend Full Maude).

One of the most interesting challenges was to make the MFE design highly extensible and amenable to tool interoperability. In MFE, there is no constraint on how each tool should model its particular domain or how it maintains its internal state. We implemented an object-based version of the *model-view-controller* pattern to separate the modeling of the domain for each tool, the presentation of information, and the actions based on user input into separate objects. This pattern isolates the application logic for the user from the user input and presentation, permitting independent development, testing and maintenance of each. We also followed good object-oriented design practices that kept the cohesion high and the coupling low among objects. In our experience, these good design practices proved key for the integration of tools in MFE.

We report here on the integration of five tools with highly different designs and implementations in MFE. Namely, the Maude Termination Tool (MTT) [**?**], the Church-Rosser Checker (CRC) [**?**,**?**], the Coherence Checker (ChC) [**?**,**?**], the Sufficient Completeness Checker (SCC) [**?**,**?**], and Maude's Inductive Theorem Prover (ITP) [**?**,**?**]. Despite their heterogeneousness and isolated conception, these tools were integrated in MFE with very few code alterations, many of these due to renaming of sorts and operators. For tools which depend on external utilities not directly available from Maude such as MTT and SCC, we have extended the Maude system to a non-official distribution with *built-in* operators associated with appropriate C++ code that interacts with the external tools. A similar extension was already performed for SCC [**?**].

MFE, with these five tools, as well as some examples and some preliminary documentation, is available at `http://maude.lcc.uma.es/MFE`.

Outline of the paper. Section **??** gives a summarized account of Maude's object-based programing and support for user interoperability. Section **??** discusses the main design aspects of MFE. Section **??** describes the tools available from the current version of MFE. Section **??** illustrates how to extend MFE with a concrete tool and Section **??** presents a case study in which a user interacts with several tools within MFE. Finally, Section **??** presents related and future work, and some concluding remarks.

2 Object-Based Programming and User Interfaces in Maude

We assume that the reader is familiar with the basics of rewriting logic and Maude, and refer to [**?**] for an introduction to these.

Maude can be used not only to define domain-specific languages or tools, but also to build environments for such languages and tools. In such applications, the predefined `LOOP-MODE` module can be used to handle the input/output and to maintain the persistent state of the language environment or tool. This section explains some basic background on how object-based systems, which naturally model distributed systems in Maude, and the `LOOP-MODE` module are used to define MFE's interactive infrastructure as an extension of Full Maude.

Object-based programming. Maude supports the modeling of object-based systems in the predefined `CONFIGURATION` module that declares sorts representing the essential concepts of object, message, and configuration, along with notation for object syntax that serves as a common language for specific object-based systems. The basic sorts defined in `CONFIGURATION` are `Object`, `Msg`, and `Configuration`. A configuration is a multiset of objects and messages that represent a possible system state. Configurations are formed by multiset union, represented by empty syntax __, starting from singleton objects and messages. The empty configuration is represented by the constant `none`.

```
sort Configuration .
subsorts Object Message < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration
    [ctor assoc comm id: none] .
```

In general, a rewrite rule for an object-based system has the form

```
crl [r] :
  Obj_1 ... Obj_m Msg_1 ... Msg_n
  => Obj'_1 ... Obj'_j Obj_{m+1} ... Obj_p Msg'_1 ... Msg'_q
  if Cond .
```

where objects $Obj'_1 \ldots Obj'_j$ are updated versions of objects $Obj_1 \ldots Obj_m$, for $j \leq m$, $Obj_{m+1} \ldots Obj_p$ are newly created objects, $Msg_1 \ldots Msg_n$ are consumed messages, and $Msg'_1 \ldots Msg'_q$ are new messages.

The user is free to define any object or message syntax that is convenient. However, for uniformity in identifying objects and message receivers, the adopted convention is that *the first argument of an object constructor should be its identifier*, and *the first argument of a message constructor should be the identifier of its addressee*. Module `CONFIGURATION` provides an object syntax that serves as a common notation that can be used by developers of object-based system specifications, as is the case in Full Maude. This module introduces sorts `Oid` for object identifiers, `Cid` for class identifiers, `Attribute` for named elements of an object's state, and `AttributeSet` for multisets of attributes. In this syntax, objects have the general form $< O : C \mid a_1 : v_1, \ldots, a_n : v_n >$ where O is an object identifier, C is a class identifier, and the $a_i : v_i$ are pairs of an attribute name a_i and a value v_i, for $1 \leq i \leq n$.

Full Maude provides convenient notation for object-oriented modules in which classes are declared with the syntax

```
class C | a_1 : S_1, ..., a_n : S_n .
```

where C is the name of the class, the a_i are attribute identifiers, and the S_i are the sorts of the corresponding attributes. Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration

```
subclass C < C' .
```

is just a particular case of a subsort declaration $C < C'$. The effect of a subclass declaration is that the attributes, messages, and rules of all the superclasses, together with the newly defined attributes, messages, and rules of the subclass, characterize the structure and behavior of the objects in the subclass. In what follows, we use this convenient object-oriented notation for defining classes. See [**?**] for further details on this notation and on the transformation of object-oriented modules into system modules.

User interfaces. Module `LOOP-MODE` specifies in Maude a general input/output facility by a read-eval-print loop using object-based concepts. A *loop object* is a term of the form `[In,St,Out]` where `In` is an input stream, `Out` is an output stream, and `St` is its state. One can think of the input and output events as implicit *rewrites* that transfer the input and output data between two objects, namely the loop object and the user (or terminal) object.

