A Guide to Extending Full Maude Illustrated with the Implementation of Real-Time Maude (Extended Version)*

Francisco Durán[1] and Peter Csaba Ölveczky[2]

[1] Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga
[2] Department of Informatics, University of Oslo

January 22, 2008

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 entire 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. We assume familiarity with the Maude language. We do not intend to give a methodology of extension, but rather a simple guideline based in our experience.

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 [26, 27], probabilistic [1, 19], (stochastic) hybrid systems, etc.), as well as new kinds of analysis techniques of ordinary and extended Maude specifications (such as inductive theorem proving [3], 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 giving an overview of how the Real-Time Maude tool [27] 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 [6]. An early use 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:

*Francisco Durán was partially supported by the EU (FEDER) and the Spanish MEC, under grant TIN2005-09405-C02-01

1. Doing it all from scratch.

2. 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 chose 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 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 [28, 29, 20, 24, 25].

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 [5, 14, 10, 7], the Maude MSOS tool for modular structural operational semantics [2], the automated circular coinductive prover CiRC [22, 21], and the strategy language proposed in [23]. 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 [30]. 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 [8] 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). 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.

Related work. In [15], Durán and Meseguer show in detail how Full Maude can be extended with new module expressions and to enter such new modules into Full Maude's database, and in the papers [12, 11] 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 of not such a single new command or module expression, but of an entire mature language and 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 would-be Maude extender to give 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.

2 Experiences Gained Implementing and Using Real-Time 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:

1. The above mentioned reluctance towards trying to understand the huge, complex, and barely documented implementation of Full Maude.
2. Doubts about the user-friendliness and lack of functionality of Full Maude.
3. 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 former approach, which did not treat object-oriented modules, made him appreciate the huge task of implementing an extension directly in Maude. Therefore, if you want that your extension retains the ability for the user 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 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, such as the ITP and the SCC, 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 (1), 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 [13, 9].

Regarding (2), 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. Full Maude makes it possible to make your implementation quite robust and to provide good diagnostic error messages without ending your Maude session, as exemplified by the timed commands in Real-Time Maude. The latest versions of Full Maude have benefited from some of these error handling features, so that Full Maude is now much more robust. This paper shows how the timed rewrite command has been made robust.

As for the lack of functionality, Full Maude cannot, e.g., exhibit a path reached in a search command, or continue on a rewriting command. The reason the missing features is the lack of

the corresponding metafunction, or for the complexity of the implementation. There is no reason however why they cannot be provided in the future.

Regarding (3), we show in this paper that an extension of Full Maude can be implemented in a modular way by just appending code to the implementation of Full Maude. 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 in such a way 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

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 meta-level 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 a *pair*, consisting of a *timed* module M and a *timed* command C, into a pair $(\widetilde{M}, \widetilde{C})$ of a Maude module $\widetilde{M}$ and Maude command $\widetilde{C}$ as described in [27].[1]

Write a small Maude interpreter from scratch. To gain invaluable understanding of the different aspects of writing an interpreter (such as the different phases of parsing, treatment of "bubbles," the loop mode, etc.), 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, Section 17.4] (combined with the user interface approach given in [4, Sections 17.1 to 17.3]).

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 enough in most cases. In the case of Real-Time Maude, the treatment of timed modules mirrors quite closely the handling of object-oriented modules in Full Maude, the implementatin of new timed commands follow the same scheme as their Full-Maude counter-parts, and so on.

3 Overview of Real-Time Maude

Real-Time Maude [27] 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, including: time-sensitive cryptographic protocols such as WMF and Kerberos [25, 16], the AER/NCA protocol suite for multicast in active networks [28], parts of the NORM multicast protocol developed by the IETF [20], the CASH scheduling algorithm [24], and the OGDC density control algorithm [29] and the LMST topology control algorithm [18] for

[1] While most Real-Time Maude commands are executed in this way, there are some very specific Real-Time Maude commands, such as '`find latest`', which cannot be executed so.

wireless sensor networks. Real-Time Maude has been particularly useful for specifying distributed real-time systems in an object-oriented style. Below we briefly outline how real-time systems are specified and analyzed in Real-Time Maude. The paper [27] gives the full details about Real-Time Maude and its semantics.

In Real-Time Maude, a *timed module* (syntax `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

`crl [`l`] : {`t`} => {`u`} in time` τ `if` *cond* `.`

where `{_}` is an operator that encloses the state and τ is a term of the sort `Time` defining the domain. Object-based specifications can be defined as *timed object-oriented modules* (syntax `tomod ... endtom`) that extend Full Maude's object-oriented modules.

Real-Time Maude extends Maude's analysis commands to the timed setting and provides some additional time-specific commands. Real-Time Maude's *timed rewrite* command has the syntax[2]

`(trew [[`n`]] [in` *module* `:] `t_0 `in time <=` τ `.)`

and extends Maude's rewrite command to only simulate a behavior up to duration τ. The *timed search* command allows us to search for only states that can be reached within a certain time interval, and has the syntax

`(tsearch [[`n`]] [in` *mod* `:] `t_0 `=>*` t `such that` *cond*
`in time-interval between >=` τ_1 `and <=` τ_2 `.)`

(or without upper or lower time bounds, or with arrows such as `=>!` instead of `=>*`). The tool also offers different forms of timed and untimed LTL model checking, as well as time-specific analysis commands, such as

`(find latest` t_0 `=>*` t `in time <=` τ `.)`

For dense time, to cover the whole time domain, tick rules typically have the form

`crl [`*tick*`] : {`t`} => {`u`} in time` T `if` T `<=` $mte(t)$ `[nonexec] .`

where T is a variable that does not occur in t. Such rules are not directly executable (`nonexec`); for execution purposes, Real-Time Maude provides a choice of a set of *time sampling strategies* which guides the application of such tick rules.

We can summarize the different ways in which Real-Time Maude extends Full Maude as follows:

- Provides a library of predefined modules (e.g., for time domains) that should be available for importation by user-defined modules.
- Provides new kinds of modules with new syntax.
- Provides new kinds of rules (tick rules) with the syntactic construct `in time`.
- Provides many additional analysis commands with their own syntax and semantics.

[2]Optional parts are enclosed by '[' and ']'.

- It must keep certain additional data, namely, the current *time sampling strategy*, in its *persistent state*, since the last time sampling strategy selected should be used, even if new modules are introduced or selected.

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 `TIMED-PRELUDE` module:

```
fmod TIMED-PRELUDE is including TIME .
  sorts System GlobalSystem ClockedSystem .
  subsort GlobalSystem < ClockedSystem .

  op {_} : System -> GlobalSystem [format (g o g so)] .
  op _in time_ : GlobalSystem Time -> ClockedSystem [format (o g g y o)] .

  eq (CLS:ClockedSystem in time R:Time) in time R':Time
    = CLS:ClockedSystem in time (R:Time plus R':Time) .
endfm
```

Likewise, a timed object-oriented module has a basic transformation into a Maude module that imports the module `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.

To support the construction of execution environments and formal tools, Maude provides some useful features for parsing, pretty printing, and input/output: The *syntax definition* can be accomplished by defining the data type `Grammar`; particularities at the *lexical* level can be accommodated by user-definable *bubble sorts* —having bubbles implies at least two parsing steps—; parsing and pretty printing is accomplished by the `metaParse` and `metaPrettyPrint` functions in the `META-LEVEL` module; input/output is accomplished by the predefined module `LOOP-MODE`, which provides a generic read-eval-print loop.

Given any input enclosed in parentheses, the read-eval-print loop gives us a list of quoted identifiers $\mathcal{I}$ obtained automatically 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 in the terminal after removing the quotes. Calling the function `metaParse` with the input $\mathcal{I}$ and the metarepresentation of the signature `Grammar` in which we want to parse it, if there is a parse, we get the corresponding *parse tree* as a term $\mathcal{T}_{\mathcal{I}}$ of sort `Term`. The reverse process is accomplished by the `metaPrettyPrint` function, which takes a term of sort `Term`, and returns its representation as a list of quoted identifiers.

