Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 238 (2009) 121–138

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Patterns for Maude Metalanguage Applications

Eugen-Ioan Goriac[a,1,2] Georgiana Caltais[a,1,2] Dorel Lucanu[a,1,2]
Oana Andrei[b,3] Gheorghe Grigoraş[a,1,2]

[a] *Faculty of Computer Science*
Alexandru Ioan Cuza University
Iaşi, Romania

[b] *INRIA Nancy Grand-Est & LORIA*
Nancy, France

Abstract

One of the most effective ways of improving the quality of software engineering, system design and development, and communication between the people concerned with these problems, is provided by software patterns. In this paper we present a set of basic patterns for Maude metalanguage applications. We show the viability of the defined patterns by comparing them to the developing approaches for several well-known Maude tools.

Keywords: Maude language, Maude metalanguage applications, software patterns.

1 Introduction

Maude [6] is a high-level language and high-performance system based on equational and rewriting logic computation. It is a flexible and general semantic framework suitable for giving semantics to a wide range of languages and models of concurrency. It is also a good logical framework, i.e., a metalogic in which many other logics can be naturally represented and implemented. The reflective property of rewriting logic permits the development of many advanced metaprogramming and metalanguage applications.

A *metalanguage application* is a particular type of application in which Maude is used to define modules for specifying an object language syntax, parser, way of execution and manner of printing execution results. These kinds of applications

[1] This work is partially supported by the PNII grant IDEI 393.

[2] Email: [egoriac,gcaltais,dlucanu,grigoras]@info.uaic.ro

[3] Email: Oana.Andrei@loria.fr

doi:10.1016/j.entcs.2009.05.016

are implemented using the Maude metalevel capabilities. For a detailed description of working at metalevel and metalanguage applications, see Chapters 14 and 17, respectively, from [6].

There is already a significant number of metalanguage applications in Maude. Some of them are written by people with Maude programming experience; their applications have a high performance and a good quality design which make them reusable for other software engineers. An application written by someone less familiar with Maude has a low degree of re-usability. The success of the learning process by a new Maude user strongly depends on the kinds of examples he/she studies. We think that Maude has reached a certain maturity level when the best practices should be accessible to a large class of users. This goal can be reached by the means of a good software engineering problem-solving discipline. Such a discipline is given by *patterns*. The general goal of the patterns is to support design and development.

The contribution of this paper is the definition of a set of basic patterns which may be used in a wide range of Maude metalanguage applications. The design of these patterns is based on the experience acquired by the authors during the development of some applications [10,2] or by studying other applications like those presented in [11,8,7]. The referred applications are: the *Inductive Theorem Prover* (ITP), the *Maude Termination Tool* (MTT), the *Church-Rosser Checker* (CRC) the *Real Time Maude Tool* (RTM) and the *Strategy language for Maude* (STR).

The idea of defining patterns for Maude metalanguage applications came when we started to develop a new implementation of membrane systems [13] using strategy controllers [3]. The design of the new application is based on adapting some technologies used by other applications and on the Maude strategy language [11]. During the adaptation process we had to answer some standard questions like which part is application dependent and which one is independent. We realized that these questions can be avoided if we follow a problem solving discipline. That was the moment when we started thinking about patterns. The current version of the Maude strategy language includes good design practices which inspired us in defining the patterns presented in this paper. Actually defining the patterns did not prove to be an easy task. We needed to answer many difficult questions: how should a pattern be structured, which pattern should a certain development activity be associated to, which activities are repetitive and which ones need not be performed more than once during the development of a system, how are the patterns related to each other, how should a system implementation process be formalized.

We have identified four software patterns that should help Maude users in building metalanguage applications: *User Interface* (the implementation of a communication flow between the user and the system), *System Language Signature* (the validation of system inputs), *System Language Parser* (the implementation of a Full Maude parser for translating system input to its Maude semantics) and *Error Handling* (the detection and handling of user input errors).

The organization of the paper is as follows. Section 2 presents a template for *patterns*. Section 3 introduces a *general context* for Maude metalanguage applications, including definitions and conventions used throughout the paper. The patterns

presented in Section 4 work under this context. An *iterative strategy* of using the patterns is described in Section 5.