Loop objects are constructed with the operator

```
  op [_,_,_] : QidList State QidList -> System [...] .
```

Besides having input and output streams, terms of sort `System` give a generic way for maintaining a state in its second component. In fact, sort `State` in `LOOP-MODE` does not have any constructors, giving complete flexibility for defining the terms we want to have for representing the state of the loop. In MFE, we represent state terms as configurations of objects and messages, by declaring sort `Configuration` as subsort of `State`.

Rewrite rules define the interaction of the state with the loop and the changes produced in the state by the actions requested by the user. In order to generate in Maude an *interface* for interacting with an application, the language for interaction needs to be defined by a data type for commands and other constructs. In this way, a rule can detect when a valid request has been introduced by the user, and if the state of the system allows it, passes it as the next action to be attempted. For the other direction of interaction, a rule detects when the state has a response to be output and, in that case, it places it in the output component of the loop object.

Full Maude. In Full Maude, the persistent state of the read-eval-print loop provided by module `LOOP-MODE` is given by a single object of class `DatabaseClass`. Objects of this class have: an attribute `db` of sort `Database` to keep the actual database where all the modules entered to the system are stored, an attribute `default` denoting the name of the current default module, and attributes `input` and `output` that simplify the communication between the read-eval-print loop and the database object. Using the above syntactic sugar for object-oriented modules, we can declare such a class as:

```
 class DatabaseClass |
   db : Database, default : ModName, input : TermList, output : QidList .
```

Inputs from the user into Full Maude are parsed using the built-in `metaParse` function. For such parsing, Full Maude uses the `FULL-MAUDE-SIGN` module, in which we can find the declarations so that any valid input can be parsed. In particular, we find in these modules, among others, sorts `@Module@`, `@ModExp@`, and `@Command@`, of modules, module expressions, and commands, respectively, and syntax declarations such as:

```
 op select_. : @ModExp@ -> @Command@ .
```

```
op show module_. : @ModExp@ -> @Command@ .
op mod_is_endm : @Interface@ @SDeclList@ -> @Module@ .
op omod_is_endom : @Interface@ @ODeclList@ -> @Module@ .
```

for commands `select` and `show module`, and for system and object-oriented modules.

The behavior of Full Maude upon the reception of new inputs from the user is specified by rewrite rules. For the different commands, different actions are accomplished.

3 The Design of MFE

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

The `Controller` object orchestrates the behavior of the environment with the user and of the environment with its tools. The user interacts with the environment via commands that are encapsulated as messages in the object configuration. Each tool object and the controller object have a module defining the grammar 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, it will delegate it to the current *active* tool. If the active tool can parse the delegated command, then it notifies the controller and handles the command. Otherwise, it notifies the failure to the controller that in turn will notify the failure to the requester.

Classes `Proof` and `Tool` define some basic functionality for tools and, as we will see in Section **??** for a sample tool, are provided to simplify the task of incorporating new tools to the environment. However, tools can be added to the environment by defining the expected interaction with the controller object without using classes `Proof` and `Tool`. This was the case, for example, with the ITP tool that does not use in MFE the infrastructure provided by classes `Proof` and `Tool`.

In the following subsections we describe in more detail the object-based model of MFE and its interaction mechanism.

3.1 Proof Objects

Proof objects maintain the state of specific proof requests to a tool object. Every proof object maintains in its state the information of the module associated to the proof obligation and a set of object identifiers corresponding to the objects submitting the proof obligation.

```
class Proof | module : Module, requester : Set{Oid} .
```

The concept of a proof object representing the state of a proof requirement, is key for enabling tools in MFE with support for multiple proof requirements. Namely, handling a "new proof" command corresponds to instantiating a proof object with the appropriate attribute data. Commands that incrementally modify the status of a proof obligation result in updates to the attributes of the proof object. For example, a CRC proof object will keep track of confluence and/or sort-decreasingness checks, and will be updated every time a new proof obligation is discharged; when all proof obligations have been proved, it will realize the check's completion and will inform all its requesters.

A subclass of `Proof` may be defined for each tool in the environment, adding the additional attributes and behavior required by the specific tool. Proof obligation objects, for instance, can be extended with additional attributes for keeping track of dependencies of subgoals that should be handled by other tools, timeouts, number of attempts, or any other required information.

3.2 Tool Objects

A tool object is responsible for maintaining the life cycle of its proof objects. When a tool object receives a request for a new proof, it tries to create a new proof object for it and, if successful, it sets the new proof object as active so that any command from the user or message from other tools in the environment are forwarded to it. There is exactly one tool object for each tool in the formal environment.

Every tool object has an attribute `grammar` that defines the grammar of user commands the tool can handle. Tools may rely on other tools, hence a tool object has attribute `tools` with a map from tool names to the object identifiers of the corresponding tools in the environment. If proof obligations are due, this attribute will be used to submit them to the appropriate tools. The references of proof objects associated to a tool object are maintained with a map from module names to object identifiers in attribute `reg`. In MFE, a tool may perform several analyses on a module, so that the information of such analyses is kept in the attributes of the corresponding proof object. A tool object also keeps in attribute `current` a reference to one of its proof objects, if any, which is referred by the tool as its *active* proof obligation.

```
class Tool | grammar : Module,
             tools : Map{ToolName, Oid},
             reg : Map{ModuleName, Oid},
             current : Oid,
             index : Nat .
```

Integration and interoperability of tools within MFE revolves around modules, and therefore, typically, the "new goal" commands have a module expression as parameter, although in general commands may have other parameters. For example, a check of the Church-Rosser property would take just the module to be checked as parameter. However, one can perform checks of coherence in the coherence checker with the option of checking coherence or ground coherence via an additional parameter. The decision of whether a new proof object is generated or not for a module when attempting different kinds of checks is up to the tool developer.

