A Guide to Extending Full Maude Illustrated with the Implementation of Real-Time Maude

Francisco Durán\textsuperscript{1,3}

\textit{Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga}

Peter Csaba Ölveczky\textsuperscript{2,4}

\textit{Department of Informatics, University of Oslo}

Abstract

The goal of this paper is to serve as a practical guide for implementing extensions of Maude by giving an overview of how the Real-Time Maude tool has been developed by extending the implementation of Full Maude. After giving a high-level summary of the key functionality and structure of the implementation of Full Maude, we describe the implementation of the Real-Time Maude language and tool. This extension includes key issues such as adding new kinds of modules, rules, and commands; as well as the need to store additional information in the persistent state of the execution environment.

1 Introduction

The success of Maude [4] in modeling and analyzing concurrent systems has inspired, and will continue to inspire, extensions to different kinds of systems (such as real-time [23], probabilistic [1], (stochastic) hybrid systems, etc.), as well as new analysis techniques for ordinary and extended Maude specifications (such as inductive theorem proving [7], narrowing analysis, TLR model checking, timed analysis, probabilistic analysis, etc.). The goal of this paper is to serve as a practical guide for implementing extensions of Maude by showing how the Real-Time Maude tool [23] has been developed by extending the implementation of Full Maude [4].

The Maude system provides powerful meta-programming facilities that allow us to develop execution environments for a wide range of languages and logics with much less effort than using conventional programming languages [5]. An early use

\textsuperscript{1} Francisco Durán was partially supported by the EU (FEDER) and the Spanish MEC, under grant TIN2005-09405-C02-01.

\textsuperscript{2} Peter Ölveczky was partially supported by The Research Council of Norway.

\textsuperscript{3} Email: duran@lcc.uma.es

\textsuperscript{4} Email: peterol@ifi.uio.no

This paper is electronically published in

\textit{Electronic Notes in Theoretical Computer Science}

URL: www.elsevier.nl/locate/entcs
of these facilities was the implementation of Full Maude, a language that extends Maude with support for object-oriented specification and advanced module operations. The implementation of Full Maude includes code for parsing user input and pretty-printing; storing modules, theories, and views; transforming object-oriented modules into system modules; and so on. Therefore, you essentially have two choices for implementing in Maude an extension of Maude:

(i) Doing it all from scratch.

(ii) Extending the implementation of Full Maude, taking advantage of the infrastructure provided.

Another significant early extension of Maude was to add support for the formal specification and analysis of real-time systems. The second author initially started to implement the Real-Time Maude tool from scratch in Maude. However, it soon became apparent that:

• It would require a lot of work to incorporate useful features, including the crucial issue of support for object-oriented specification.

• The implementation of Real-Time Maude would end up including, in essence, a re-implementation of Full Maude.

The second author therefore abandoned this effort, and decided to implement Real-Time Maude by extending the implementation of Full Maude. Although it is far from tempting to try to understand, modify, and extend Full Maude’s more than 13,000 lines of meta-level Maude code, this choice has clearly shown to have been the right choice. Real-Time Maude is now a mature tool that has been successfully applied to a wide range of challenging applications (see, e.g., [24,25,22,18]).

Our belief that extensions of Maude can be conveniently built by extending the implementation of Full Maude has been underscored by the more recent development of tools that have followed this approach. They include: the Church-Rosser and coherence checkers for Maude [13,9,6], the Maude MSOS tool for modular structural operational semantics [3], the automated circular coinductive prover CiRC [20], the strategy language proposed in [21], and the implementation of membrane systems in [2]. In addition, parts of the infrastructure provided by the implementation of Full Maude has been exploited to implement tools also for formalisms that are not extensions or variations of Maude, such as the LOTOS tool by Verdejo [26]. It is worth mentioning that there are also formal tools for Maude, written in Maude, that are not implemented on top of Full Maude, such as the ITP [7] and the SCC [17].

Having developed Full Maude and some of its extensions, we have repeatedly been asked how to extend Full Maude. This paper attempts to give a practical answer to that question by summarizing our experiences in developing such extensions (Section 2), by giving an overview of the implementation of Full Maude (Section 4), and by showing how Real-Time Maude has been implemented (Section 5). We do not intend to give a general methodology on extending Full Maude, but provide a guide based on our experience. The choice of Real-Time Maude is motivated by the fact that it is a mature and significant extension of Full Maude that includes additional module syntax and many new analysis commands. This paper assumes familiarity with the Maude language, including the features of its meta-level.
1.1 Related Work

In [14], Durán and Meseguer show in detail how Full Maude can be extended with new module expressions, and in the papers [11,10] it is shown how new evaluation strategies can be added to Full Maude. How to add new commands and module expressions to Full Maude is also explained in [4, Chapter 18]. This paper differs from those papers in the following ways:

- We describe the implementation not of a single new command or module expression, but of an entire mature tool that significantly extends Full Maude.
- We give an overview of the structure of the implementation of Full Maude, which we hope will make it easy for a Maude extender to get an overview of what to do.
- This paper discusses experiences with extending Full Maude.
- At the more technical level, aspects such as adding new attributes to the persistent state and, in particular, extending Full Maude with new kinds of modules (timed modules) have never been explained before.