2 A template for Maude patterns

Patterns were first introduced by Christopher Alexander [1] and used in urban design and building architecture. Patterns provide a common language that people use in order to formulate problems and to solve them. Briefly, a pattern describes a design problem, a context in which the problem occurs and the core of a solution to that problem. The same solution sketch can be used by different people in order to solve their own particular problems and speed up the development process.

In software engineering, the patterns are used in various domains such as object-oriented design [9], software architecture [5], software testing [4], and so on. The template we use for Maude patterns includes the following six elements: the pattern *name*, *problem*, *context*, *solution*, *result* and *known uses*.

A well chosen *name* should express in a few words the problem. The *problem* is described using a concise statement in order to help the reader deciding whether the pattern is appropriate for his particular problem or not. The *context* specifies the conditions under which the pattern is applied. We may also mention here the built-in Maude modules or the related patterns involved in the description of the problem. Changing the context should invalidate the pattern. In particular, we have identified a general context corresponding to the family of applications we consider in this paper, namely Maude metalanguage applications. This general context is inherited by each of particular contexts of the patterns. The *solution* describes how the problem is solved. Here we mention the steps one should follow in order to implement the pattern. The description is general enough to address a wide range of situations. The *result* refers to the outcome of applying the pattern. Here we may also mention the related patterns. In the end, the *known uses* subsection emphasize the way some Maude tools have applied the presented pattern. The patterns are flexible enough to be combined in different ways in order to design various applications.

3 General Context

The patterns we define in this paper are intended to be used in the context given by the large family of Maude metalanguage applications developed for specifying and analyzing a specific system. Examples of specific systems are prototypes, simulators, provers, extensions of Maude, logics, models of computations and so on.

Such systems are specified using a specification language, called *system specification language (SSL)*. The implementation consists of defining a Maude semantics for SSL and a user interface for introducing specifications and analyzing and executing these specifications using a language of commands, called *system command language (SCL)*.

We assume that SSL is given by a grammar defining its syntax and a set of

rules or equations defining its semantics. We further assume that the syntax of SSL includes a special construct defining *units*. A unit is intended to describe a component of the system; in particular, the system is described either by a simple unit or by a compound unit. The user interface facilitates the communication between the user and the system: the user may introduce units or commands and the system displays the result obtained after a command is executed.

The commands and the units will be referred to as *top input*, whereas parts of commands or units that match the provided grammar will be referred to as *partial input*.

Maude implementation of a system consists of:

- Maude descriptions of the SSL and SCL semantics,
- the structure describing the states of the metalanguage application in Maude terms,
- a set of rewrite rules over system states modeling the execution of the commands in SCL.

Neither the Maude semantics of SSL, nor that of SCL can be described by means of patterns at this level of generality. However, we suppose that a Maude module describing the SSL and SCL semantics, `SYSTEM-LANG-SEMANTICS`, is defined.

The current system state can be changed or interrogated by using commands from SCL. The execution of a command is guided by the system specification language semantics and depends on the current system state and the provided parameters. Each command *Cmd* is associated an operation `proc`*Cmd* that denotes its semantics.

`proc`*Cmd* : *CurrentState Parameters* → *Result*

We assume that all these operations are included or defined in the previously mentioned `SYSTEM-LANG-SEMANTICS` module. This module can also be thought of as an API of the system. It is used by the *User Interface* pattern but can as well be used by other applications. For instance, the semantics for membranes is given by using the API of the Maude strategy language [11].

The application of the patterns presented in the next section is illustrated by the implementation of a simple system able to perform topological sort. The system (referred to as *TOPO* [4]) is provided first of all with a unit specifying the *definition of a partial order set*. After that, the system is able to interpret a topological sorting command which receives a parameter representing a *linearization of the set to be sorted*. The dialog with TOPO is intended to be as follows:

```
Maude> (poset SIMPLE-POSET is
          rel a < b .
          rel e < b .
          rel b < c .
        end)

Maude> tsort  c d a b e .

result: a d e b c .
```

[4] The implementation of the system is found at: http://circidei.info.uaic.ro/pmma2008/topo.maude

Regarding the implementation, an auxiliary module TOPO-DESCRIPTION is used for describing the core behavior of the system. It defines the universe of the poset (in our case, the letters from a to z) and the sorting algorithm (implemented as a rule that modifies a list of objects according to the partial order relation):

```
ops a b c d e f g h i j k l m n o p q r s t u v w x y z : -> Obj .
crl O :Obj LO:ListOfObjects O':Obj =>
    O':Obj LO:ListOfObjects O :Obj if O':Obj < O:Obj .
```

4 Basic Patterns for Maude Metalanguage Applications

4.1 User Interface

Problem. Define the communication flow between the user and the system under implementation.