Despite this flexibility, in the general approach a tool object will create a new proof object for a module *M* whenever there is no record of *M* being previously handled by the tool, namely, if name of module *M* is not in the domain of the `reg` attribute. More precisely, when a proof is requested, the `reg` attribute is checked: if there is a proof for a module with such a name, then the module itself is compared with the current one, to make sure that the module has not changed. In case there is some parameter, as for example an alternative transformation for a termination proof, if the proof has not succeeded before then the check is attempted: the existing proof object is replaced with the one corresponding to the new proof.

The different tools may perform tool specific checks on a parameter module *M* and if these checks succeed, then a new proof object is instantiated with a unique object identifier, with *M* as associated module (in attribute `module`), and the corresponding reference of the requesting object (in attribute `requester`). The remaining attributes of the proof object are set according to the purpose of the tool.

See Section **??** for an example of this behavior in the case of SCC.

3.3 The Controller Object

The task of the controller object is twofold: it provides a centralized entry point for handling user requests and it manages the tools that are available in the environment.

The `Controller` class inherits from Full Maude's `DatabaseClass`, and in addition to its module database and all the functionality for handling all Full Maude commands, the controller object stores information on the tools available in MFE. This is achieved by using the attribute `tools` that is a map from tool names into object identifiers. In the attribute `current-tool` the controller object maintains a reference to the active tool:

```
class Controller | current-tool : Oid, tools : Map{ToolName, Oid} .
subclass Controller < DatabaseClass .
```

The controller object is *the singleton* instance `mfe` of class `Controller`. To handle a command, the object `mfe` first tries using its grammar (which extends that of Full Maude). If the command can be parsed with its grammar, then `mfe` handles the request. Otherwise, it delegates the command to the active tool and waits for an answer. The user can select the active tool via a "select" command. The answer to the delegated request can be either affirmative or not, meaning that the tool can parse the command and will handle it, or that the given command does not conform to the grammar and therefore cannot be handled by the tool. Because of the way user and tool interaction has been designed and implemented in MFE, there is no need to enforce a policy of uniqueness of commands among its tools: if two tools share a command syntax, then such command will be handled by the controller object or else by the active tool. This simplifies the integration of existing tools, because most of their implementations can directly be used and because all proof scripts available are still usable after adding the appropriate selection and submission commands.

The `Controller` class adds the following commands to those available in Full Maude:

`(select tool <tool-name> .)` sets the tool `<tool-name>` as active tool.
`(MFE help .)` shows information on the commands available.
`(show global state .)` shows the state of the framework.

To illustrate the way in which the behavior of the controller object works, we present the `select-tool` rewrite rule that implements tool selection in MFE for the controller:

```
var X@Controller : Controller .   var Ct : Constant .
var TS : Map{ToolName, Oid} .     var QIL : QidList .
var Atts : AttributeSet .         var O : Oid .

crl [select-tool] :
   < mfe : X@Controller | input : ('select`tool_.[Ct]),
      output : QIL,
      current-tool : O,
      tools : TOOLS,
      Atts >
   => < mfe : X@Controller | input : nilTermList,
         output : QIL 'The getName(Ct) 'has 'been 'set 'as 'active 'tool.,
         current-tool : TOOLS[qid2tool(getName(Ct))],
         tools : TOOLS,
         Atts >
   if TOOLS[qid2tool(getName(Ct))] =/= null .
```

When the result of parsing a "select tool" command in the grammar of the controller is placed in the `input` attribute of the `mfe` object, and it corresponds to a tool in the environment (see the condition in rule `select-tool`), then such a tool is set as the active one. Functions `getName` and `qid2tool` return the name of a given constant and transforms a quoted identifier into a tool name, respectively. A second rule (omitted here) handles the case in which the argument of the select command does not correspond to a tool in the environment; this rule reports on the situation by creating an error message.

3.4 User Interaction and Tool Interoperability

In the object configuration of MFE, user interaction is achieved via commands and tool interoperability via messages. Upon successful parsing, commands are converted into messages. With this approach, requests from users and from tools are handled in a uniform manner by just distinguishing the requester.

MFE internal messages identify their contents with "to-from" information and name the different operations offered by the tools and their answers. Using the following syntax for messages, each tool defines its corresponding message bodies.

```
sort MFEMsgBody .
op to_from_:_ : Oid Oid MFEMsgBody -> Msg [ctor] .
```

If a user command parses in the controller's grammar, then the controller handles the command. If it fails, then the input is submitted to the active tool. The tool is expected to return an "input parsed" message indicating whether or not it is able to parse the command or not. Rewrite rule `input`, below, defines the behavior of a tool object that is able to parse the input command. Observe that when a tool object can parse a command, it sends two messages. Namely, it creates a copy of the original message but with a parsed version of the input command and it sends the requester an output message indicating that the input can be parsed.

```
  var  X@Tool : Tool .                 vars  O O' : Oid .
  var  Atts : AttributeSet .           var  QIL : QidList .
  var  G : Module .

 crl [input] :
    < O : X@Tool | grammar : G, Atts >
    (to O from O' : input(QIL))
    => < O : X@Tool | grammar : G, Atts >
       (to O from O' : getTerm(metaParse(G, QIL, '@Input@)))
       (to O' from O : input-parsed(QIL, true))
    if RP:ResultPair := metaParse(G, QIL, '@Input@) .
```

If the tool cannot parse the input, then another rule (omitted here) sends the requester an `input-parsed` message with its second argument set to `false`. When the controller receives the `input-parsed` message, with `true` or `false`, it proceeds either by letting the corresponding tool resolve the command or by displaying an error message.