The term $\mathcal{T}_{\mathcal{I}}$ can be seen as the representation of the input $\mathcal{I}$ as a term of sort `Term`. We could manipulate this term $\mathcal{T}_{\mathcal{I}}$ directly, applying some function to it to perform some kind of analysis or computation. Our experience, however, is that it is simpler and more appropriate to transform this term into another term in some data types `Module`, `Command`, or `Term`, respectively, for modules, commands, or terms.

Of course, we not only want to give some command or module expression with some particular set of arguments and get some result, or to just apply some transformation on some particular

input. We also want to be able to interact with the system, entering modules, theories, views, and commands of different types.

Modules can be parameterized by theories, and both of them can import other modules or theories, or combinations of them given by *module expressions*. We need to be able to store modules, theories, and views in a *database*, so that they can be referred later in order to evaluate module expressions, evaluate commands, and so on. The specification of Full Maude includes a data type `Database` for the database of modules, theories, and views. Thus, for example, the processing of a term obtained from a call to `metaParse` with an input corresponding to a module is accomplished by the function

```
op procModule : Term Database -> Database .
```

The read-eval-print loop provided by `LOOP-MODE` allows us to store the current state of the database.

If the input parse tree corresponds to a command, the processing is done by the `procCommand` function. Commands usually refer to previously entered modules, even without explicitly referring to them. For example, we may have reduction commands like

```
Maude> (reduce 3 + 5 .)
```

or

```
Maude> (reduce in LIST{Nat} : 3 + 5 .)
```

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. The signature of the `procCommand` function is then as follows:

```
op procCommand : Term ModuleExpression Database -> Tuple{Database, QidList} .
```

The first argument is the parse tree corresponding to the command to be executed, the second one is the name of the by-default module, and the third one is the current state of the database. Since there might be commands that change the database, for example, a command may remove a module from the database or create new modules, a new database is given as result. The function `procCommand` therefore returns a tuple consisting of the resulting database and the result of the command to be given to the user.

The persistent state of the read-eval-print loop provided by the `LOOP-MODE` module is given by a single object of class `DatabaseClass`, which maintains the database of the system. Objects of this class have: an attribute `db`, of sort `Database`, to keep the actual database in which all the modules being entered are stored (as a set of records); an attribute `default`, to keep the identifier of the current default module; and attributes `input` and `output`, to simplify the communication of the read-eval-print loop with 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 .
```

The rules dealing with the read-eval-print loop are given in the modules `DATABASE-HANDLING` and `FULL-MAUDE`.

4.1 The FULL-MAUDE-SIGN module

To parse some input using the built-in function `metaParse`, Full Maude needs the metarepresentation of the signature in which the input is going to be parsed. Such a grammar is provided by the `FULL-MAUDE-SIGN` module, and its submodules, in which we can find the declarations so that any valid input, namely modules, theories, views, and commands, can be parsed. In particular, we find in these modules, among others, sorts `@Module@`, `@View@`, `@Bubble@`, and `@Command@`, of modules, views, bubbles, and commands, respectively, and syntax declarations as

```
op red_. : @Bubble@ -> @Command@ .
op rew_. : @Bubble@ -> @Command@ .
...
op search_=>*_. : @Bubble@ @Bubble@ -> @Command@ .
op search_=>!_. : @Bubble@ @Bubble@ -> @Command@ .
...
op show module . : -> @Command@ .
op show module_. : @ModExp@ -> @Command@ .
...
op fmod_is_endfm : @Interface@ @FDeclList@ -> @Module@ .
op mod_is_endm : @Interface@ @SDeclList@ -> @Module@ .
op omod_is_endom : @Interface@ @ODeclList@ -> @Module@ .
```

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

`rewrite` [[⟨*Nat*⟩]] [`in` ⟨*ModId*⟩ `:`] ⟨*Term*⟩ `.`

The syntax for the `[_]` and `in_:` optional parts will be given later, when solving the bubbles. If we tried to give the different alternatives in `FULL-MAUDE-SIGN`, we would get ambiguous parses. The operators declared as constructors of sort `@Module@` in the signature of Full Maude, are declared with two arguments: the header of the unit—either its name, or its name plus its interface for parameterized modules—and the list of declarations of such a unit.

In the following module `META-FULL-MAUDE-SIGN` we declare a constant `GRAMMAR` of sort `FModule`, and give an equation identifying this constant with the metarepresentation of a module in which there is a declaration importing `FULL-MAUDE-SIGN`. 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. The bubble sorts `@Token@`, `@Bubble@`, `@SortToken@`, and `@NeTokenList@` are declared in a submodule of `FULL-MAUDE-SIGN`. These sorts are used in the declarations describing the syntax of the system.

```
fmod META-FULL-MAUDE-SIGN is
  including META-LEVEL .
  including UNIT .

  op BUBBLES : -> FModule .
  op GRAMMAR : -> FModule [memo] .

  eq BUBBLES
    = (fmod 'GRAMMAR is
```

```
        including 'QID-LIST .
        sorts none .
        none
        op 'token : 'Qid -> '@Token@
             [special(
               (id-hook('Bubble, '1 '1)
                op-hook('qidSymbol, '<Qids>, nil, 'Qid)))] .
        op 'viewToken : 'Qid -> '@ViewToken@
             [special(
               (id-hook('Bubble, '1 '1)
                op-hook('qidSymbol, '<Qids>, nil, 'Qid)))] .
        op 'sortToken : 'Qid -> '@SortToken@
             [special(
               (id-hook('Bubble, '1 '1)
                op-hook('qidSymbol, '<Qids>, nil, 'Qid)
                id-hook('Exclude, ''[ ''] '< 'to ''', '. ''( ''') ''{ ''} ':
                                  'ditto 'precedence 'prec 'gather
                                  'assoc 'associative 'comm 'commutative
                                  'ctor 'constructor 'id: 'strat 'strategy
                                  'poly 'memo 'memoization 'iter 'frozen
                                  'config 'object 'msg)))] .
        op 'neTokenList : 'QidList -> '@NeTokenList@
             [special(
               (id-hook('Bubble, '1 '-1 ''( ''))
                op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                op-hook('qidSymbol, '<Qids>, nil, 'Qid)
                id-hook('Exclude, '.)))] .
        op 'bubble : 'QidList -> '@Bubble@
             [special(
               (id-hook('Bubble, '1 '-1 ''( ''))
                op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                op-hook('qidSymbol, '<Qids>, nil, 'Qid)))] .
        none
        none
      endfm) .

  eq GRAMMAR = addImports((including 'FULL-MAUDE-SIGN .), BUBBLES) .

endfm
```

4.2 The FULL-MAUDE module

The top FULL-MAUDE module provides the rules to initialize the loop, and to specify the communication between the loop—the input/output of the system—and the database. Depending on the kind of input that the database receives, its state will be changed, or some output will be generated.

```
mod FULL-MAUDE is
  pr META-FULL-MAUDE-SIGN .
  pr DATABASE-HANDLING .
  inc LOOP-MODE .
  pr BANNER .
```

The state of the persistent system, which is supported by the built-in module `LOOP-MODE`, is represented as a single object of class `DatabaseClass`:

```
subsort Object < State .
```

The initial state is defined by a constant `init`, which will start a loop, with an object (with identifier `o`) of class `DatabaseClass` as state:

```
op o : -> Oid [ctor] .
op init : -> System .

var  Atts : AttributeSet .
var  X@DatabaseClass : DatabaseClass .
var  O : Oid .
var  DB : Database .
var  ME : Header .
var  QI : Qid .
vars QIL QIL' QIL'' : QidList .
var  TL : TermList .
var  N : Nat .
vars RP RP' : ResultPair .
```

The `init` rule initializes the persistent state of Full Maude. The `initialDatabase` operator defined in the `DATABASE-HANDLING` module represents the initial state of the database.