In this volume, Goriac, Caltais, Lucanu, Andrei, and Grigoras provide a set of basic patterns which may be used in Maude metalanguage applications [16].

2 Our Experiences in Extending Full Maude

This section shares some of the experiences gained by the authors while implementing and using Real-Time Maude and other extensions of Full Maude, and gives some suggestions on how to approach the task of extending the latter.

2.1 Extending the Implementation of Full Maude?

The first question we are usually asked by colleagues who wish to extend Maude is: “Do I have to extend the implementation of Full Maude?” Reasons for not wanting to do so may include:

(i) The above mentioned reluctance toward trying to understand the huge, complex, and barely documented implementation of Full Maude.
(ii) Doubts about the user-friendliness and robustness of Full Maude.
(iii) The effort must be redone for each new version of Full Maude.

The second author has implemented Real-Time Maude both “directly” in Maude and as an extension of the implementation of Full Maude. The direct implementation did not provide any support for object-oriented specification and analysis\(^5\), but made the second author appreciate the huge task of implementing an extension directly in Maude. Therefore, if you want your extension to retain the ability to use Maude’s flexible syntax, the answer to the question above is: “I would certainly extend the Full Maude implementation.” If, in addition, you want to retain Full Maude’s support for object-oriented specification, the answer is: “Yes, you have no choice.” Otherwise, you would in the best case essentially develop your own

\(^5\) This is a significant omission, since all large Real-Time Maude applications have been specified in an object-oriented style.
implementation of Full Maude on your way to the final goal. This view is also in accordance with the fact that Maude tools that are not implemented on top of Full Maude typically only support restricted fragments of Maude specifications that do not include object-oriented modules.

As for the reasons for not wanting to extend Full Maude: Regarding (i), the implementation of Full Maude is indeed hard to grasp. It is the aim of this paper to give an overview of that implementation and provide a good starting point for extending it. Detailed information on it is provided in [12,8].

Regarding (ii), one main problem with Full Maude has traditionally been its lack of robustness. Small errors in specification or command often resulted in a blank line and the need to restart the whole Full Maude session without getting any information about the source of the error. However, Full Maude makes it possible to make your implementation quite robust and to provide good diagnostic error messages without ending your Maude session, as shown in this paper.

Regarding (iii), we show in this paper that an extension of Full Maude can be implemented in a modular way by just appending code to its implementation. The second author’s experience is that the first author is now very keenly aware of the fact that other advanced tools extend the implementation of Full Maude, and that he takes great care in updating Full Maude so that updating the extensions should cause as little trouble as possible. This impression is supported by the fact that it took the second author almost no time to update Real-Time Maude from extending version 2.2 of Full Maude to extending its version 2.3.

2.2 Some Suggestions

2.2.1 Use Maude’s “Execution Commands” as Much as Possible

It is no surprise that there is a substantial performance penalty to pay for interpreting an analysis command by repeatedly calling basic descent functions such as metaApply. It is much more efficient to do a fair amount of “pre-processing” and then call a function such as metaRewrite or metaSearch (or Maude’s built-in LTL model checker) to do most of the work. In contrast to the first version of Real-Time Maude, the current implementation of the tool puts great emphasis on achieving performance on par with Maude by translating, when possible, a pair, consisting of a timed module $M$ and a timed command $C$, into a pair $(\tilde{M}, \tilde{C})$ of a Maude module $\tilde{M}$ and Maude command $\tilde{C}$ as described in [23].

2.2.2 Write a Small Maude Interpreter from Scratch

To gain invaluable understanding of the different aspects of writing an interpreter, and of the Full Maude implementation, we have found it a very useful exercise to write from scratch an interpreter for a very small subset of Maude, for example using the fragment and techniques in [4, Sections 17.1 to 17.4].

2.2.3 Reuse and Exploit Full Maude as Much as Possible

Trying to completely understand the entire specification of Full Maude is clearly not a good idea. Considering those pieces that deal with features similar to the ones we are concerned about is sufficient in most cases, as we show in this paper.
3 Overview of Real-Time Maude

Real-Time Maude [23] is a mature tool that extends Full Maude to support the formal specification and analysis of real-time systems. The tool has been successfully applied to a diverse set of advanced state-of-the-art systems [24,25,22,18,19], and has been particularly useful for specifying real-time systems in an object-oriented style.