4 Tools in MFE

Five formal analysis tools are available in the current release of MFE. The Maude Termination Tool (MTT) can be used to prove termination of functional and system modules, the Church-Rosser Checker (CRC) can be used to check the Church-Rosser property of functional modules, the Coherence Checker (ChC) can be used to check the coherence of system modules, the Inductive Theorem Prover (ITP) can be used to verify inductive properties of functional modules, and the Sufficient Completeness Checker (SCC) can be used to check completeness of functional modules and deadlock freedom of system modules. One important aspect in the integration task is the interaction complexity due to the nontrivial dependencies among tools. Figure **??** depicts the tool-dependency graph for these five tools.

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

Fig. 1. Tool-dependency graph in MFE.

In the following paragraphs we summarize the main features and dependencies of the five tools available in MFE. For further details on these tools, including user manuals, restrictions, and examples, we refer the reader to the given references and web sites.

4.1 The Maude Termination Tool

Maude has expressive features including advanced typing constructs with sorts, subsorts, kinds, and memberships; matching modulo axioms; evaluation strategies for both equations and rewrite rules; and very general conditional equations and rewrite rules. Proving termination of programs having such features is nontrivial. Furthermore, some of these features are not supported by standard termination methods and tools. Yet, the use of such features may be essential to ensure termination. MTT uses several non-termination preserving theory transformations [**?**,**?**] which are applied in a kind of pipeline to a module M, obtaining a module M' in such a way that a proof termination of M' witnesses the termination of M. For instance, by sequentially using four theory transformations, a conditional order-sorted context-sensitive system module can be transformed into an unconditional unsorted context-sensitive term rewrite system, which can be handled by the back-end tools.

The Maude Termination Tool (MTT) is a tool that checks the termination of (possibly conditional) order-sorted Maude functional or system modules. The current implementation takes a module as input and tries to prove its termination by applying theory transformations and then invoking back-end termination tools, such as MU-TERM [**?**] and AProVE [**?**], that can prove termination of (variants of) rewriting. For the current version of MFE, a new "hook" to a C++ library was included in non-official distribution of Core Maude for invoking these back-end termination tools. MTT is the only tool in the MFE that does not depend on any other tool in the environment.

A stand-alone version of MTT, with a graphical user interface, is available from `http://www.lcc.uma.es/~duran/MTT/`.

4.2 The Church-Rosser Checker

The Church-Rosser Checker (CRC) checks the Church-Rosser property of Maude functional (possibly conditional) order-sorted modules. For order-sorted modules, being Church-Rosser and terminating means not only confluence, but also a sort-decreasingness property: each normal form has the least possible sort among those of all equivalent terms. CRC depends on MTT for checking termination assumptions and on ITP for inductive theorem proving (see Section **??**). Some of the proof obligations are not currently handled by any tool. We will se in Section **??** that although they are submitted to ITP, these proofs may be "trusted" by the user.

CRC can be used to check equational specifications $(\Sigma, E \cup Ax)$ with an initial semantics that have already been proved terminating and need to be checked (ground) Church-Rosser. The tool performs both a local confluence check by computing all *(conditional) critical pairs* of the equations E modulo the structural axioms Ax and a sort-decreasingness test in the form of membership assertions for each equation in E modulo Ax. If the (conditional) critical pairs or the sort decreasingness tests cannot be discharged by CRC, proof obligations are displayed as a guide to the user. In MFE, CRC can submit these proof obligations to ITP for inductive reasoning.

CRC, with its documentation and some examples, is available from `http://maude.lcc.uma.es/CRChC/`.

4.3 The Coherence Checker

(Ground) coherence allows reducing the problem of computing rewrites of the form $[t]_{E \cup Ax} \rightarrow [t']_{E \cup Ax}$ that in general is undecidable, to the much simpler and decidable problem of computing rewrites of the form $[t]_{Ax} \rightarrow [t']_{Ax}$, for t and t' any (ground) Σ-terms. Intuitively, coherence means that rewriting with R modulo $E \cup Ax$ can be achieved by adopting the strategy of first simplifying to canonical form with E modulo Ax and then applying a rule in R modulo Ax.

The Coherence Checker (ChC) is a tool that provides a (ground) coherence decision procedure for order-sorted system modules $\mathcal{R} = (\Sigma, E \cup Ax, R)$. The tool calculates the set of critical pairs between the equations E and the rewrite rules R modulo the structural axioms Ax, whose equational validity guarantees the (ground) coherence of $\mathcal{R}$. ChC depends on MTT for termination assumptions and on CRC for the equations E being Church-Rosser. Since this property is inductive, in some cases ITP can be used to prove some proof obligations.

ChC, with its documentation and some examples, is available from `http://maude.lcc.uma.es/CRChC/`.

4.4 The Sufficient Completeness Checker

Maude Sufficient Completeness Checker (SCC) is a tree automata based tool for checking *sufficient completeness* and *deadlock freedom* of terminating and sort-decreasing Maude modules. Both sufficient completeness and deadlock freedom are relative to *constructor* subsignatures. For $\mathcal{R} = (\Sigma, E, R)$, a signature pair (Υ, Ω), with $\Upsilon \subseteq \Omega \subseteq \Sigma$, is a pair of constructors for $\mathcal{R}$ if for each sort s in Σ and each ground Σ-term with sort s, (i) there is a Ω-ground term u with sort s satisfying the equality $t = u$ in E, and (ii) there is a ground Υ-term v with sort s satisfying the sequent $t \rightarrow v$ in R. Intuitively, sufficient completeness is the property that every operation in a specification is equationally defined for all inputs, and deadlock freedom is the property that every nondeterministic computation leads to a terminal state [**?**,**?**].

Sufficient completeness and deadlock freedom are important properties both for developers of specifications, to check that they have not missed a case in defining operations, and to inductive theorem provers, to check the soundness of a proposed induction scheme. In the case of equational specifications, SCC assumes the input specification is ground sort-decreasing and terminating; in the case of rewrite specifications, SCC assumes the input specification is ground sort-decreasing, terminating, Church-Rosser, and coherent.