```
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)] .
```

The `banner` constant is a string with the welcome message, and is transformed into a list of quoted identifier by the `string2qidList` function.

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 following `in` rule, that enters an input such as a module or a command from the user into the database. The constant `GRAMMAR` names the module containing the signature defining the top level syntax of Full Maude (see Section 4.1). 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; otherwise, an error message is placed in the output channel of the loop. (Of course, the input may be syntactically valid, but not semantically valid, since further processing—for example, of bubbles—may reveal a semantic inconsistency, or simply a syntax error.)

```
rl [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
                          '\r '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:

```
  rl [out] :
    [QIL,
     < O : X@DatabaseClass |
         db : DB, input : TL, output : (QI QIL'), default : ME, Atts >,
     QIL'']
    => [QIL,
        < O : X@DatabaseClass |
            db : DB, input : TL, output : nil, default : ME, Atts >,
        (QI QIL' QIL'')] .
endm
```

4.3 The DATABASE-HANDLING module

This module does the following:

1. It defines Full Maude's persistent database as an object of a class `DatabaseClass` with the four attributes `input`, `output`, `db`, and `default` described above.

2. It checks the `input` attribute, and decides what "kind" of input the user last entered. 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 of all modules (stored in the `db` attribute of 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 `module` rule below processes the different types of modules entered to the system.

```
crl [module] :
  < O : X@DatabaseClass |
      db : DB,
      input : (F[T, T']),
      output : nil,
      default : ME,
      Atts >
  => < O : X@DatabaseClass |
         db : procModule(F[T, T'], DB),
         input : nilTermList,
         output : ('Introduced 'module header2Qid(parseHeader(T)) '\n),
         default : parseHeader(T),
         Atts >
  if (F == 'fmod_is_endfm) or-else
      ...
    ((F == 'mod_is_endm)   or-else
     (F == 'omod_is_endom))) .
```

The condition of this rule checks the top operator of the parse tree. The state of the `DatabaseClass` object is changed in several ways:

- 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 (defined in the `UNIT-PROCESSING` module, see Section 4.4).
- The input term is removed from the `input` attribute.
- 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.
- The newly entered module becomes the default module.

The `parseHeader` operator parses the argument corresponding to the header of the module, and the `header2Qid` gives a single-quoted-identifier representation of such a header.

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.

```
crl [error] :
  < O : X@DatabaseClass |
     db : db(MIS, MNS, VIS, VES, MNS', MNS'', MNS3, QIL),
     input : TL, output : nil, default : ME, Atts >
  => < O : X@DatabaseClass |
        db : db(MIS, MNS, VIS, VES, MNS', MNS'', MNS3, nil),
        input : TL, output : QIL, default : ME, Atts >
  if QIL =/= nil .
```

This rule takes a message (a `QidList`) from the eighth argument of the `db` operator, and put it in the output channel of the `DatabaseClass` object.

All objects in the rules handling the database are given using a variable of sort `DatabaseClass` as class, and they all include a variable `Atts` of sort `AttributeSet` that grabs any additional attributes of the object. This allows defining subclasses of class `DatabaseClass` to add additional attributes to the persistent state in a very straightforward manner, as explained in Section **??** for Real-Time Maude.

The `red/rew/frew` rule below handles `parse`, `reduce`, `rewrite`, and `frewrite` commands by invoking the `procCommand` function (defined in the `COMMAND-PROCESSING` module):

```
crl [red/rew/frew] :
  < O : X@DatabaseClass | db : DB, input : (F[T]), output : QIL, default : ME,  Atts >
  => < O : X@DatabaseClass |
        db : getDatabase(procCommand(F[T], ME, DB)),
        input : nilTermList,
        output : getQidList(procCommand(F[T], ME, DB)),
        default : ME,
        Atts >
  if (F == 'parse_.)   or-else
    ((F == 'red_.)     or-else ((F == 'reduce_.)  or-else
    ((F == 'rew_.)     or-else ((F == 'rewrite_.) or-else
    ((F == 'frew_.)    or-else  (F == 'frewrite_.)))))) .
```

Similar rules are provided for the different other inputs.

4.4 The `UNIT-PROCESSING` module

Processing a term resulting from parsing some input corresponding to a unit (a module or a theory) is done by the 4procModule function. This function takes as arguments a term of sort `Term`, which represents some module or theory, and a database, and returns the updated database:

```
op procModule : Term Database -> Database .
```

The `procModule` function parses one by one each of the declarations in the module. To do so, it uses the `parseDecl` auxiliary function. 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—notice that a module at the metalevel does not include variable declarations other than those declared on the fly, but the 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.

```
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—without identity elements of operators—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 Full Maude, there is an equation for `procCommand` for each available command. Adding new commands implies adding new equations for this operator. Here, we explain how the `rew` command is implemented in Full Maude, and then how the *timed rewrite* command `trew` is implemented in Real-Time Maude. The following module defines the function `procCommand`:

```
fmod COMMAND-PROCESSING is
  pr UNIT-PROCESSING .
  pr UNIT-META-PRETTY-PRINT .

  op {_,_} : Term Type ~> ResultPair [ctor] .
  op getTerm : ResultPair ~> Term .
  op getType : ResultPair ~> Type .
  ...

  vars T T' T'' : Term .
  var  TL : TermList .
  vars DB DB' DB'' : Database .
  vars M M' : Module .
  vars ME ME' : ModuleExpression .
  vars D D' : Bound .
  var  B : Bool .
  var  QIL : QidList .
  var  VDS : OpDeclSet .
  vars QI QI' F V O : Qid .
  var  TMVB : [Tuple<Term|Module|OpDeclSet|Bound>] .
  var  RP : [ResultPair] .
  ...

  sorts Tuple<Term|Module|OpDeclSet|Bound>  ... .
  op `_`,_`,_`,_` : Term Module OpDeclSet Bound -> Tuple<Term|Module|OpDeclSet|Bound> .

  op getTerm   : Tuple<Term|Module|OpDeclSet|Bound> ~> Term .
  op getModule : Tuple<Term|Module|OpDeclSet|Bound> ~> Module .
  op getVars   : Tuple<Term|Module|OpDeclSet|Bound> ~> OpDeclSet .
  op getBound  : Tuple<Term|Module|OpDeclSet|Bound> ~> Bound .

  op procCommand : Term ModuleExpression Database -> QidList .
```

In a first step for any command, `procCommand` produces a flat module corresponding to the given "current" module `ME`. Furthermore, it might be the case that the current module needs to be recompiled (remember that Full Maude uses lazy recompilation of modules when submodules are changed); if so, Full Maude recompiles them (the `else`-part below):[3]

```
  eq procCommand('rew_.['bubble[T]], ME, DB)
    = if compiledModule(ME, DB)
      then procRew(ME, getFlatModule(ME, DB), 'bubble[T], unbounded, getVars(ME, DB), DB)
```

[3]Strictly speaking, the recompilation could probably be defered until you are sure that the current module is the one in which the command will be executed.

```
    else 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)))
    fi .
```

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 `rew` t, `rew [`n`]` t, `rew in` M `:` t, and `rew [`n`] in` M `:` t; all these options are still on the table, and
2. before the above question is settled, we do not know in which module the term t is to be parsed.

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 `rew [`n`] ...`), and finally the set of variables in the module in which to execute the command.[4] From this tuple (`TMVB`), it is easy to extract the term, module, and bound, and do the actual rewriting by calling the Maude descent function `metaRewrite`:

```
op procRew : ModuleExpression Module Term Bound OpDeclSet Database -> QidList .
op solveBubblesRew : Term Module Bool Bound OpDeclSet Database
                     -> [Tuple<Term|Module|OpDeclSet|Bound>] .

ceq procRew(ME, M, T, D, VDS, DB)
  = if RP :: ResultPair
    then (...
          '\b 'result '\o '\s eMetaPrettyPrint(getType(RP))
          '\s '\b ': '\o '\n '\s '\s
          eMetaPrettyPrint(getModule(TMVB), getTerm(RP))  '\n)
    else getMsg(getTerm(TMVB))
    fi
  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 `metaRewrite` went well, the result `RP` is a term of sort `ResultPair`, and the resulting term and its least sort are returned.