Context. Maude uses the loop mode (see [6], Section 17.1) to design user interfaces. The loop mode works with triples [*input*:QidList, *state*:State, *output*:QidList], also called *loop objects*. These triples are provided by the LOOP-MODE Maude module. The *input* argument is the text introduced by the user (if any), the *output* argument is the text displayed by the system (if any) and the *state* argument is the (current) system state.

When handling a user request, the system converts the input stream into a list of quoted identifiers and then places this list on the first position of the loop object. When handling the output, the system unquotes the list of identifiers placed on the third position of the loop object and displays the result. The LOOP-MODE module is used not only for defining user interfaces but also for defining the system state structure and the rewrite rules used for modifying this state and for interacting with the loop. The system state structure can be defined in different ways for Maude metalanguage applications, by importing LOOP-MODE in the module used for state definition.

Solution. The solution is based on defining three modules in Maude used for introducing the system grammar and defining the system state structure in order to specialize the system loop:

```
mod META-SYSTEM-LANG-SIGN is
  including META-LEVEL .
  op SYSTEM-GRAMMAR : -> FModule .
  define the SYSTEM-GRAMMAR module
endm

mod SYSTEM-STATE-HANDLING is
  including SYSTEM-LANG-SEMANTICS .
  define the state structure
endm
```

```
mod SYSTEM-INTERFACE is
  including LOOP-MODE .
  including META-SYSTEM-LANG-SIGN .
  including SYSTEM-STATE-HANDLING .
  define the initial state
  define the input rule
  define the output rule
endm
```

In what follows we describe how each of the above modules is implemented.

META-*SYSTEM*-LANG-SIGN. A module which specifies the system grammar is defined at metalevel (*SYSTEM*-GRAMMAR). This module imports the definition of the system language signature (*SYSTEM*-LANG-SIGN - see Section 4.2) and defines some

system-specific operators, like those able to handle input tokens (identifiers) or bubbles (lists of identifiers) (see [6], Section 17.4).

If avoiding the definition of the system grammar from scratch is desired, the existing Full Maude grammar can be used. By doing this, the created system will be able to handle anything the Full Maude system can. This is accomplished by appending *SYSTEM*-LANG-SIGN to the provided GRAMMAR module. The addImports metalevel appending operator is defined within the UNIT module and GRAMMAR is defined at metalevel within the META-FULL-MAUDE-SIGN module.

```
eq SYSTEM-GRAMMAR = addImports((including 'SYSTEM-LANG-SIGN .), GRAMMAR) .
```

When implementing the TOPO system, we choose to use the Full Maude provided grammar:

```
fmod META-TOPO-LANG-SIGN is
  including META-LEVEL .
  protecting META-FULL-MAUDE-SIGN .
  op TOPO-GRAMMAR : -> FModule .
  eq TOPO-GRAMMAR = addImports((including 'TOPO-LANG-SIGN .), GRAMMAR) .
endfm
```

SYSTEM-STATE-HANDLING. The best way to define the system state structure is inspired from Full Maude (see [6], Section 18.6) and it consists of using a Maude class:

```
class SystemStateClass |
  input : TermList,
  output : QidList,
  attribute1 : Type1
  ...
```

In practice, a "compiled" form of this class is used when the system state structure is defined (see [12], Section 12.4.2). This is accomplished by declaring the class identifier, an operator of sort SystemStateClass and the attributes (including input and output) in an explicit way:

```
subsort SystemStateClass < Cid .
op SystemState : -> SystemStateClass .
op input :_ : TermList -> Attribute .
op output :_ : QidList -> Attribute .
op attribute1 :_ : Type1 -> Attribute .
...
```

The input and output attributes are used in order to create a user interface. This way we can provide a mechanism for passing the user input to the value of the system state input attribute and also for forwarding the system state output attribute's content to the user output.