The tool is designed for unparameterized, order-sorted, left-linear, and unconditional Maude specifications that are ground terminating and Church-Rosser. It is a decision procedure for this class of specifications when every associative symbol is also commutative. For associative symbols that are not commutative it uses machine learning techniques that work well in practice. If the specification is not sufficiently complete, SCC returns a counterexample to aid the user in identifying errors. The tool is not complete for specifications with non-linear or conditional axioms, but nevertheless has proven useful in identifying errors in such specifications. SCC accepts interactive commands to check the sufficient completeness of a Maude module, and internally constructs a propositional tree automaton whose language is empty if and only if the Maude module is sufficiently complete. The emptiness check is performed by a C++ tree automata library named CETA.

The tool also supports several important completeness and freeness problems of context-sensitive specifications involving both equations and rewrite rules [**?**,**?**].

SCC is available from its website at `http://maude.cs.uiuc.edu/tools/scc`.

4.5 The Maude Inductive Theorem Prover

The Maude Inductive Theorem Prover (ITP) is an experimental proof assistant aiding in the task of proving inductive properties of the initial algebra associated to a membership equational theory. It is based on Membership Equational Logic, a good fit for reasoning inductively about functions and data structures involving partiality, subsorts, and conditions. ITP supports proofs by structural induction and complete induction, in which operations need not be completely specified. Goals are either conditional equations or conditional memberships, and inference steps are available through a series of user commands. ITP depends on MTT for checking termination assumptions, on SCC for checking sufficient completeness and freeness of equational constructors, and on CRC for checking the Church-Rosser property of the equations.

The ITP tool, documentation, and some examples are available from `http://maude.cs.uiuc.edu/tools/itp/`.

5 Extensibility by Example: the Integration of SCC

In this section we present a brief overview of the steps undergone to integrate SCC in MFE. In order to take as much advantage as possible of the infrastructure offered by MFE, some classes offered by MFE such as `Tool` and `Proof` are specialized with new attributes and behavior specific to SCC. In this way, SCC inherits the behavior defined already for these classes in MFE. Internal messages defining SCC's public interface are created and the rewrite rules defining SCC's behavior are updated so they fit into MFE's message passing interaction mechanism. Also, the controller object is modified to account for SCC in the object-based configuration of the formal environment.

5.1 SCC Proof Objects

An SCC proof object is an instance of class `SCCProof`, which is a subclass of `Proof`:

```
class SCCProof | sc-result : SCCResult,
                 terminating : 3Bool,
                 sort-dec : 3Bool,
                 sound : Bool,
                 complete : Bool,
                 trusted : Bool .
subclass SCCProof < Proof .
```

Attribute `sc-result` registers the sufficient completeness result of sort `SCCResult`. If the emptiness test is successful, it holds the value `empty`; if it is unsuccessful, it holds a counter-example for sufficient completeness. A counter-example for sufficient completeness is a ground irreducible term that does not belong to the subsignature of constructors. Sort `Bool` is Maude's predefined sort for Boolean values and operations, and sort `3Bool` is an extension of sort `Bool` with a third "undefined" value.

Ground termination and ground sort-decreasingness are assumed in SCC's stand-alone version. An SCC proof object in MFE registers the ground termination and ground sort-decreasingness status of its associated module in the three-valued attributes `termi-nating` and `sort-dec`, respectively, so that it can be checked whether these assumptions hold (indicated by value `true`), do not hold (indicated by value `false`), or have not been submitted (indicated by value `maybe`). Thanks to the interoperability offered by MFE and as explained in Section **??**, there are tools in MFE that can check for ground termination and ground sort-decreasingness of Maude specifications. Therefore, SCC proof objects can always submit these checks to the corresponding tools.

In some situations, some SCC's requirements such as left-linearity of equations can be relaxed. For instance, if the emptiness check is successful when ignoring the non left-linear equations in a Maude module, then this module is sufficiently complete with all its equations. However, a counter-example to the sufficient completeness in the reduced module is not necessarily a sufficient completeness counter-example for the module with all its equations. SCC offers checks for these two properties and, in MFE, SCC proof object registers the information with Boolean attributes `sound` and `complete`. SCC proof objects define the Boolean valued attribute `trusted` for supporting a "trust" command for sufficient completeness proofs. This is useful for dealing with modules that are not supported by SCC or for sufficient completeness proofs that are obtained outside SCC.

5.2 The SCC Tool Object

The SCC tool object is an instance of class `SCCTool`, which is a subclass of MFE's `Tool`. Class `SCCTool` does not declare any new attributes.

```
  class SCCTool .
  subclass SCCTool < Tool .
```

The functional module `SCC-SIGN` defines the grammar of the user commands supported by the SCC tool object. These are the commands available from SCC's stand-alone version in addition to a new one for showing the state of `SCCProof` objects.

```
fmod SCC-SIGN is
  including FULL-MAUDE-SIGN .
  op scc_. : @ModExp@ -> @Command@ .
  op submit . : -> @Command@ .
  op trust . : -> @Command@ .
  op show state . : -> @Command@ .
  op SCC help . : -> @Command@ .
  ...
endfm
```

The command `(scc` *MN*`.)` checks the sufficient completeness of module with name *MN*. The command `(submit .)` submits the termination and sort-decreasingness proof obligations to the corresponding tools in MFE for the active SCC proof object, if any. The command `(trust .)` trusts the sufficient completeness proof for the active SCC proof object, if any. The command `(show state .)` displays the state of each SCC proof object, and command `(SCC help .)` displays the help menu of Maude's SCC.