5 The Implementation of Real-Time Maude

This section describes the implementation of Real-Time Maude from a very practical perspective in order to give useful guidelines for extending Full Maude. In particular, we explain how to implement

[4] 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.

central tasks in any extension of Full Maude: how to define the user-level syntax of the modules and commands of your language (Section 5.3); how to define "built-in" modules available to the user (Section 5.2); how to read and execute user input, etc. As for Full Maude, the tool must first determine what the input corresponds to. If the input represents a module (or theory, etc.), it must be parsed, processed, and stored in the database. You may want to define a new data type for meta-representing modules in your language. For example, timed modules can be meta-represented as terms of a new subsort `TModule` of `Module`. If the user input is an analysis command, it is treated in the same way as a Full Maude. We show how to execute the timed rewrite command, as well as the '`find latest`' command which has no corresponding Full Maude version. Finally, Section 5.5 explains how the persistent state of Full Maude is extended.

5.1 Overall Structure of the Real-Time Maude Implementation

Just like Full Maude, Real-Time Maude is a Maude specification/application, defined in the `real-time-maude.maude` file, which is available at the Real-Time Maude web page `http://www.ifi.uio.no/RealTimeMaude`. This file is an extension of the `full-maude.maude` file that implements Full Maude and that can be downloaded from the Maude web page `http://maude.cs.uiuc.edu`.

The `real-time-maude.maude` file does not *modify* the file `full-maude.maude`, except for moving the `loop init` to the end of the `real-time.maude` file. The latter file is loaded into the system after the `full-maude.maude` file. The specification of Real-Time Maude is built on the specification of Full Maude, without redefining or modifying any module in it. This way extending Full Maude is very advantageous for clarity and, most importantly, it allows you to very easily upgrade your system every time Full Maude is updated.

The following sections describe the Real-Time Maude extension. 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. Obviously, we only show a very small fraction of the 5,000+ lines of code that define the *extension* to Real-Time Maude. Large chunks of code are not shown and are replaced by '`...`'.

5.2 Predefined Modules

The real-time extension starts by giving the "predefined" modules available to the Real-Time Maude user, as well as some other modules used by the system:

```
fmod TIME is
  sorts Time NzTime .
  subsort NzTime < Time .
  op zero : -> Time .
  op _plus_ : Time Time -> Time [assoc comm prec 33 gather (E e)] .
  ...
endfm

fmod TIMED-PRELUDE is including TIME .
  sorts System GlobalSystem ClockedSystem .
  subsort GlobalSystem < ClockedSystem .

  op {_} : System -> GlobalSystem [format (g o g so)] .
```

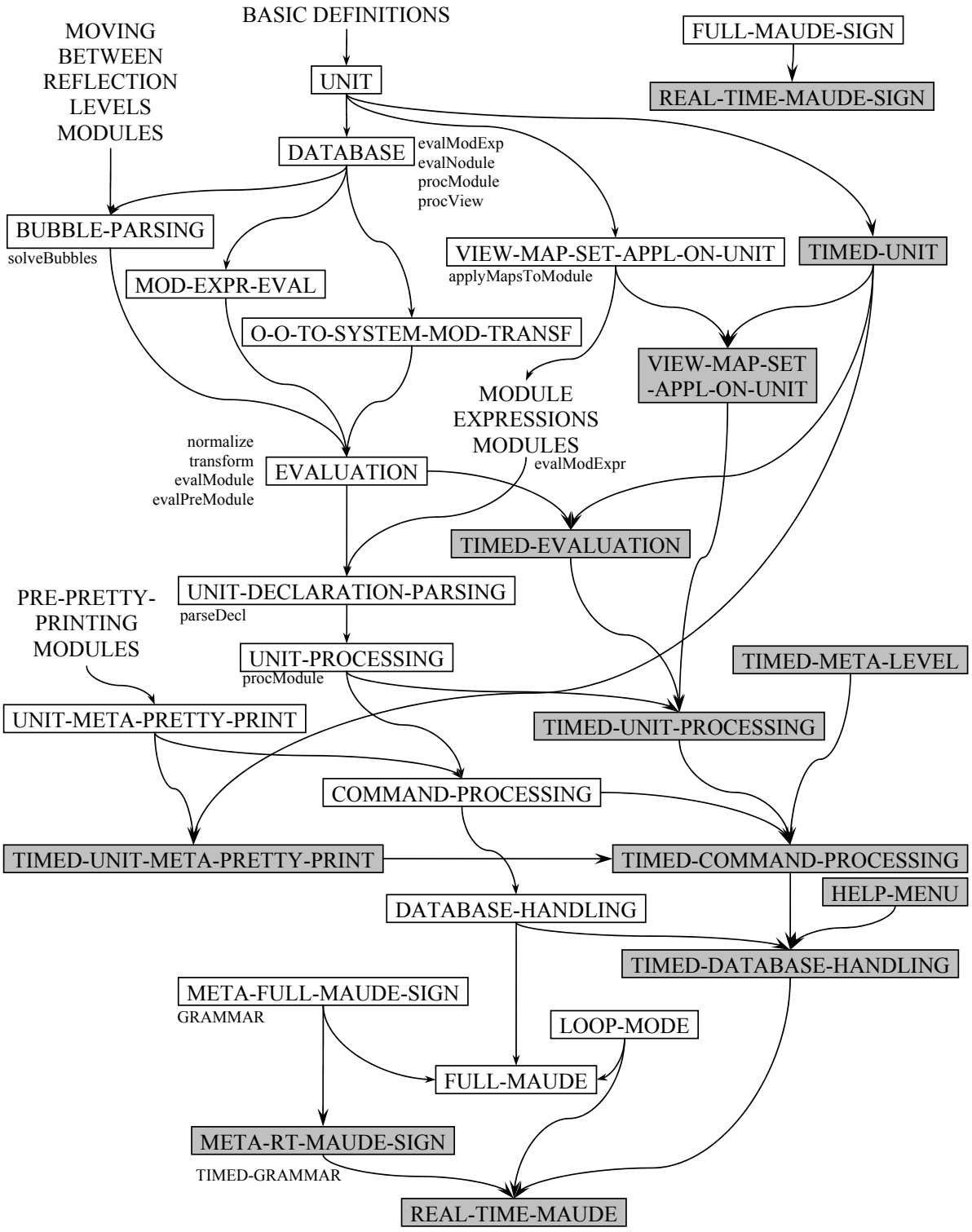

Figure 1: Significant modules and functions in the implementation of Full Maude and Real-Time Maude.

```
  op _in time_ : GlobalSystem Time -> ClockedSystem  [format (o g g y o)] .

  eq (CLS:ClockedSystem in time R:Time) in time R':Time
    = CLS:ClockedSystem in time (R:Time plus R':Time) .
endfm

...

fmod NAT-TIME-DOMAIN is
  including LTIME .
  protecting NAT .
  subsort Nat < Time .
  subsort NzNat < NzTime .
  ...
endfm

mod TIMED-OO-PRELUDE is
  including CONFIGURATION .
  including TIMED-PRELUDE .
  ...
endm

...
```

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.

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 Maude's module and command syntax. As explained in Section 4.1, the `FULL-MAUDE-SIGN` module (and its submodules) defines the syntax of Full Maude modules, theories, views, and commands.