A timed module (syntax \texttt{tmod ... endtm}) specifies a real-time system. Data types and instantaneous (i.e., zero-time) state changes are specified, as in Maude, by, respectively, a membership equational logic specification and a set of rewrite rules. Time advance is modeled by tick rules, which have the form
\[
\text{crl } [l] : \{t\} \Rightarrow \{u\} \text{ in time } \tau \text{ if } \text{cond} .
\]
for \(\tau\) a term of sort \texttt{Time}. Object-based specifications can be defined as timed object-oriented modules (syntax \texttt{tomod ... endtom}) that extend Full Maude’s object-oriented modules. To cover a dense time domain, tick rules typically have the form
\[
\text{crl } [\text{tick}] : \{t\} \Rightarrow \{u\} \text{ in time } T \text{ if } T \leq \text{mte}(t) \text{ [nonexec]} .
\]
where \(T\) is a variable that does not occur in \(t\). Real-Time Maude provides a choice of a set of time sampling strategies to guide the application of such tick rules [23].

Real-Time Maude extends Maude’s analysis commands to the timed setting and provides additional timed commands. The timed rewrite command has syntax \(^6\)
\[
\text{(trew } [\{n\}] \text{ in module :} \text{ } t_0 \text{ in time } \leq \text{ } \tau \text{ )}
\]
and simulates a behavior of the system up to duration \(\tau\). The tool also offers different forms of timed and untimed reachability and LTL model checking analysis, as well as time-specific analysis commands, such as finding the least and most time it takes to reach a desired state.

Real-Time Maude therefore extends Full Maude by providing: a library of pre-defined modules (e.g., for time domains) that should be available for importation by user-defined modules; new kinds of modules and rules (tick rules); and additional analysis commands. Furthermore, the tool must store certain additional data, such as the current time sampling strategy, in its persistent state.

Although a timed module can be transformed into many different Maude modules, depending on the command to be executed, there is a basic “clocked” transformation in which a timed module becomes an ordinary Maude module by just importing the following module \texttt{TIMED-PRELUDE} [23]:
\[
\text{fmod TIMED-PRELUDE is including TIME .}
\]
\text{sorts System GlobalSystem ClockedSystem .}
\text{subsort GlobalSystem < ClockedSystem .}
\text{op } \{\_\} : \text{System } \rightarrow \text{GlobalSystem [format}(g \text{ o g so})\].
\text{op } \_\text{in time}_\_ : \text{GlobalSystem Time } \rightarrow \text{ClockedSystem [format}(o \text{ g g y o})\].
\text{eq (CLS:ClockedSystem in time } R:\text{Time}) \text{ in time } R':\text{Time}
\text{ = CLS:ClockedSystem in time (R:Time plus R':Time) .}
\]
\(^6\) Optional parts are enclosed by ‘[’ and ‘]’.
Likewise, a timed object-oriented module has a basic transformation into a Maude module that imports the module \texttt{TIMED-OO-PRELUDE}.

4 Overview of the Implementation of Full Maude

This section gives an overview of some of the main functions and modules in the implementation of Full Maude.

Given any input enclosed in parentheses, the read-eval-print loop provided by the \texttt{LOOP-MODE} module gives us a list of quoted identifiers $I$ by putting a quote in front of each of the identifiers in the input. Similarly, any list of quoted identifiers placed in the “output channel” of the loop will be printed to the terminal after removing the quotes. Calling the function \texttt{metaParse} with the input $I$ and the meta-representation of the signature \texttt{Grammar} in which we want to parse it, if there is a parse, we get the corresponding parse tree as a term $T_I$ of sort \texttt{Term}. The reverse process is accomplished by the \texttt{metaPrettyPrint} function, which takes a term of sort \texttt{Term}, and returns its representation as a list of quoted identifiers. We could manipulate this term $T_I$ directly. However, it is simpler and more appropriate to transform this term into another term in some data types \texttt{Module}, \texttt{Command}, etc.

Of course, we not only want to give some command or module expression with some arguments and get some result. We also want to interact with the system, entering modules, theories, views, and commands of different types. Therefore, we need to be able to store modules, theories, and views in a database, so that they can be referred later to evaluate module expressions and commands. The specification of Full Maude includes a data type \texttt{Database} for the database of modules, theories, and views. If not explicitly given, most commands are supposed to be executed on some by-default module, usually the last entered module. We need to keep track of such by-default module as part of the persistent state of the system.

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

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

We focus here on three key issues in Full Maude: syntax definition, input handling, and module and command processing.

4.1 The \texttt{FULL-MAUDE-SIGN} Module

To parse some input using the built-in function \texttt{metaParse}, Full Maude needs the meta-representation of the signature in which the input is going to be parsed. Such a grammar is provided by the \texttt{FULL-MAUDE-SIGN} module and its submodules, 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@, @Bubble@, and @Command@, of modules, bubbles\footnote{A bubble is any non-empty list of Maude identifiers. The intuition behind bubbles is that they correspond to a piece of text that can only be parsed once the grammar introduced by the signature of the module is available.}, and commands, respectively, and syntax declarations as

\begin{verbatim}
  op red_ : @Bubble@ -> @Command@ .
  op rew_ : @Bubble@ -> @Command@ .
  op search_=>*_ : @Bubble@ @Bubble@ -> @Command@ .
  op show module_ : @ModExp@ -> @Command@ .
  op mod_is_endm : @Interface@ @SDeclList@ -> @Module@ .
  op omod_is_endom : @Interface@ @ODeclList@ -> @Module@ .
\end{verbatim}

for the red, rew, search, and show module commands and for system and object-oriented modules. The syntax for, e.g., the rew command is more complex:

\begin{verbatim}
  rewrite [(\langle Nat\rangle)] in \langle ModId \rangle : | \langle Term \rangle .
\end{verbatim}

The syntax for the optional parts will be given when solving the bubbles.