Two bodies for internal messages are defined. Namely, a message body for checking sufficient completeness and a message body for acknowledging that a module is sufficiently complete. The latter type of message is only created when the sufficient completeness check has been successful, and the termination and sort-decreasingness assumptions have been checked.

```
  op check sc_ : Module -> MFEMsgBody .
  op module_is sufficiently complete : Module -> MFEMsgBody .
```

Observe that command `(scc _.)` and internal message body `check sc_` refer to the same functionality offered by the SCC tool object, but with different inputs: the former takes a module expression as input, while the latter takes a module as input. Regardless of this typing difference, it is convenient to have exactly one entry point for commands referring to the same functionality. On the one hand it facilitates source code maintainability and debugging. On the other hand, it helps to avoid repetition of significant amounts of source code. To address this issue, the SCC tool object exclusively handles internal messages while it rewrites user commands in parsing messages to internal messages: it amounts to evaluating the module expression given by the user to a module in the database of modules and, if successful, to creating an internal message with the module and with the "from-to" data of the parsing message. There is an additional rule handling the case in which the module expression in the parsing message cannot be evaluated for a module, notifying the failure to the user. These rules correspond to updated versions of previously existing rules that handled user commands in SCC.

Rewrite rule `check-sc` below specifies the creation of an SCC proof object for checking the sufficient completeness of a module *M*. Here, function `processSCCheck` encapsulates the calls to functionality already available from SCC for the sufficient completeness check, and function `createSCCProof` encapsulates the instantiation of the new SCC proof object and updates to the attributes of the SCC tool object.

```
 var X@SCCTool : SCCTool .   vars O O' : Oid .   var M : Module .
 var Atts : AttributeSet .   var MNReg : Map{ModuleName, Oid} .

 crl [check-sc] :
   < O : X@SCCTool | reg : MNReg, Atts > (to O from O' : check sc M)
   => if not isParameterized?(M) and-else not M :: STheory
      then createSCCProof(O', < O : SCCTool | reg : MNReg, Atts >,
             processSCCheck(M))
      else < O : SCCTool | reg : MNReg, Atts >
           (to O' from O : output(
              mfe-error('SCC 'cannot 'check 'parameterized 'modules
                        'or 'theories. '\n)))
      fi   if not getName(M) in domain of MNReg .
```

SCC operates on unparameterized modules with initial semantics. If module *M* conforms to these two constraints, then a new SCC proof object is instantiated with the emptiness result of the corresponding automaton. The registry attribute `reg` of the SCC tool object is updated with the name of module *M* and the unique object identifier of the newly created proof object. In the case that module *M* does not conform to these constraints, an error message is issued to the user, no SCC proof obligation is instantiated, and the SCC tool object remains unchanged (this is done by a rule omitted here).

Dependencies. The SCC tool object depends on termination and sort-decreasingness checks by MTT and CRC tool objects. In general, tool dependencies can be resolved at instantiation, for which the controller object provides information of the tools available in the environment (including itself).

The SCC tool object is instantiated by defining an *instantiation token* that takes a map representing the available tools as input and by a rewrite rule that creates the instance of the tool object with the information provided in the map. The map in the instantiation token identifies a (tool) name *TN* with a (tool) object identifier *O* whenever the tool object for *TN* has object identifier *O*. Observe that this instantiation mechanism with a map as a parameter benefits the modularity and extensibility of MFE: if more tools become available in MFE, there is no need to modify the way tool objects are currently instantiated in MFE.

A new SCC tool object is created by a term `init-scc(`*TS*`)` of sort `Configuration`, with *TS* a term of sort `Map{ToolName, Oid}`. The following rewrite rule `init-scc-to` instantiates the SCC tool object and creates a message to display.

```
op init-scc : Map{ToolName, Oid} -> Configuration .

var TS : Map{ToolName, Oid} .

rl [init-scc-to] :
  init-scc(TS)
  => < TS["SCC"] : SCC |
        tools : TS,
        grammar : SCC-GRAMMAR,
        current : null-oid,
        index : 0,
        reg : empty >
     (to TS["MFE"] from TS["SCC"] : output(string2qidList(scc-banner))) .
```

The tool map *TS* is used to assign the object identifier to the SCC tool object, namely, the object identifier with associated tool name `"SCC"`. Tool names are globally known and the controller is responsible for their uniqueness. Since SCC proof objects do not exists when the SCC tool object is created, attribute `index` is set to value `0` and attribute `reg` is set to value `empty`. As the result of the SCC tool object successful creation, an output message is sent to the controller object with a welcoming message, here encoded by constant term `scc-banner`.

5.3 Making SCC Operational in MFE

In order to make the SCC tool object operational in MFE, it needs to be registered with the controller object and added to the object configuration. A unique tool name and a unique object identifier are required for registering the SCC tool object with the controller object. In MFE, the string `"SCC"` is the global tool name for the SCC tool object and `scc` its object identifier. Therefore, the tools map in the controller object is updated with the pair `"SCC" |-> scc`.

The SCC tool object is added to the initial configuration of MFE by means of the initialization token `init-scc(`*T*`)`, with the updated tools map *T*.

```
op TOOLS+SCC : -> Map{ToolName, Oid} .
eq TOOLS+SCC = (TOOLS, "SCC" |-> scc) .

rl [init] :
  init
  => [ nil,
       < mfe : Controller |
          db : initialDatabase, input : nilTermList, output : nil,
          default : 'CONVERSION, current-tool : mfe, tools : TOOLS+SCC >
       ...
       init-scc(TOOLS+SCC),
       nil ] .
```

6 Using MFE

In this section, we illustrate some features and commands of MFE on the classical example of the readers and writers, using a slightly modified version of the presentation in [**?**, Sections 12.3 and 12.4].[3] In fact, a similar proof can be found in [**?**], but using the tools in isolation and with no support for keeping track of the pending proof obligations.