Instead of defining TopoStateClass, the TOPO system makes use of the Full Maude-specific DatabaseClass sort in order to characterize the current state. The only extra attribute added to the state structure is defaultPOSet, the name of the currently selected module defining the partial order relation:

```
op defaultPOSet :_  :  Header -> Attribute .
```

SYSTEM-INTERFACE. System loop specialization consists of specifying the initial state, the rule processing the user input and the rule displaying the output messages.

Assuming that `SystemStateClass` is the only class used for defining the system states, we consider the relation `Object < State` in order to specify the admissible states within the persistent state of `LOOP-MODE`. The initial state of the system is defined in the same way as in Full Maude:

```
subsort Object < State .
op o : -> Oid . --- the persistent system state object
op init : -> System .
rl [init] : init => [nil, < o : SystemState | input : nilTermList, output : nil,
                            attribute1 : initialValue1, ...
                          >, ('State 'Initialization 'Succeeded)] .
```

The concrete initialization rule from the TOPO system is:

```
rl [init] : init => [ nil, < o : Database | db : initialDatabase,
                             input : nilTermList, output : nil,
                             defaultPOSet : nullHeader
                           >, ('TOPO 'State 'Initialization 'Succeeded)] .
```

Here, the Full Maude-specific `Database` constant is used to instantiate the state class `DatabaseClass`. The inherited attribute `db` contains detailed information regarding Full Maude loaded units. Our attribute `defaultPOSet` receives the constant value `nullHeader`, denoting that in the initial state no module is selected to describe a partial order relation.

The input rewrite rule `[in]` parses the text introduced by the user. It calls the `metaParse` operator and sets the resulting term as the `input` attribute of the next state.

```
crl [in] :
  [ Q QL,
    < o : SystemState | input : nilTermList, Atts >, QL']
  =>
  [ nil,
    < o : SystemState | input : getTerm(RP), Atts >, QL']
  if RP := metaParse(SYSTEM-GRAMMAR, Q QL, '@Input@) ∧
     RP :: ResultPair .
```

The output rewrite rule `[out]` transfers the value of the `output` attribute to the system output:

```
rl [out] : [ QL, < o : SystemState | output : (Q QL'), Atts >, QL'' ] =>
           [ QL, < o : SystemState | output : nil, Atts >, (Q QL' QL'') ] .
```

The template presented for these rules is applied directly when implementing the TOPO system: *System*`State` is replaced by `Database` and *SYSTEM*`-GRAMMAR` is replaced by `TOPO-GRAMMAR`.

Result. Three modules used for defining the system grammar and specializing the loop: *SYSTEM*`-GRAMMAR`, *SYSTEM*`-STATE-HANDLING` and *SYSTEM*`-INTERFACE`. The only thing that remains to be done in order to initialize the loop after having loaded the specified modules is to call the `loop init .` command from the Maude environment.

The *SYSTEM*`-STATE-HANDLING` module can be refined by applying the *Error Handling* pattern (Section 4.4).

Known uses. All the Maude tools referred to in Section 1 apply this pattern. The

only tool that applies the pattern by defining all required data within the same module is ITP (it is not implemented using the exemplified modules structure).

The MTT, CRC, RTM and STR tools define the grammar module within a `META-`*`SYSTEM`*`-LANG-SIGN`-like module. MTT, CRC, RTM and STR extend the Full Maude grammar while ITP defines its own grammar from scratch. STR defines a second grammar from scratch for internal use.

RTM and STR define new attributes for the system state in the manner described by the pattern. The attributes are defined in a *`SYSTEM`*`-STATE-HANDLING`-like module. The ITP tool defines attributes using an internal defined sort, other than `Attribute`. All the tools define the `[init]`, `[in]` and `[out]` rules in a *`SYSTEM`*`-INTERFACE`-like module.

4.2 System Language Signature

Problem. Define the system language signature used in order to validate system inputs (a system input is either a unit or a command).

Context. When referring to the system language signature both the system specification language signature and the system command language signature are considered. For these languages it is necessary to know the grammars defining their syntax.