In the META-FULL-MAUDE-SIGN module a constant GRAMMAR of sort FModule is defined as the meta-representation of a module in which there is a declaration importing the FULL-MAUDE-SIGN module. Declarations for the constructors of the bubble sorts are also included in this module, in a constant BUBBLES which will be useful for parsing bubbles later on.

fmod META-FULL-MAUDE-SIGN is
  including UNIT .
  ops BUBBLES GRAMMAR : -> FModule [memo] .
  eq BUBBLES = ... .
  eq GRAMMAR = addImports((including 'FULL-MAUDE-SIGN .), BUBBLES) .
endfm

4.2 The FULL-MAUDE Module

The top module in Full Maude is FULL-MAUDE, which provides the rules to initialize the loop, and to specify the communication between the loop—the input/output of the system—and the database.

The initial state of the persistent state is defined by the constant init, which will start a loop, with an object of class DatabaseClass as state. The following init rule initializes the persistent state of Full Maude:

\begin{verbatim}
  rl [init] :
    init
    => [nil,
        < o : Database |
        db : initialDatabase, input : nilTermList,
        output : nil,
        default : 'CONVERSION >,
        ('\n \s 's 's 's 's 's 's string2qidList(banner) '\n)] .
\end{verbatim}

The initialDatabase operator represents the initial state of the database. The banner constant is a string with the welcome message.

When some text has been introduced in the loop, the first argument of the operator [_,_,_,_] is different from nil, and we can use this fact to activate the
in rule below, that enters an input such as a module or a command from the user into the database. If the input is syntactically valid w.r.t. GRAMMAR, the parsed input is placed in the input attribute of the DatabaseClass object and is further treated as defined in the DATABASE-HANDLING module (see Section 4.3); otherwise, an error message is placed in the output channel of the loop. (Of course, the input may be syntactically valid w.r.t. GRAMMAR, but further processing—for example, of bubbles—may reveal syntax errors or semantic inconsistencies.)

```plaintext
rl1 [in] :
  [QI QIL,
   < O : X@DatabaseClass |
    db : DB, input : nilTermList, output : nil, default : ME, Atts >,
   QIL']
=> if metaParse(GRAMMAR, QI QIL, '@Input@') :: ResultPair
  then [nil,
       < O : X@DatabaseClass | db : DB,
        input : getTerm(metaParse(GRAMMAR, QI QIL, '@Input@)),
        output : nil, default : ME, Atts >,
       QIL']
  else [nil,
       < O : X@DatabaseClass | db : DB, input : nilTermList,
        output : ('\r 'Warning:
         printSyntaxError(
          metaParse(GRAMMAR, QI QIL, '@Input@), QI QIL)
          '\n 'Error: '\o 'No 'parse 'for 'input. '\n'),
        default : ME, Atts >,
       QIL']
  fi .
```

When the output attribute of the persistent object contains a nonempty list of quoted identifiers, the out rule moves it to the third argument of the loop.

4.3 The DATABASE-HANDLING Module

The DATABASE-HANDLING module contains definitions that do the following:

(i) Define Full Maude’s persistent database as an object of a class DatabaseClass.

(ii) Check the input attribute of the DatabaseClass object, and decide what to do with it. If that input represents an analysis command, a rule calls an appropriate function (such as procCommand) to put the result in its output field. If the input represents a new module, the database is updated (by calling an appropriate function such as procModule), and so is the attribute default, to reflect that the introduced module is now the “current” module.

The rules in the DATABASE-HANDLING module define the behavior of the system for the different commands, modules, theories, and views entered into the system. For example, the following module rule processes the different types of modules entered in the system:

```plaintext
crl [module] :
  < O : X@DatabaseClass |
   db : DB, input : (F[T, T']), output : nil, default : ME, Atts >
=> < O : X@DatabaseClass |
```

The condition of the rule checks the top operator of the parse tree. By the application of the module rule, the state of the DatabaseClass object is changed as follows: (1) The current state of the database is replaced with the result of processing the input, a module in this case, which is handled by the procModule function; (2) the input term is removed from the input attribute; (3) a message informing on the input of the module is placed in the output attribute (the out rule in the FULL-MAUDE module will pass the message to the loop’s output channel); and (4) the newly entered module becomes the default module.

The parsing accomplished in the in rule only deals with the top-level syntax, the input can still contain errors. Some of these errors may be detected when the different declarations in the module are analyzed, but others will have to wait until the signature is completed and the bubbles can be processed. To report on these errors, one of the components of the Database constructor will keep such messages. The DATABASE-HANDLING module includes the following rule, which takes a message (a QidList) from the eighth argument of the db operator, and puts it in the output channel of the DatabaseClass object.