The only additional module syntax in Real-Time Maude is that we have *timed modules* and *timed object-oriented modules*. Following the examples presented in Section 4.1 from the definitions for Full Maude (and without necessarily trying to understand the sorts `@Interface@`, `@FDeclList@`, etc.), the implementation of Real-Time Maude contains the following definition of the syntax of Real-Time Maude modules and theories:

```
fmod TIMED-MODULE-SYNTAX is
  including FULL-MAUDE-SIGN .
  *** Timed modules and theories:
  op tmod_is_endtm : @Interface@ @SDeclList@ -> @Module@ .
  op tth_is_endtth : @Interface@ @SDeclList@ -> @Module@ .

  *** Object-oriented timed modules and theories:
  op tomod_is_endtom : @Interface@ @ODeclList@ -> @Module@ .
  op toth_is_endtoth : @Interface@ @ODeclList@ -> @Module@ .
endfm
```

To define the syntax of the commands of Real-Time Maude, we again resort to the definition of the syntax of Full Maude's commands, and extend the module `FULL-MAUDE-SIGN` to define the syntax of the Real-Time Maude commands as follows:

```
fmod RTM-COMMAND-SYNTAX is
  including FULL-MAUDE-SIGN .
  *** Help commands:
  op help_. : @Token@ -> @Command@ .
  ...
  op show timed modules . : -> @Command@ .
  ...
  *** "Default" timed rewrite:
  op trew_in time <=_. : @Bubble@  @Bubble@ -> @Command@ .
  op trew_in time <_. : @Bubble@ @Bubble@ -> @Command@ .
  op trew_with no time limit . : @Bubble@ -> @Command@ .
  *** Default "fair" timed rewrite
  op tfrew_in time <=_. : @Bubble@  @Bubble@ -> @Command@ .
  ...
  *** Timed search commands:
  op tsearch_=>*_in time <=_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
  op tsearch_=>!_in time <=_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
  op tsearch_=>*_in time >=_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
  op tsearch_=>*_with no time limit . : @Bubble@ @Bubble@ -> @Command@ .
  op tsearch_=>*_in time-interval between__and__. :
        @Bubble@ @Bubble@ @Token@ @Bubble@ @Token@ @Bubble@ -> @Command@ .
  ...
  op find latest_=>*_in time <=_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
  ...
  *** Commands for setting the time sampling strategy:
  op set tick def_. : @Bubble@ -> @Command@ .
 op set tick max def_. : @Bubble@ -> @Command@ .
  ...
 op show tick mode . : -> @Command@ .
endfm
```

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

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

The following `META-RTM-SIGN` module introduces the `TIMED-GRAMMAR` constant, which is defined as module that extends the `GRAMMAR` defined in the `META-FULL-MAUDE-SIGN` module:

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

5.4 The REAL-TIME-MAUDE Module

Remember that the FULL-MAUDE module deals with: (i) initializing the loop used for user I/O; (ii) reading input from the user; and (iii) printing to the user some result that is ready to be printed. In an extension of Full Maude you probably need to extend the persistent state to store additional data. Real-Time Maude defines a subclass TimedDatabaseClass of DatabaseClass, which adds a new attribute timedData that is supposed to be initialized to some value initTimedData. The appropriate definitions are included in the TIMED-DATABASE-HANDLING module in Section 5.5.

Real-Time Maude's REAL-TIME-MAUDE top module is as the FULL-MAUDE module with the following changes: print a Real-Time Maude greeting; initialize the persistent state to an object of the class TimedDatabaseClass; and parse input in TIMED-GRAMMAR instead of in GRAMMAR (changes from FULL-MAUDE given in italics):

```
mod REAL-TIME-MAUDE is
  pr META-RTM-SIGN .
  pr TIMED-DATABASE-HANDLING .
  inc LOOP-MODE .
  pr BANNER .

  subsort Object < State .  --- state for LOOP mode

  op o : -> Oid [ctor] .
  op init : -> System .

  var  Atts : AttributeSet .
  var  X@Database : DatabaseClass .
  var  O : Oid .
  var  QI : Qid .
  vars QIL QIL' QIL'' : QidList .

  rl [init] :
     init
    =>
     [nil,
      < o : TimedDatabaseClass | db : initialDatabase,  input : nilTermList,
                                 output : nil, default : 'CONVERSION,
                                 timedData : initTimedData >,
      ('\n '\t '\s '\s '\s '\s '\s string2qidList(banner) '\n
       '\n '\t '\s '\s '\! '\m 'Real-Time 'Maude '2.3 '\o
       '\c 'extension 'May '2 '`, '\s '2007 '\o '\n)] .


  rl [in] :
     [QI QIL,
      < O : X@Database | input : nilTermList, output : nil, Atts >,
      QIL']
    =>
     if metaParse(TIMED-GRAMMAR, QI QIL, '@Input@) :: ResultPair then
```

```
        [nil,
         < O : X@Database | input : getTerm(metaParse(TIMED-GRAMMAR, QI QIL, '@Input@)),
                            output : nil, Atts >,
         QIL']
     else
        [nil,
         < O : X@Database | input : nilTermList,
                            output :
                      ('\r 'Warning:
                       printSyntaxError(metaParse(TIMED-GRAMMAR, QI QIL, '@Input@), QI QIL)
                       '\n '\r 'Error: '\o 'No 'parse 'for 'input. '\n),
                            Atts >,
         QIL']
     fi .

  rl [out] :
     [QIL,
     < O : X@Database | output : (QI QIL'), Atts >,
     QIL'']
    =>
     [QIL,
      < O : X@Database | output : nil, Atts >,
      (QI QIL' QIL'')] .
endm
```

5.5 Database Handling

Real-Time Maude's `TIMED-DATABASE-HANDLING` module extends Full Maude's `DATABASE-HANDLING` module—that treats the user input parsed in the top-level grammar—for the following purposes:

1. The real-time part needs to store additional data, such as the last *time sampling strategy* selected by the user, in the database.
2. Parse and store *timed* modules given by the user.
3. Parse and execute *timed* analysis commands and other commands not available in Full Maude (such as `help` and `show tick mode`).

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.

```
  mod TIMED-DATABASE-HANDLING is
    including DATABASE-HANDLING .
    protecting TIMED-UNIT-PROCESSING .
    protecting TIMED-COMMAND-PROCESSING .
    protecting TIMED-DATA .

    sort TimedDatabaseClass .
```

```
subsort TimedDatabaseClass < DatabaseClass .

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

A term of sort `TimedData` is a pair `<` *tmodNames* `,` *tss* `>`, where *tmodNames* denotes the names of all timed modules, and *tss* denotes the current time sampling strategy.

```
sort TimedData .
op <_,_> : QidList TickMode -> TimedData [ctor] .

sort TickMode .
ops det max : -> TickMode [ctor] .
ops def maxDef : Term -> TickMode [ctor] .

op initTimedData : -> TimedData .
eq initTimedData = < nil, det > .
```

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 therefore looks very much like the treatment of module declarations in the Full Maude implementation, with the only difference that the module name is added to the `timedData` attribute:

```
var ATTS : AttributeSet .
var TIMEDDATABASE : TimedDatabaseClass .
var DB : Database .
vars F Q : Qid .
vars T T' T'' T''' : Term .
var TL : TermList .
var O : Oid .
var MN : ModuleName .
var TIMEDDATA : TimedData .
var ME : ModuleExpression .
var QIL : QidList .

crl [readTimedModule] :
    < O : TIMEDDATABASE | db : DB, input : (F[T, T']), output : nil,
                          default : ME, timedData : TIMEDDATA, ATTS >
    =>
    < O : TIMEDDATABASE | db : procModule(F[T, T']),  DB),
                          input : nilTermList,
                          output : ('\n '\c 'Introduced 'timed 'module: '\o
                                    header2Qid(parseHeader(T)) '\n),
                          default : parseHeader(T),
                          timedData : addModName(TIMEDDATA, pureModName(T)),
                          ATTS >
    if (F == 'tmod_is_endtm) or-else (F == 'tomod_is_endtom) .
```

Executing timed analysis commands. In the `DATABASE-HANDLING` module, Full Maude treats an analysis command by just calling a function `procCommand`, with the command and some additional data, such as the "current" module and the module database, as arguments. We follow this

approach, and execute timed analysis commands by just calling the function `procTimedCommand`, which gets the current time sampling strategy as an additional argument:[5]