Solution. The general idea of this pattern is to define the modules:

```
fmod SYSTEM-SPEC-LANG-SORTS is
  declare the metavariables
endfm

fmod SYSTEM-SPEC-LANG-SIGN is
  including OPERATOR-ATTRIBUTES .
  protecting SYSTEM-SPEC-LANG-SORTS .
  declare constructors for metavariables
endfm
```

```
fmod SYSTEM-CMD-LANG-SIGN is
  including COMMANDS .
  declare command operators
endfm

mod SYSTEM-LANG-SIGN is
  including SYSTEM-SPEC-LANG-SIGN .
  including SYSTEM-CMD-LANG-SIGN .
endm
```

`SYSTEM``-SPEC-LANG-SORTS`. This module declares the metavariables from the *SSL* grammar: for each metavariable *MetaVar*, a sort `@`*MetaVar*`@` is defined. Whenever a list of metavariables of sort `@`*MetaVar*`@` is desired, a view must be considered:

```
view @MetaVar@ from TRIV to SYSTEM-SPEC-LANG-SORTS is
   sort Elt to @MetaVar@ .
endv
```

These views are used for implementing the parameterized modules defining the generic data types for lists. Now `List{@`*MetaVar*`@}` is a sort representing a list of separated metavariables, denoted in the SSL grammar by *MetaVar**.

The metavariables from the TOPO sytem are `POSet` and `DeclRelation`. The former one represents a unit defining a partial order relation, while the latter represents an element of the relation (recall the example presented at the end of Section 3). The `TOPO-SPEC-LANG-SORTS` module includes declarations for the sorts `@POSet@` and `@DeclRelation@`. A view for the `@DeclRelation@` is created in order to specify the list of elements characterizing the partial order relation associated

nwith the set.

SYSTEM-SPEC-LANG-SIGN. This module includes an operator declaration for each metaexpression occurring in the specification language grammar. For instance, the operator for a metaexpression of the form *resWord1 metaVar1 resWord2 metaVar2 resWord3* is defined as follows:

```
op resWord1_resWord2_resWord3 : @MetaVar1@ @MetaVar2@ -> @ResultMetaVar@ .
```

When `@`*MetaVarX*`@` is used for specifying an identifier or a list of identifiers, it is replaced by either `@Token@` or `@Bubble@`, respectively. These sorts are declared in the Full Maude-specific module `OPERATOR-ATTRIBUTES`, which must be imported.

The operators declared for the TOPO system are:

```
op poset_is_end : @Token@ List{@DeclRelation@} -> @POSet@ .
op rel_<_. : @Token@ @Token@ -> @DeclRelation@ .
```

In order to be handled as valid system input (see the `[in]` rule in Section 4.1), each sort denoting a top metavariable (a metavariable corresponding to a top input, see Section 3) must be declared as a subsort of the predefined `@Input@` sort:

```
subsort @TopMetaVar@ < @Input@ .
```

For the TOPO system, `@`*TopMetaVar*`@` is replaced by `@POSet@`.

SYSTEM-CMD-LANG-SIGN. This module is used for command declarations. For each command a new operator defining its signature must be added. If, for instance, the form of the command is *resWord param1 param2* ., the operator defining its signature is:

```
op resWord__. : @Param1@ @Param2@ -> @Command@ .
```

In most cases, commands receive basic identifiers or lists of basic identifiers as parameters, meaning that `@`*ParamX*`@` is either `@Token@` or `@Bubble@`, respectively. Commands are operators of the predefined `@Command@` sort. The declaration of this sort is found in the Full Maude-specific `COMMANDS` module.

The TOPO system is able to interpret two commands - one for setting the default module defining the partial order relation and one for actually performing the topological sort:

```
op set`default`poset_. : @Token@ -> @Command@ .
op tsort_. : @Bubble@ -> @Command@ .
```

SYSTEM-LANG-SIGN. This module includes both the specification language signature and commands language signature modules. It is used at metalevel for defining the grammar module: *SYSTEM*-GRAMMAR (see Section 4.1).

Result. The *SYSTEM*-SPEC-LANG-SORTS module used for defining the specification language sorts and three other modules: *SYSTEM*-SPEC-LANG-SIGN for the specification language signature, *SYSTEM*-CMD-LANG-SIGN for the commands language signature and a module combining them both, *SYSTEM*-LANG-SIGN.