The DATABASE-HANDLING module includes the following rule, which takes a message (a QidList) from the eighth argument of the db operator, and puts it in the output channel of the DatabaseClass object.

4.4 The UNIT-PROCESSING Module

The function procModule is used to process a term resulting from parsing input corresponding to a unit (a module or theory). This function takes as arguments the term of sort Term to process, and the database, and returns the updated database:

The procModule function parses one by one each of the declarations in the module. During the processing of modules, the procModule function builds: (1) a module with bubbles, corresponding to the terms in it, which will be later parsed using the signature of the module—such a module with bubbles is usually called a pre-module—`; (2) a module without bubbles, corresponding to the signature of the module; and (3) the set of variables declared in the module, given as constant
declarations, since a module at the metalevel does not include variable declarations other than those declared on the fly, whereas the variable declarations are needed for parsing the bubbles, and possibly for some commands, like the search command.

The pre-module, the signature, and the variable declarations resulting from the processing of a module are then used by the evalPreModule function.

\[
\text{op evalPreModule : Module Module OpDeclSet Database -> Database .}
\]

The evalPreModule function first normalizes the structure of the module, by calling the normalize function, and then all the subunits in the structure are collected. A single flattened module is built with all the subunits in the structure, which is then used to create a first version of the signature in which all the bubbles in the top pre-module are parsed using the solveBubbles function. The final version of the signature and the flat unit are generated once the bubbles have been parsed.

### 4.5 The procCommand Function

In the implementation of Full Maude, there is an equation for procCommand for each command. Thus, adding new commands implies adding new equations for this operator. In this paper, we explain how the rew command is handled in Full Maude, and then we consider the case of the timed rewrite command in Real-Time Maude.

As Maude, Full Maude follows a lazy recompilation of modules scheme. When a submodule of a module is changed, such a module is not deleted, but the results of its compilation are. Thus, before using a module, we must check whether it is compiled or not, and compile it (again) if it is not.

\[
\text{eq procCommand('rew_.['bubble[T]], ME, DB)}
\]

\[
= \begin{cases} 
\text{procRew(ME, getFlatModule(ME, DB), 'bubble[T], unbounded, getVars(ME, DB), DB)} & \text{if compiledModule(ME, DB)} \\
\text{procRew(modExp(evalModExp(ME, DB)), getFlatModule(modExp(evalModExp(ME, DB)), database(evalModExp(ME, DB))), 'bubble[T], unbounded, getVars(modExp(evalModExp(ME, DB)), database(evalModExp(ME, DB))), database(evalModExp(ME, DB)))} & \text{else}
\end{cases}
\]

The first parameter to the rewrite command is a bubble, that is, an unparsed term, since: (1) the rewrite command could have any of the forms \( \text{rew } t, \text{rew } [n] t, \text{rew in } M : t \), and \( \text{rew } [n] \text{ in } M : t \), and (2) before the above question is settled, we do not know in which module the term \( t \) is to be parsed.

The procRew function processes the rewrite command. After parsing the bubbles in it, the metaRewrite function is invoked. The function solveBubblesRew does the heavy lifting of solving the above problems, and returns a tuple consisting of: the module in which the command is to be executed, the term to be rewritten parsed in the module above, the bound on the number of rewrites (unbounded if the command was not of the form \( \text{rew } [n] \ldots \)), and finally the set of variables
in the module in which to execute the command. From this tuple \((\text{TMVB})\), we can get the term, module, and bound, and do the actual rewriting by calling the Maude descent function \text{metaRewrite}:

\[
\text{op procRew : ModuleExpression Module Term Bound OpDeclSet Database -> QidList .}
\]

\[
\text{op solveBubblesRew : Term Module Bool Bound OpDeclSet Database -> [Tuple<Term|Module|OpDeclSet|Bound>].}
\]

\[
\text{ceq procRew}(ME, M, T, D, VDS, DB) = \text{if RP :: ResultPair then (...} \text{\'b \ '}\text{\result} \text{\'}}\text{\s \'}\text{\s eMetaPrettyPrint(getType(RP))} \text{\'}}\text{\s \'}\text{\b \'}:\text{\n \'}\text{\o \'}\text{\n \'}\text{\s eMetaPrettyPrint(getModule(TMVB), getTerm(RP))} \text{\'}}\text{\n) else getMsg(getTerm(TMVB)) fi}
\]

\[
\text{if TMVB := solveBubblesRew(T, M, included('META-MODULE, getImports(getTopModule(ME, DB)), DB), D, VDS, DB) /\ RP := metaRewrite(getModule(TMVB), getTerm(TMVB), getBound(TMVB)) .}
\]

If the call to \text{metaRewrite} went well, the result \(RP\) is a term of sort \text{ResultPair}, and the resulting term and its least sort are returned. Additional equations specify what it does in the case of error.