In this specification, a state is represented by a term `< R, W >` where `R` and `W` are, respectively, the number of readers and writers accessing a critical resource. In the system, there should not be more than one writer, or writers and readers at the same time. To obtain this behavior, a writer can only access the critical resource if no nobody else is using it, and a reader can gain access to the critical resource only if there are no writers using it. Readers and writers can leave the critical resource at any time.

The following modules `MBOOL` and `MNAT` define, respectively, sorts `MBool` and `MNat` of Boolean values and natural numbers.

```
(fmod MBOOL is
   sort MBool .
   ops true false : -> MBool [ctor] .
 endfm)
```

[3] The changes introduced are due to the different treatment of built-ins by the different tools. To avoid conflicts we do not use any built-in in the example. We set the automatic inclusion of the `BOOL` module off and, although not used, the automatic inclusion of module `TRUTH-VALUE` on.

```
(fmod MNAT is
   sort MNat .
   op 0 : -> MNat [ctor] .
   op s : MNat -> MNat [ctor] .
 endfm)
```

The readers-writers system can be specified as follows.

```
(mod READERS-WRITERS is
   protecting MNAT .
   sort Config .
   op <_`,_> : MNat MNat -> Config [ctor] .
   vars R W : MNat .
   rl [wrt+] : < 0, 0 > => < 0, s(0) > .
   rl [wrt-] : < R, s(W) > => < R, W > .
   rl [rdr+] : < R, 0 > => < s(R), 0 > .
   rl [rdr-] : < s(R), W > => < R, W > .
 endm)
```

Before reducing or rewriting any term in this module, we must check the expected execution requirements, namely, the equations being ground terminating and Church-Rosser, and the rules being ground coherent with respect to the equations. We can perform all these checks in MFE. During the verification process, tools keep record of pending proof obligations, are able to submit them to appropriate tools in the environment, and complete their proofs upon reception of messages announcing the discharging of assumptions. In fact, we can choose to complete the proof in different orders. For example, we could check first the termination, then the Church-Rosser property, and then its coherence, or we could choose to directly attempt the coherence proof and let the submission of the proof obligations help in the process.

Let us do first the Church-Rosser proof. To carry such a check we first select the CRC tool.

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

Since there are no equations in `READERS-WRITERS`, this module is trivially (ground) Church-Rosser.

```
Maude> (check Church-Rosser .)
Church-Rosser check for READERS-WRITERS
    There are no critical pairs.
    The specification is confluent.
    The module is sort-decreasing.
    Success: The module is therefore Church-Rosser.
```

We now check module `READERS-WRITERS` ground coherent. We select the ChC tool:

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

and then issue the checking command:

```
Maude> (check coherence .)
Coherence checking of READERS-WRITERS
    All critical pairs have been rewritten and no rewrite with rules
    can happen at non-overlapping positions of equations left-hand sides.
    The termination and Church-Rosser properties must still be checked.
```

The coherence property assumes the ground termination and ground Church-Rosser properties of equations in module `READERS-WRITERS`. We use command `(submit .)` to ask the environment to submit the pending proof obligations to the corresponding tools.

```
Maude> (submit .)
The Church-Rosser goal for READERS-WRITERS has been submitted to CRC.
The termination goal for the functional part of READERS-WRITERS has been
    submitted to MTT.
Success: The functional part of module READERS-WRITERS is terminating.
Church-Rosser check for READERS-WRITERS
    There are no critical pairs.
    The specification is confluent.
    The module is sort-decreasing.
    Success: The module is therefore Church-Rosser.
The functional part of module READERS-WRITERS has been checked terminating.
The module READERS-WRITERS has been checked Church-Rosser.
Success: The module READERS-WRITERS is coherent.
```

The ground termination and Church-Rosser properties of the functional part of module `READERS-WRITERS` are checked, answers to the requests are received, and the coherence checker automatically completes the proof and sends to the controller object the success message. The proof of the Church-Rosser property was already in the environment, and therefore the answer is directly returned without further checking. When requested, the termination proof is attempted and it succeeds. When ChC receives all the messages informing of successful discharging of both proof obligations, it completes the proof and sends the corresponding success message.

We are now ready to prove some properties of module `READERS-WRITERS`. For instance, we verify mutual exclusion in `READERS-WRITERS`, that is, at most one reader or one writer uses the critical resource at a particular time. We could do this by using Maude's `search` command or its LTL model checker. However, since the reachable state space is infinite for any initial state, we first need to define an abstraction of the system and proof its correctness. We use the following predicates and abstraction, in which all states having readers are collapsed to a simpler state (it is not relevant to know how many readers are using the critical resource, but whether there is any or not using the critical resource).

```
(mod READERS-WRITERS-PREDS is
   protecting MBOOL .
   protecting READERS-WRITERS .
   ops mutex one-writer : Config -> MBool [frozen] .
```

```
    vars M N : MNat .
    eq mutex(< s(N), s(M) >) = false .
    eq mutex(< 0, N >) = true .
    eq mutex(< N, 0 >) = true .
    eq one-writer(< N, s(s(M)) >) = true .
    eq one-writer(< N, s(0) >) = true .
    eq one-writer(< N, 0 >) = true .
  endm)

 (mod READERS-WRITERS-ABS is
    including READERS-WRITERS-PREDS .
    var N : MNat .
    eq [abs] : < s(s(N)), 0 > = < s(0), 0 > .
  endm)
```

To check both the execution and the invariant-preservation properties of this abstraction, we need to check: the equations being ground confluent, sort-decreasing, and terminating; the equations being sufficiently complete; and the rules being ground coherent with respect the equations.

The use of CRC, ChC, SCC, and MTT to carry on these proofs is presented in [**?**, Section 12.4]. Here we show how the different proofs can be completed inside the environment without the need of switching between execution environments.