Two other patterns are related to this one: *System Language Parser* (see Section 4.3) and *Error Handling* (see Section 4.4). They use the language signature in order to handle and validate the system input.

Known uses. All the Maude tools referred to in Section 1 define their signatures in the manner resembling the one described by this pattern. ITP defines the signature within the *SYSTEM*-INTERFACE-like module, using an internal defined 'Input sort, instead of deriving from the @Input@ provided module.

In one or more dedicated *-SIGN-like modules, MTT, CRC, RTM and STR define commands. They all use the provided @Command@ sort in order to declare commands. RTM and STR define unit input-like signature, the former using the provided @Module@ sort and the latter using an internal defined sort. STR does not define a *SYSTEM*-SORTS-like module because it defines the external unit corresponding sorts within the *SYSTEM*-LANG-SIGN-like module.

4.3 System Language Parser

Problem. Develop a parser in Full Maude for transforming the input matching the system language grammar into a semantics in terms of the Maude language.

Context. We assume that the system specific loop is implemented (see Section 4.1) and the system signature is defined (see Section 4.2). The parsing of the input requires the use of the metaParse operator, already mentioned in Section 4.1.

The presentation on how to write a parser for the SSL is made for the following simple generic grammar (recall from Section 4.2 that *MetaVar** represents a list of separated metavariables):

MetaVar ::= *MetaExpr*
MetaExpr ::= *resWord1 MetaVar1 resWord2 MetaVar2 ... resWordEnd* |
*MetaVar** | ...

The grammar for the TOPO system specification language is:

POSet ::= poset *Name* is *Relation** end
Name ::= *Identifier*
Relation ::= rel *LHS* < *RHS* .
LHS ::= *Obj*
RHS ::= *Obj*
Obj ::= a | b | ... | z

Solution. Each top or partial input must be handled by its own parsing operator. Besides these operators, a set of rules which transfer the input to the corresponding top unit parsing operator must be implemented. This is achieved by defining a parsing-dedicated module and writing extra code for the previously defined *SYSTEM*-STATE-HANDLING module (see Section 4.1):

```
mod SYSTEM-LANG-PARSER is
  including SYSTEM-LANG-SIGN .
  define parsing operators for metavariables
endm
```

```
mod SYSTEM-STATE-HANDLING is
  including SYSTEM-LANG-PARSER .
  ...
  define parsing rules for each top input
  ...
endm
```

SYSTEM-LANG-PARSER. Let *TopMetaVar* and *PartialMetaVar* denote a top and a partial input, respectively, related to a unit. The parsing operators that must be added are:

```
parseTopMetaVar : Term Term ... Term -> MaudeCodeSort .
parsePartialMetaVar : Term MaudeCodeSort -> MaudeCodeSort .
```

Here, *MaudeCodeSort* is the result of the parsing operation and represents the mapping of the input unit to Maude specific code. For instance, a unit from SSL can be mapped to a Maude module.

A top input parsing operator must make calls to partial input parsing operators in this manner:

```
parseTopMetaVar(T1, T2, ..., Tn) =
                    parsePartialMetaVar1(T1,
                      parsePartialMetaVar2(T2,
                        ...
                          parsePartialMetaVarN(Tn, initialMaudeCode)...)) .
```

An input parsing operator `parse`*MetaVar* can handle the input in a few different ways:

a) it can use the provided term "as is" (*e.g.*, as a `QID`, without any restrictions) and generate the related Maude code (see Fig. 1.*a*)

b) it can use `metaParse` in order to check if it corresponds to a previously defined signature and to generate the Maude code (see Fig. 1.*b*)

c) it may call other parsing operators if the term is the metarepresentation of a list of terms (see Fig. 1.*c*)

a)
```
ceq parseMetaVar(T, ...) =
      useQidDirectly(QID)
    if T' := 'token[T] ∧
      QID := downTerm(T', 'nil) ∧
      ... .
```

b)
```
ceq parseMetaVar(T, ...) =
      useParsedTerm(S)
    if T' := 'bubble[T] ∧
      QL := downTerm(T', 'nil) ∧
      RP := metaParse(MetaModule,
        QL, 'DesiredSort) ∧
      RP :: ResultPair ∧
      S := getTerm(RP) ∧
      ... .
```