## 5 The Implementation of Real-Time Maude

This section describes the implementation of Real-Time Maude in order to give useful guidelines for extending Full Maude. In particular, we explain how to implement central tasks in any extension of Full Maude: defining the user-level syntax of a language (Section 5.3); defining “built-in” modules available to the user (Section 5.2); reading and executing user input, etc. We show how to meta-represent timed modules as terms of a new sort \text{TModule}, and how to execute the timed rewrite command. Section 5.5 explains how the persistent state of Full Maude is extended.

### 5.1 Overall Structure of the Real-Time Maude Implementation

The implementation of Real-Time Maude extends the implementation of Full Maude by importing modules from the latter. This way of extending Full Maude allows you to easily upgrade your system each time Full Maude is updated. Figure 5.1 gives an overview of some of the most significant modules (and their inclusions) and functions in the implementation of Full Maude and Real-Time Maude. The modules defining the Real-Time Maude extension are given in shaded boxes.

### 5.2 Predefined Modules

The “predefined” Real-Time Maude modules, as well as some other modules used by the system (such as \text{TIMED-PRELUDE}), are defined as ordinary Maude modules

---

\[\text{\# The third argument to solveBubblesRew checks whether meta-level modules are imported in the module to be executed, since such modules add special functionality that must get specific treatment.}\]
5.3 Defining the Syntax of Real-Time Maude

The next step is to define the syntax of Real-Time Maude modules and commands. Since Real-Time Maude is a proper extension of Full Maude, we just extend Full

in the implementation of Real-Time Maude. Since Full Maude (and, hence, Real-Time Maude) can access (Core) Maude modules, no further action is needed to make these modules available to the Real-Time Maude user.
Maude’s module and command syntax defined in the FULL-MAUDE-SIGN module.

The only additional module syntax in Real-Time Maude are timed modules and timed object-oriented modules. Following the definition of the syntax of Full Maude (and without necessarily trying to understand the sorts @Interface@, @FDeclList@, etc.), the syntax of Real-Time Maude modules and theories is defined as follows:

```maude
fmod TIMED-MODULE-SYNTAX is
  including FULL-MAUDE-SIGN .
  op tmod_is_endtm : @Interface@ @SDeclList@ -> @Module@ .
  op tth_is_endtth : @Interface@ @SDeclList@ -> @Module@ .
  op tomod_is_endtom : @Interface@ @ODeclList@ -> @Module@ .
  op toth_is_endtoth : @Interface@ @ODeclList@ -> @Module@ .
endfm
```

To define the syntax of the Real-Time Maude commands, we again consult the definition of the Full Maude syntax, and extend FULL-MAUDE-SIGN as follows:

```maude
fmod RTM-COMMAND-SYNTAX is
  including FULL-MAUDE-SIGN .
  op trew_in time <=_. : @Bubble@ @Bubble@ -> @Command@ . --- timed rew
  op tsearch_=>*_in time <=_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
  op find latest_=>*_in time <=_. : @Bubble@ @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
  op set tick max def_. : @Bubble@ -> @Command@ . --- set time sampling
endfm
```

The syntax of Real-Time Maude is then summarized in the following module:

```maude
fmod REAL-TIME-MAUDE-SYNTAX is
  inc TIMED-MODULE-SYNTAX .
  inc RTM-COMMAND-SYNTAX .
endfm
```

The following module introduces the TIMED-GRAMMAR constant, which extends the Full Maude GRAMMAR defined in the META-FULL-MAUDE-SIGN module:

```maude
fmod META-RTM-SIGN is
  inc META-FULL-MAUDE-SIGN .
  op TIMED-GRAMMAR : -> FModule [memo] .
  eq TIMED-GRAMMAR
    = addImports((including 'REAL-TIME-MAUDE-SYNTAX .), GRAMMAR) .
endfm
```

This constant TIMED-GRAMMAR defines the grammar in which the user input is parsed.

### 5.4 The REAL-TIME-MAUDE Module

Real-Time Maude’s REAL-TIME-MAUDE top module is as the FULL-MAUDE module with a few changes: First of all, the rule init prints a Real-Time Maude greeting and initialize the persistent state to an object of the subclass TimedDatabaseClass, which adds a new attribute timedData to Full Maude’s DatabaseClass:

```maude
rl [init] :
  init
=>
```

13
5.5 Database Handling

Real-Time Maude’s TIMED-DATABASE-HANDLING module extends Full Maude’s DATABASE-HANDLING module to parse and process timed modules and commands.

5.5.1 Extending the Database

Real-Time Maude extends Full Maude’s persistent database by declaring a subclass TimedDatabaseClass of the class DatabaseClass defined in the Full Maude implementation. This class has an additional attribute timedData, which stores all real-time-specific persistent data.

```
sort TimedDatabaseClass.
subsort TimedDatabaseClass < DatabaseClass.

op TimedDatabase : -> TimedDatabaseClass [ctor].
op timedData :_ : TimedData -> Attribute [ctor].
```

A term of sort TimedData is a pair consisting of the names of all timed modules, and the current time sampling strategy.