```
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
     F == 'utsearch_=>*_.            or    F == 'utsearch_=>!_. or
     F == 'find'earliest_=>*_.       or    F == 'find'latest_=>*_in'time'<=_.  or
     F == 'tsearch_=>*_in'time-interval'between__and__. or
     ... .
```

We must previously have defined the crucial function `procTimedCommand`, which is done in the module `TIMED-COMMAND-PROCESSING` explained in Section 5.7.

Parsing and executing other kinds of commands. Finally, we have other commands which can be executed "directly." One such command is setting the time sampling strategy to be the maximal strategy, with a default value (`T`, as yet unparsed). The result of this command is to update the current time sampling strategy in the `timedData` attribute:

```
rl [maxDefMode] :
   < O : TIMEDDATABASE | input : ('set'tick'max'def_.[T]),
                         output : nil, timedData : TIMEDDATA, ATTS >
  =>
   < O : TIMEDDATABASE | input : nilTermList,
                         output : ('\n '\c 'Tick 'mode 'set 'to  ... '\o '\n),
                         timedData : setTickMode(TIMEDDATA, maxDef(T)), ATTS > .
```

Finally, we define the command that prints the names of all *timed* modules:

```
rl [showTimedModules] :
   < O : TIMEDDATABASE | input : ('show'timed'modules'..@Command@),
                         output : nil, timedData : TIMEDDATA, ATTS >
  =>
   < O : TIMEDDATABASE | input : nilTermList,
                         output : ('\n '\c 'Timed 'modules 'are: '\y
                                   getModNames(TIMEDDATA) '\o '\n),
                         timedData : TIMEDDATA, ATTS > .
endm
```

[5] We only show a fraction of the commands handled by the `timedExecution` rule.

5.6 Processing Timed Modules

As described above, it the task of the `procModule` function to process the input corresponding to timed modules and introduce 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. In the case of object-oriented modules and theories this `transform` function is in charge of transforming an object-oriented module into a system module. This same scheme can be used for other types of modules, in particular in the case of the new modules in Real-Time Maude.

The first thing we need is to declare operators to metarepresent the new types of modules. In the case of Real-Time Maude we have two new types of modules and two new types of theories, namely, timed system modules (`tmod...endtm`) and theories (`tth...endtth`) and timed object-oriented modules (`tomod...endtom`) and theories (`toth...endtoth`). The module `TIMED-UNIT`, which extends `UNIT`, adds the following declarations:

- New sorts `TModule`, `TTheory`, `TOModule`, and `TOTheory`, with the subsort declarations

```
subsorts SModule < TModule OModule < TOModule < Module .
subsorts STheory < TTheory OTheory < TOTheory < Module .
```

- New operators for representing the different types of modules. For instance, the operator

```
op tomod_is_sorts_._________endtom : Header ImportList
      SortSet SubsortDeclSet ClassDeclSet SubclassDeclSet OpDeclSet
      MsgDeclSet MembAxSet EquationSet RuleSet -> TOModule
      [ctor  gather (& & & & & & & & & &)
       format (r! o r! n++io ni d d ni ni ni ni ni ni ni ni n--ir! o)] .
```

  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 for the different components of a module or theory for the new types of modules and theories are added. Specifically, equations for the `getName`, `getImports`, `getPars`, `getSorts`, `getSubsorts`, `getOps`, `getMbs`, `getEqs`, `getRls`, `getClasses`, `getSubclasses`, `getMsgs`, `setName`, `setImports`, `setPars`, `setSorts`, `setSubsorts`, `setOps`, `setMbs`, `setEqs`, `setRls`, `setClasses`, `setSubclasses`, and `setMsgs` functions. For example, for the `setOps` operator, the following equations are included:

```
eq setOps(tmod H is IL sorts SS . SSDS OPDS MAS EqS RlS endtm, OPDS')
  = tmod H is IL sorts SS . SSDS OPDS' MAS EqS RlS endtm .
eq setOps(tth MN is IL sorts SS . SSDS OPDS MAS EqS RlS endtth, OPDS')
  = tth MN is IL sorts SS . SSDS OPDS' MAS EqS RlS endtth .
eq setOps(tomod H is IL sorts SS . SSDS CDS SCDS OPDS MDS MAS EqS RlS endtom, OPDS')
  = tomod H is IL sorts SS . SSDS CDS SCDS OPDS' MDS MAS EqS RlS endtom .
eq setOps(toth MN is IL sorts SS . SSDS CDS SCDS OPDS MDS MAS EqS RlS endtoth, OPDS')
  = toth MN is IL sorts SS . SSDS CDS SCDS OPDS' MDS MAS EqS RlS endtoth .
```

- Operators defining empty modules and theories of the new types. For timed object-oriented modules, a new constant `emptyTOModule` is declared with the following definition:

```
eq emptyTOModule
  = tomod nullHeader is nil sorts none . none none none none none none
      none none endtom .
```

- Finally, equations for the `empty` operator, which returns an empty module of the same type of the one given as argument are added. For example, for timed object-oriented modules the following equation is given:

```
eq empty(tomod H is IL sorts SS . SSDS CDS SCDS OPDS MDS MAS EqS RlS endtom)
  = (tomod H is nil sorts none . none none none none none none none none endtom) .
```

The `TIMED-EVALUATION` module imports the `EVALUATION` and `TIMED-UNIT` modules and add equations for the `transform` operator. In the case of the timed modules, 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 :: TOTheory .

op tmod2mod : Module Database -> Module .
eq tmod2mod(tmod H is IL sorts SS . SSDS OPDS MAS EqS RlS endtm, DB)
  = (mod H is IL sorts SS . SSDS OPDS MAS EqS RlS endm) .
eq tmod2mod(tth MN is IL sorts SS . SSDS OPDS MAS EqS RlS endtth, DB)
  = (th MN is IL sorts SS . SSDS OPDS MAS EqS RlS endth) .
eq tmod2mod(tomod H is IL sorts SS . SSDS CDS SCDS OPDS MDS MAS EqS RlS endtom, DB)
  = omod2mod(omod H is IL sorts SS . SSDS CDS SCDS OPDS MDS MAS EqS RlS endom, DB) .
eq tmod2mod(toth MN is IL sorts SS . SSDS CDS SCDS OPDS MDS MAS EqS RlS endtoth, DB)
  = omod2mod(oth MN is IL sorts SS . SSDS CDS SCDS OPDS MDS MAS EqS RlS endoth, DB) .
```

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. Auxiliary functions, already available in Full Maude, apply the renaming maps to the different components of a module:

```
eq applyMapsToModuleAux(VMAPS, VMAPS',
     tomod H is IL sorts SS . SSDS CDS SCDS OPDS MDS MAS EqS RlS endtom, M)
  = tomod H is
        IL
        sorts applyMapsToSortSet(VMAPS, SS) .
        applyMapsToSubsorts(VMAPS, SSDS)
        applyMapsToClassDeclSet(VMAPS, VMAPS', CDS)
        applyMapsToSubclassDeclSet(VMAPS', SCDS)
```

```
        applyMapsToOps(VMAPS, VMAPS', OPDS, M)
        applyMapsToMsgDeclSet(VMAPS, VMAPS', MDS, M)
        applyMapsToMbs(VMAPS, VMAPS', MAS, M)
        applyMapsToEqs(VMAPS, VMAPS', EqS, M)
        applyMapsToRls(VMAPS, VMAPS', RlS, M)
    endtom .
```

- The TIMED-UNIT-META-PRETTY-PRINT module includes the UNIT-META-PRETTY-PRINT and TIMED-UNIT modules and defines how the new modules and theories are meta-pretty-printed. For example, the equation for timed object-oriented modules is the following:

```
eq eMetaPrettyPrint(M, tomod ME is IL sorts SS . SSDS CDS SCDS OPDS MDS MAS
                            EqS RlS endtom)
  = ('\n '\b
     'tomod '\o eMetaPrettyPrint(ME) '\b 'is '\o
        eMetaPrettyPrint(IL)
        (if SS == none
         then nil
         else ('\n '\s '\s '\b 'sorts '\o eMetaPrettyPrint(SS) '\b '. '\o)
         fi)
        eMetaPrettyPrint(SSDS)
        eMetaPrettyPrint(CDS)
        eMetaPrettyPrint(SCDS)
        eMetaPrettyPrint(M, OPDS)
        eMetaPrettyPrint(MDS)
        eMetaPrettyPrint(M, MAS)
        eMetaPrettyPrint(M, EqS)
        eMetaPrettyPrint(M, RlS) '\n '\b
     'endtom '\o '\n) .
```