c)
```
eq parseMetaVar('__[T], ...) =
   parseTermList(T) .
```

Fig. 1. Partial input handling

For the TOPO system, the parsing operators decompose the input unit in order to obtain the elements of the partial order relation. The elements are transformed into Full Maude-specific equations and the unit itself is transformed into a Full Maude module. This module imports the `TOPO-DESCRIPTION` module introduced in Section 3 in order to be able to recognize the objects the relation is defined on. The parsing operators are:

```
op parsePOSet : Term Term ~> Module .
op parseDeclRelation : Term Module ~> Module .

eq parsePOSet(T, T') = parseDeclRelation(T',
     addImports((including 'BOOL .
                 including 'TOPO-DESCRIPTION .),
                setName(emptySModule, parseHeader(T)))) .

eq parseDeclRelation('__[T], M) = parseDeclRelation(T, M) .
eq parseDeclRelation('__[T, TL], M) =
     parseDeclRelation('__[TL], parseDeclRelation(T, M)) .

ceq parseDeclRelation('rel_<_.['token[Q], 'token[Q']], M) =
```

```
addEqs(eq '_<_[T, T'] = 'true.Bool [none] ., M)
if RP := metaParse(upModule('TOPO-DESCRIPTION, false),
                   downTerm(Q, 'nil), 'Obj) ∧
   RP :: ResultPair ∧
   T := getTerm(RP) ∧
   ... --- same for RP', Q' and T'
```

SYSTEM-STATE-HANDLING. For a top input constructor of the form *resWord1 MetaVar1 resWord2 MetaVar2 ... resWordN MetaVarN resWordEnd*, defined in one of the system language signature modules, the following parsing rule is added:

```
crl [parseTopMetaVar] :
  < ... input : ('resWord1_resWord2_...resWordN_resWordEnd[T1, T2, ..., Tn]) ... > =>
  < ... input : nilTermList ... >
  if ... M:MaudeCodeSort := parseTopMetaVar(T1, T2, ..., Tn) ... .
```

Sometimes the parsing rule can handle the input by its own, with or without making use of the `metaParse` operator. This usually happens when parsing commands. Obviously, no extra parsing operators must be added in this case.

The rules added to the state handling module from the TOPO system are:

```
--- Transforms the input order specification unit into a Full Maude module
--- and adds it to the database.
crl [parsePOSet] :
  < O : X@Database | db : DB, input : ('poset_is_end[T, T']),
    output : nil, Atts > =>
  < O : X@Database | db : insTermModule(getName(M), M, DB),
    input : nilTermList, output : ('Introduced 'order 'spec. getName(M)), Atts >
  if M := parsePOSet(T, T') .

--- Uses the procRew metalevel operator in order to apply the rewriting rule
--- that performs the topological sort on the objects provided on input.
crl [parseCommand--topo] :
  < O : X@Database | db : DB, input : (Q[T]),
    defaultPOSet : H, output : nil, Atts > =>
  < O : X@Database | db : DB, input : nilTermList,
    output : (QL'), defaultPOSet : H, Atts >
  if (Q == 'tsort_.) ∧ QL := downTerm(unBubble(T), 'nil) ∧
     RP := metaParse(upModule('TOPO-DESCRIPTION, false), QL, 'ListOfObj) ∧
     RP :: ResultPair ∧ M := getFlatModule(H, DB) ∧
     QL' := procRew(H, M, T, none, DB) .

--- Sets the default module that specifies the partial order set.
crl [parseCommand--set-default-poset] :
  < O : X@Database | db : DB, input : (Q[T]),
    output : nil, defaultPOSet : H, Atts > =>
  < O : X@Database | db : DB', input : nilTermList,
    output : ('Default 'order 'set 'to: H'),
    defaultPOSet : H', Atts >
  if (Q == 'set‘default‘poset_.) ∧
     < DB' ; H' > := evalModExp(downTerm(unToken(T), 'nil), DB) .
```

Result. The *SYSTEM*-LANG-PARSER module defining parsing operators and the enrichment of the state handling module with rules dedicated to transforming top input into Maude code.