5.5.2 Parse and Store Timed Modules

Real-Time Maude also uses the procModule function to process timed modules, as explained in Section 5.6. The corresponding rule in TIMED-DATABASE-HANDLING therefore looks very much like the treatment of module declarations in the Full Maude implementation, and is not shown.

5.5.3 Executing Timed Analysis Commands

Full Maude treats an analysis command by calling the function procCommand, with the command, the name of current module, and the database as arguments. We follow this approach, and execute timed analysis commands by calling the function procTimedCommand, with the time sampling strategy as an additional argument:

```
crl [timedExecution] :
< O : TIMEDDATABASE | db : DB, input : (F[TL]), output : QIL,
  default : ME, timedData : TIMEDDATA, ATTS >
=>
< O : TIMEDDATABASE | db : DB, input : nilTermList,
  output : procTimedCommand(F[TL], ME, DB,
    getTickMode(TIMEDDATA)),
  default : ME, timedData : TIMEDDATA, ATTS >
if F == 'trew_in'time'<_. or F == 'trew_in'time'<=_. or
  F == 'tsearch_=>*_in'time'<_. or F == 'tsearch_=>!_in'time'<=_. or
```
The crucial function procTimedCommand is explained in Section 5.7.

5.6 Processing Timed Modules

As described above, the procModule function processes the input corresponding to timed modules and introduces the resulting module into the module database.

Given the number of different types of modules in Maude, the processing done in Full Maude of these modules is generic. In addition to the modules defined in Maude’s metalevel, a generic transform operator (in the EVALUATION module) is invoked in this processing. For example, this function is in charge of transforming an object-oriented module into a system module. This scheme can be used for the new modules in Real-Time Maude.

We first need to define data types to meta-represent the new modules. In Real-Time Maude, we have two new types of modules and two new types of theories: timed system modules and theories, and timed object-oriented modules and theories. The module TIMED-UNIT, which extends UNIT, adds the following declarations:

- Sorts TModule, TTheory, TOModule, and TOTTheory, with the subsort declarations
  
  ```
  subsorts SModule < TModule OModule < TOModule < Module .
  subsorts STheory < TTheory OTheory < TOTTheory < Module .
  ```

- Operators for representing the new types of modules. For instance, the operator

  ```
  op tomod_is_sorts_:________endtom : Header ImportList
    SortSet SubsortDeclSet ClassDeclSet SubclassDeclSet OpDeclSet
    MsgDeclSet MembAxSet EquationSet RuleSet -> TOModule [ ... ] .
  ```

  is declared for representing timed object-oriented modules.

- Equations for the theory function, which checks whether the given term of sort Module corresponds to a theory or not.

- Equations for the get and set functions getName, getImports, ..., getMsgs, setName, setImports, ..., and setMsgs for the new types of modules.

- Operators defining empty modules and theories of the new types.

- Equations for the empty operator, which returns an empty module of the same type as its argument.

The TIMED-EVALUATION module adds equations for the transform operator. It uses an auxiliary function tmod2mod which invokes the omod2mod function in Full Maude to complete the transformation of object-oriented modules. The TIMED-EVALUATION module includes the following declarations:

```
ceq transform(U, DB) = tmod2mod(U, DB)
  if not U :: OModule /
    not U :: OTheory
  /
  U :: TModule or U :: TTheory or U :: TOModule or U :: TOTTheory .

op tmod2mod : Module Database -> Module .
eq tmod2mod(tmod H is IL sorts SS . SSDS OPDS MAS EqS R1S endtm, DB)
  = (mod H is IL sorts SS . SSDS OPDS MAS EqS R1S endm) .
eq tmod2mod(
    tomod H is IL sorts SS . SSDS CDS SCDS OPDS MDS MAS EqS R1S endtom,
```
There are only three other operators for which equations dealing with the new types of modules need to be considered:

- The `applyMapsToModuleAux` function, from the `VIEW-MAP-SET-APPL-ON-UNIT` module, applies renaming maps to a module.
- The `TIMED-UNIT-META-PRETTY-PRINT` module defines how the new modules and theories are meta-pretty-printed.
- The `procModule` function processes the term resulting from parsing the user input. In this processing, a module of the right type is created, and one by one each declaration in the module is processed and added to it. In order to get the signature in which to parse the bubbles, additional modules need to be specified. This is the case of the `CONFIGURATION` module in object-oriented modules, and of `TIMED-PRELUDE` and `TIMED-OO-PRELUDE` in timed modules. The `procModule` function proceeds by calling successive auxiliary functions which complete the different tasks described above. The function `procModule2` is particularly relevant for our purpose, since it creates an empty module of the right type. Since this module will be later used for parsing the pending bubbles, any module to be imported, such as `TIMED-PRELUDE` for timed modules, is added. Thus, the module `TIMED-UNIT-PROCESSING` adds equations to the `procModule2` operators to appropriately consider the new types of modules:

\[
\text{eq procModule2}(T, 'tmod_is_endtm[T', T''], DB) = \text{procModule3}(T, T', T'', \\
\quad \text{addImports}((\text{including 'TIMED-PRELUDE .}), \text{emptyTModule}), \\
\quad \text{DB}).
\]

\[
\text{eq procModule2}(T, 'tomod_is_endtom[T', T''], DB) = \text{procModule3}(T, T', T'', \\
\quad \text{addImports}((\text{including 'TIMED-OO-PRELUDE ., including 'CONFIGURATION+ .}), \text{emptyTOModule}), \\
\quad \text{DB})).
\]

### 5.7 Processing the Timed Analysis Commands

We explain how the `procTimedCommand` function, that executes the timed analysis commands, is defined by showing how it handles the timed rewrite command. Two additional parameters in the timed rewrite command must be taken into account: the time limit and the tick mode. Both of these are at the current stage represented by "bubbles," since we do not know initially in which module to parse them.