We start by checking the sufficient completeness of READERS-WRITERS-ABS:

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

Maude> (scc READERS-WRITERS-ABS .)
Checking sufficient completeness of READERS-WRITERS-ABS ...
To complete the proof the specification must be proved ground
sort-decreasing and weakly-terminating.
```

We then submit these assumptions to other tools in the environment.

```
Maude> (submit .)
The sort-decreasingness goal for READERS-WRITERS-ABS has been submitted
    to CRC.
The termination goal for the functional part of READERS-WRITERS-ABS has
    been submitted to MTT.
Success: Module READERS-WRITERS-ABS is sort-decreasing.
Success: The functional part of module READERS-WRITERS-ABS is terminating.
Success: Module READERS-WRITERS-ABS is sufficiently complete.
```

CRC has not requested a termination proof for READERS-WRITERS-ABS, and therefore has not been informed of the result of the termination proof. Let us just do that. We first select the CRC tool as active tool.

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

By requesting the CRC's, we realize that the check was completed, and that only ground termination is necessary to complete the ground confluence proof and, with it, obtain a ground Church-Rosser proof for `READERS-WRITERS-ABS`.

```
Maude> (show state .)
State of the tool:
- Church-Rosser check for READERS-WRITERS :
    There are no critical pairs.
    The specification is confluent.
    The module is sort-decreasing.
    The module is therefore Church-Rosser.
- Church-Rosser check for READERS-WRITERS-ABS :
    All critical pairs have been joined.
    The specification is locally-confluent.
    The module is sort-decreasing.
```

We submit the pending proof obligation.

```
Maude> (submit .)
The termination goal for the functional part of READERS-WRITERS-ABS has
    been  submitted to MTT.
The functional part of module READERS-WRITERS-ABS has been checked
    terminating.
Success: The module READERS-WRITERS-ABS has been checked Church-Rosser.
```

Then, we check module `READERS-WRITERS-ABS` ground coherent.

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

Maude> (check ground coherence READERS-WRITERS-ABS .)
Ground coherence checking of READERS-WRITERS-ABS
The following critical pairs cannot be rewritten:
  cp READERS-WRITERS-ABS1 for abs and rdr-
    < s(0),0 >
    => < s(#1:MNat), 0 > .
The termination and Church-Rosser properties must still be checked.
```

A critical pair is returned by the ChC tool in the form of a reachability goal. Also, ground termination and Church-Rosser proofs must be obtained. Some of these proofs have already been found, but ChC has not been informed. We submit the pending proof obligations.

```
Maude> (submit .)
The Church-Rosser goal for READERS-WRITERS-ABS has been submitted to CRC.
The goal for critical pair READERS-WRITERS-ABS1 has been submitted to ITP.
The termination goal for the functional part of READERS-WRITERS-ABS has
    been submitted to MTT.
The module READERS-WRITERS-ABS has been checked Church-Rosser.
The functional part of module READERS-WRITERS-ABS has been checked
    terminating.
```

The ITP does not provide methods to prove the joinability of critical pairs. However, we can carry on a proof by reasoning by cases and using Maude's searching command as in [**?**, Section 12.4]. We can then use the `(trust .)` command to inform the tool that the proof was completed out of the ITP.

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

Maude> (trust .)
Eliminated current goal.
The critical pair READERS-WRITERS-ABS1 has been trusted.
Success: The module READERS-WRITERS-ABS is ground-coherent.
```

Now that the abstraction has been proved correct, we can check both invariants:

```
Maude> (search in READERS-WRITERS-ABS :
          < 0, 0 > =>* C:Config
          such that mutex(C:Config) = false .)
No solution.

Maude> (search in READERS-WRITERS-ABS :
          < 0, 0 > =>* C:Config
          such that one-writer(C:Config) = false .)
No solution.
```

7 Conclusion

The Maude Formal Environment is an executable and highly extensible software infrastructure written in Maude within which a user can interact with several tools to mechanically verify properties of Maude specifications. MFE exploits Maude as a reflective declarative language and system based on rewriting logic in which computation corresponds to efficient deduction by rewriting. We have explained the main design decisions in MFE and integrated, as a proof of concept, five important formal analysis tools with highly heterogeneous designs: namely, Maude's Termination Tool, Church-Rosser Checker, Coherence Checker, Sufficient Completeness Checker, and Inductive Theorem Prover. We also presented a brief overview of the steps underwent to integrate Maude's SCC in MFE and explained how MFE's design decisions allowed for an easy integration. It is important to highlight that the approach taken here for extending MFE with the SCC is one of many possible ways to benefit from the software infrastructure offered by MFE. Finally, we gave a fair overview of some of the features and commands of MFE on the classical example of the readers and writers.

Much work remains ahead. First of all, more tools such as Maude's LTL and LTLR Model Checkers, Maude's Invariant Analyzer Tool, and Real-Time Maude could be integrated in MFE. This will result in a more interesting environment with features for handling broader applications with less effort for the user. One could also think of handling proof obligations such as those for the protecting and extending importations of modules, for the instantiation of parameterized modules, or simply the termination and

Church-Rosser assumptions for equational simplification. More ambitiously, a graphical user interface and support for better interoperability will enhance the user experience with the formal environment. The graphical user interface could be developed, for instance, as a plugin in the Eclipse environment. IMaude and the IOP platform might also be a good candidates for improving tool interoperability and providing the environment with a graphical user interface.

Acknowledgements. The authors would like to thank the anonymous referees for comments that helped to improved the paper. The first author has been partially supported by Spanish Research Projects TIN2008-03107 and P07-TIC-03184. The second author has been partially supported by NSF grants CNS 07-16638 and CCF 09-05584.