- 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 of the declarations in the module are processed and added to it. Some of these declarations are only partially processed, since they may require subsequent parsing of the bubbles in them. This is the case, e.g., of identity elements in operator declarations, terms in membership axioms, equations, and rules, etc. In order to get the signature in which the bubbles are to be parsed, 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 purposes, 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, which imports the UNIT-PROCESSING, TIMED-DATA, and TIMED-EVALUATION modules, add equations to the procModule2 operators to appropriately consider the new types of modules:

```
eq procModule2(T, 'tmod_is_endtm[T', T''], DB)
  = procModule3(T, T', T'',
```

```
        addImports((including 'TIMED-PRELUDE .), emptyTModule),
        DB) .
    eq procModule2(T, 'tomod_is_endtom[T', T''], DB)
      = procModule3(T, T', T'',
        addImports((including 'TIMED-OO-PRELUDE .
                    including 'CONFIGURATION+ .), emptyTOModule),
        DB) .
    eq procModule2(T, 'tth_is_endtth[T', T''], DB)
      = procModule3(T, T', T'',
        addImports((including 'TIMED-PRELUDE .), emptyTTheory),
        DB) .
    eq procModule2(T, 'toth_is_endtoth[T', T''], DB)
      = procModule3(T, T', T'',
        addImports((including 'TIMED-OO-PRELUDE .
                    including 'CONFIGURATION .), emptyTOTheory),
        DB) .
```

5.6.1 An Alternative Quick and Easy Way of Processing Timed Modules

Previous versions of Real-Time Maude internally represented timed and timed object-oriented modules as terms of the Full Maude sorts `Module` and `OModule` by transforming input declared as a `tmod` or `tomod` into a `mod` or `omod` *before parsing* the module body, by just adding an importation of `TIMED-PRELUDE` (resp., `TIMED-OO-PRELUDE`) into the unparsed ("bubble") module body. Then, this transformed input was further parsed and stored by Full Maude. The entire treatment of timed modules was defined by the following rule in the `TIMED-DATABASE-HANDLING` module:

```
crl [readTimedModule] :
    < O : TIMEDDATABASE | db : DB, input : (F[T, T']), output : nil,
                          default : ME, timedData : TIMEDDATA, ATTS >
    =>
    < O : TIMEDDATABASE | db : procModule(timedPreModuleToPreModule(F[T, T']),  DB),
                          input : nilTermList,
                          output : ('\n '\c 'Introduced 'timed 'module: '\o
                                    header2Qid(parseHeader(T)) '\n),
                          default : parseHeader(T),
                          timedData : addModName(TIMEDDATA, pureModName(T)),
                          ATTS >
    if (F == 'tmod_is_endtm) or-else (F == 'tomod_is_endtom) .
```

where the `timedPreModuleToPreModule` function was defined as follows:

```
op timedPreModuleToPreModule : Term -> Term .

eq timedPreModuleToPreModule('tmod_is_endtm[T, T']) =
     'mod_is_endm[T, '__['including_.['token[''TIMED-PRELUDE.Qid]], T']] .

eq timedPreModuleToPreModule('tomod_is_endtom[T, T']) =
     'omod_is_endom[T, '__['including_.['token[''TIMED-OO-PRELUDE.Qid]], T']] .
```

5.7 Processing the Timed Analysis Commands

The `COMMAND-PROCESSING` module in the `full-maude.maude` file provides the `procCommand` function, which comes with three arguments: the command, with bubbles for its parameter(s), the name of the "current" module, and the database. We extend this module in the implementation of Real-Time Maude in a module `TIMED-COMMAND-PROCESSING`.

We explain how the `procTimedCommand` function, that executes the timed analysis commands, is defined, by showing how it handles the `trew_in time <=_.` command.[6] 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 term in 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 implemented as an ordinary Maude function on Maude meta-modules and meta-terms.

```
fmod TIMED-COMMAND-PROCESSING is
  including COMMAND-PROCESSING .
  ...
  protecting TIMED-DATA .

  op procTimedCommand : Term ModuleExpression Database TickMode -> QidList .

  vars T T' T'' T''' LIMIT TERM : Term .
  var  ME : ModuleExpression .
  var  DB : Database .
  var  B : Bool .
  var  ODS : OpDeclSet .
  var  RP : [ResultPair] .
  vars M M' MOD : Module .
  vars D BOUND : Bound .
  var  VDS : OpDeclSet .
  vars TiM SOLVEDTICKMODE : TickMode .

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(modExp(evalModExp(ME, DB)),
                             getFlatModule(modExp(evalModExp(ME, DB)),
                                           database(evalModExp(ME,DB))),
                             unbounded,
                             getVars(modExp(evalModExp(ME, DB)),
                                     database(evalModExp(ME,DB))),
```

[6] The exposition, but not the essence, is somewhat simplified, since, due to the large number of commands in Real-Time Maude, we combine the treatment of some commands in the implementation of Full Maude.

```
                              database(evalModExp(ME,DB)),
                              T, T', TiM)
    fi .

op processTimedRewrite : ModuleExpression Module Bound OpDeclSet
                         Database Term Term TickMode -> QidList .
```

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

```
ceq processTimedRewrite(ME, M, D, VDS, DB, T, T',  TiM)
  = if RP :: ResultPair
    then (...  '\c '\n 'Result '\o
          eMetaPrettyPrint(getType(RP)) '\c ': '\o '\n '\s '\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)
     /\  SOLVEDTICKMODE := solveTickMode(TiM, MOD, B, getVars(getName(MOD), DB), DB) .
     /\  RP := timedMetaRewrite(MOD, TERM, BOUND, le, LIMIT, SOLVEDTICKMODE) .
```

The `solveTickMode` function takes a `TickMode`, where the argument is a bubble, and returns the same tick mode, but now with the bubble resolved by using the Full Maude function `solveBubbles`, which parses a single term in a module:

```
op solveTickMode : TickMode Module Bool OpDeclSet Database ~> TickMode .
eq solveTickMode(maxDef(T), M, B, VDS, DB) = maxDef(solveBubbles(T, M, B, VDS, DB)) .
...
```

Several error situations can happen. If the `solveBubblesRew` invocation encounters some problems in its parsing of the module, term to be rewritten, or bound, an error message is returned:

```
ceq processTimedRewrite(ME, M, D, VDS, DB, T, T', TiM)
  = ('\n '\r 'Error '\c 'in 'timed 'rewrite:
     'Command/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:

```
ceq processTimedRewrite(ME, M, D, VDS, DB, T, T', TiM)
  = ('\n '\r 'Error: '\c 'Time 'limit 'term 'does 'not
     'parse 'in 'module '\y eMetaPrettyPrint(getName(MOD)) '\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`.

As we see, apart from using `timedMetaRewrite` instead of `metaRewrite`, and from having to parse the time limit and time sampling strategy term, the treatment of `trew` is exactly as for `rew`. Notice also how we have used the infrastructure provided by the implementation of Full Maude to make the timed command processing robust.

The 'Find Latest' Command. To also give an example of a command which is not just a timed extension of Maude, we show how the `find latest` command—that finds the time it takes in the worst case to reach a desired state—is implemented. The syntax of this command is:

```
op find latest_=>*_in time <=_. : @Bubble@ @Bubble@ @Bubble@ -> @Command@ .
```

The first argument represents a term (possibly together with the module in which the command is to be executed). The second argument represents a term pattern, or a term pattern followed by `such that` and a condition, that represents the states we are searching for.