The approach is the same as for `rew`. First, recompile the current module if necessary, then use `solveBubblesRew` to extract the module in which the command is to be executed, the term representing the initial state, and the bound on the number of rewrites. In addition, we must parse the bubbles representing the time limit and the current tick mode in the module we just found. When all this is done, the command is executed by calling the function `timedMetaRewrite`, which is the timed version of the function `metaRewrite` and is defined as an ordinary Maude
Durán and Ölveczky

function on Maude meta-modules and meta-terms.

If necessary, the current module is compiled before invoking the
processTimedRewrite function with the appropriate arguments:

\[
\text{op procTimedCommand : Term ModuleExpression Database TickMode \rightarrow QidList .}
\]

\[
\text{eq procTimedCommand('trew_in'time'<=_.[T, T'], ME, DB, TiM) = if compiledModule(ME, DB) then processTimedRewrite(ME, getFlatModule(ME, DB), unbounded, getVars(ME, DB), DB, T, T', TiM) else processTimedRewrite(..., T', TiM) fi .}
\]

\[
\text{op processTimedRewrite : ModuleExpression Module Bound OpDeclSet Database Term Term TickMode \rightarrow QidList .}
\]

The timed rewrite command is executed by calling timedMetaRewrite with the
result of parsing the arguments of the command:

\[
\text{ceq processTimedRewrite(ME, M, D, VDS, DB, T, T', TiM) = if RP :: ResultPair then (... 'Result \'o eMetaPrettyPrint(getType(RP)) \' \n \s eMetaPrettyPrint(MOD, getTerm(RP)) \'\n) else ('\n 'r Error in timed rewrite. \'o \n) fi if B := included('META-MODULE, getImports(getTopModule(ME, DB)), DB) /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T, M, B, D, VDS, DB) /\ LIMIT := solveBubbles(T', MOD, B, getVars(getName(MOD), DB), DB) /\ TICKMODE := solveTickMode(TiM, MOD, B, getVars(getName(MOD), DB), DB) /\ RP := timedMetaRewrite(MOD, TERM, BOUND, le, LIMIT, TICKMODE) .}
\]

The function solveTickMode uses Full Maude function solveBubbles, which parses
a single “bubble” in a module, to resolve the bubble in the time sampling strategy.

We now illustrate how we can use the infrastructure provided by the imple-
mentation of Full Maude to make the timed command processing robust. If the
solveBubblesRew invocation encounters some problem in parsing the module, term
to be rewritten, or bound, an error message is returned:

\[
\text{ceq processTimedRewrite(ME, M, D, VDS, DB, T, T', TiM) = ('\n 'r 'Error: 'c 'Module/initterm 'does 'not 'parse. \'o \n) if B := included('META-MODULE, getImports(getTopModule(ME, DB)), DB) /\ not solveBubblesRew(T, M, B, D, VDS, DB) :: Tuple<Term|Module|OpDeclSet|Bound> .}
\]

It can also happen that the bubble T' representing the time limit does not parse in
the module MOD in which the rewrite is to take place:

\[
\text{ceq processTimedRewrite(ME, M, D, VDS, DB, T, T', TiM) = ('\n 'r 'Error: 'c 'Time 'limit 'term 'does 'not 'parse \'o \n) if B := included('META-MODULE, getImports(getTopModule(ME, DB)), DB) /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T, M, B, D, VDS, DB) /\ not solveBubbles(T', MOD, B, getVars(getName(MOD), DB), DB) :: Term .}
\]

In the same way, an error message is given if the bubble TiM representing the time
sampling strategy cannot be parsed in the module MOD.
6 Concluding Remarks

In this paper, we have given a practical guide to significantly extending Full Maude, drawing on our considerable experience in developing different extensions and explaining them to others. We have given a high-level overview of the structure and the main functions and modules of the large and complex implementation of Full Maude. We have illustrated how to extend this implementation by outlining the implementation of the Real-Time Maude tool. Real-Time Maude extends the syntax supported by Full Maude by adding real-time modules and theories, tick rules, and a variety of commands for manipulating and analyzing real-time systems.

We hope that this paper, and its extended version [15], makes the task of extending the implementation of Full Maude somewhat easier, enabling the reader to take full advantage of the infrastructure provided by Full Maude to develop advanced extensions of Maude.

References