The steps to perform are:

1. Find out in which module the command is to be executed, and get the flattened version of that module.
2. Parse the initial term, the pattern (possibly with its condition), the time limit term, and the time sampling strategy time term in the above module.
3. Call the appropriate meta-level function to execute the command.

The first step is to compile the "current" module, not necessarily the module in which the command is to be executed:

```
eq procTimedCommand('find`latest_=>*_in`time`<=_.[T, T', T''], ME, DB, TiM)
  = if compiledModule(ME, DB)
    then processFindLatest(ME, getFlatModule(ME, DB), unbounded,
                           getVars(ME, DB), DB, Q, T, T', T'',TiM)
    else processFindLatest(modExp(evalModExp(ME, DB)),
                           getFlatModule(modExp(evalModExp(ME, DB)),
                                         database(evalModExp(ME, DB))),
                           unbounded,
                           getVars(modExp(evalModExp(ME, DB)),
                                   database(evalModExp(ME, DB))),
                           database(evalModExp(ME, DB)),
                           Q, T, T', T'', TiM)
    fi .

op processFindLatest : ModuleExpression Module Bound OpDeclSet
             Database Term Term Term TickMode -> QidList .
```

The following equations use the function `solveBubblesRew` to parse the first argument, which, as mentioned, consists of the term representing the initial state, but that could additionally specify the module in which to do the analysis. First, an error message is returned if there is some error with parsing the first argument:

```
ceq processFindLatest(ME, M, D, VDS, DB, T, 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> .
```

We must next check whether the bubbles representing the search pattern, the time limit, and the tick mode value parse in the module in which to perform the command. For each of these checks, we have a rule that prints an error message if the given bubble does not parse. This is done as for `processTimedRewrite` and is not shown. Finally, if all these bubbles parse, the following equation executes the command by calling the metalevel function `findLatest`:

```
ceq processFindLatest(ME, M, D, VDS, DB, T, T', T'', TiM) =
      ('\n '\c 'Find 'latest ...  ': '\n '\o
       (if TS :: Term then
          ('\c '\n 'Result:  '\o '\t eMetaPrettyPrint(MOD, TS) '\o '\n)
        else   --- TS == noterm
          ('\c '\n 'Result: 'there 'is 'a 'path 'in 'which 'the 'pattern 'is
           'not 'reachable 'in 'time '<= eMetaPrettyPrint(MOD, LIMIT)  '\o '\n)
         fi)
       )
    if B := included('META-MODULE, getImports(getTopModule(ME, DB)), DB)
       /\ {TERM, MOD, ODS, BOUND} := solveBubblesRew(T, M, B, D, VDS, DB)
       /\ SEARCHPATTERN := searchPattern(TERM, T', MOD, B,
                                         getVars(getName(MOD), DB), DB)
       /\ SEARCH-TERM := termPart(SEARCHPATTERN)
       /\ SEARCH-COND := condPart(SEARCHPATTERN)
       /\ LIMIT := solveBubbles(T'', MOD, B, getVars(getName(MOD), DB), DB)
       /\ SOLVEDTICKMODE := solveTickMode(TiM, MOD, B,
                                          getVars(getName(MOD), DB), DB)
       /\ TS := findLatest(MOD, TERM, SEARCH-TERM, SEARCH-COND, LIMIT, SOLVEDTICKMODE) .
```

The function `searchPattern` solves the bubbles for the search pattern.

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.

The description of such a significant extension complements other extensions proposed for adding new commands and modules expressions. 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 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

[1] G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In *Proc. 3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05)*, 2005.

[2] F. Chalub and C. Braga. Maude MSOS tool. *Electronic Notes in Theoretical Computer Science*, 176(4):133–146, 2006. `http://dx.doi.org/10.1007/978-3-540-71999-1_21`.

[3] M. Clavel. The ITP tool home page. `http://maude.sip.ucm.es/itp/`.

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

[5] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In K. Futatsugi, A. T. Nakagawa, and T. Tamai, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*, pages 1–31. Elsevier, 2000. `http://maude.csl.sri.com/papers`.

[6] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods (Vol. II)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1704. Springer, 1999.

[7] M. Clavel, F. Durán, J. Hendrix, S. Lucas, J. Meseguer, and P. Ölveczky. The Maude formal tool environment. In T. Mossakowski, editor, *Algebra and Coalgebra in Computer Science (CALCO'07)*, volume 4624 of *Lecture Notes in Computer Science*, pages 173–178. Springer, 2007. Proc. of CALCO'07.

[8] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006.

[9] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, Spain, June 1999. `http://maude.csl.sri.com/papers`.

[10] F. Durán. Coherence checker and completion tools for Maude specifications. Technical Report ITI-2000-7, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, October 2000. Available at `http://maude.cs.uiuc.edu`.

[11] F. Durán, S. Escobar, and S. Lucas. On-demand evaluation for Maude. In S. Abdennadher and C. Ringeissen, editors, *Proceedings of Fifth International Workshop on Rule-Based Programming (RULE'04)*, volume 124 of *Electronic Notes in Theoretical Computer Science*, pages 25–39. Elsevier, 2004.

[12] F. Durán, S. Escobar, and S. Lucas. New evaluation commands for Maude within Full Maude. In N. Martí-Oliet, editor, *5th International Workshop on Rewriting Logic and its Applications (WRLA'04)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 263–284. Elsevier, 2005.

[13] F. Durán and J. Meseguer. The Maude specification of Full Maude. Manuscript, SRI International. Available at `http://maude.cs.uiuc.edu`, February 1999.

[14] F. Durán and J. Meseguer. A Church-Rosser checker tool for Maude equational specifications. Technical Report ITI-2000-5, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, October 2000. Available at `http://maude.cs.uiuc.edu`.

[15] F. Durán and J. Meseguer. Maude's module algebra. *Science of Computer Programming*, 66(2):125–153, April 2007.

[16] M. Grimeland. Modeling and analysis of time-dependent security protocols in Real-Time Maude. Master's thesis, Department of Informatics, University of Oslo, 2006.

[17] J. Hendrix, M. Clavel, and J. Meseguer. A sufficient completeness reasoning tool for partial specifications. In *Proc. Rewriting Techniques and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pages 165–174. Springer, 2005.

[18] M. Katelman, J. Meseguer, and J. Hou. Formal modeling, analysis, and debugging of a wireless sensor network protocol with Real-Time Maude and statistical model checking. Technical report, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 2007. In preparation.

[19] N. Kumar, K. Sen, J. Meseguer, and G. Agha. A rewriting based model of probabilistic distributed object systems. In *Proc. FMOODS 2003*, volume 2884 of *Lecture Notes in Computer Science*. Springer, 2003.

[20] E. Lien. Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude. Master's thesis, Department of Linguistics, University of Oslo, 2004.

[21] D. Lucanu and G. Roşu. CiRC: A circular coinductive prover. In T. Mossakowski, U. Montanari, and M. Haveraaen, editors, *Algebra and Coalgebra in Computer Science, 2nd International Conference, CALCO 2007, Bergen, Norway, August 20-24, 2007. Proceedings*, volume 4624 of *Lecture Notes in Computer Science*, pages 372–378. Springer, 2007.

[22] D. Lucanu and G. Roşu. CiRC tutorial and user manual. Available at `http://fsl.cs.uiuc.edu/index.php/Circ`, March 2007.

[23] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towars a strategy language for Maude. In N. Martí-Oliet, editor, *4th International Workshop on Rewriting Logic and its Applications (WRLA'04)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, January 2005.

[24] P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering (FASE'06)*, volume 3922 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2006.

[25] P. C. Ölveczky and M. Grimeland. Formal analysis of time-dependent cryptographic protocols in Real-Time Maude. In *21st International Parallel and Distributed Processing Symposium (IPDPS 2007)*. IEEE Computer Society Press, 2007.

[26] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[27] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[28] P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29(3):253–293, 2006.

[29] P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. In M. M. Bonsangue and E. B. Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science*, pages 122–140. Springer, 2007.

[30] A. Verdejo. Lotos symbolic semantics in maude. Technical Report 122-02, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain, January 2002.