This pattern is related to *Error Handling* pattern (Section 4.4) because each parsing operation must be enhanced in order to detect and handle syntactically incorrect user input.

Known uses. The Maude tool which follows most of the steps described by this pattern is STR. It creates parsing rules for the unit-like input within the system state handling module. The operators dedicated to unit parsing are also declared in this module (no *SYSTEM*-LANG-PARSER is created). The other tool defining internal units (RTM) uses some parsing operators provided by Full Maude.

MTT, RTM and STR implement parsing rules for commands processing in the system state handling module. ITP and CRC implement parsing equations instead of rules. The parsing equations/rules from all tools make calls to internal defined processing operators or to operators provided by Full Maude.

4.4 Error Handling

Problem. Detecting and handling errors resulting from syntactically incorrect user input.

Context. User input should always be checked for errors. Otherwise, the result would be a bad system behavior or even a deadlock. Let us try to understand what happens when syntactically incorrect user input is provided.

As stated in Section 4.1, one of the predefined attributes of the system state structure is the `input` attribute. Its value is the metarepresentation of the current user input. At initialization, this attribute receives the `nilTermList` value. The system can only receive input when the `input` attribute has this value (check the `[in]` rule from Section 4.1 for details).

Let us assume, for example, that the user enters a command along with some parameters. The value of the `input` attribute becomes the metarepresentation of that command. If the system is not able to parse one of the provided parameters, the associated rule fails and the `input` attribute remains unchanged (meaning that the attribute does not receive the `nilTermList` value). We say that the state of the system is *unstable* because it is not able to accept any user input.

Therefore an error handling mechanism able to avoid such situations is needed.

Solution. For a unit parsing rule `[parse`*TopMetaVar*`]`, a dual rule able to handle the error is created: `[parse`*TopMetaVar*`-error]`. The new rule fires only when `[parse`*TopMetaVar*`]` cannot be applied because of an input error. The error handling rule must stabilize the system and print an error message to the output.

Let us consider the parsing rule needed to call the `parse`*TopMetaVar* operation. As stated in Section 4.1, `metaParse` can be regarded as the basic metalevel parsing operation. Therefore `parse`*TopMetaVar* either calls this operation itself or some of its suboperators must do so. The `metaParse` operator fails when the returned value is not of sort `ResultPair`, but of sort `ResultPair?`. This is why a parsing operator that makes a direct call to `metaParse` must check its returned value and if a basic parsing failure occurs, an error must be generated.

In order to be able to receive error-prone parameters and return error values, a generic (top or partial) input operator `parse`*MetaVar* must be transformed into a Maude partial operator (see [6], Section 3.5):

```
op parseMetaVar :  Term Term ... -> MaudeCodeSort .
```
is replaced by
```
op parseMetaVar :  Term Term ... ~> MaudeCodeSort .
```
which is equivalent to
```
op parseMetaVar :  [Term] [Term] ... -> [MaudeCodeSort] .
```

An operator for error transmission must also be added:

```
op errorMetaVar :  QidList -> [MaudeCodeSort] .
```

The propagation of an error is achieved by enforcing `parse`*MetaVar* to return the `error`*MetaVar*`(`*error message*`)` value when one of the internal parsing steps fails. An internal parsing step can be either a `metaParse` call or a call to some other parsing operator `parse`*SubMetaVar*. For the latter case, the presented procedure must be applied recursively until all parsing operators are able to intercept and forward the error message:

```
--- direct call to metaParse
ceq parseMetaVar(...) = errorMetaVar(printSyntaxError(RP, QL))
  if ... RP := metaParse(MetaModule, QL, 'DesiredSort) ∧ not(RP :: ResultPair) ... .

--- call to a parsing suboperator
eq parseMetaVar(..., errorSubMetaVariable(QL)) = errorMetaVar(QL) .
ceq parseMetaVar(...) = errorMetaVar(QL)
  if ... errorSubMetaVariable(QL) := parseSubMetaVariable(...) ... .
```

The error handling rule `[parse`*TopMetaVar*`-error]` modifies the system state by operating only on the `input` and `output` attributes. The rule checks if the value returned by `parse`*TopMetaVar* is `error`*TopMetaVar*`(`*error message*`)`. If that is the case, then the `input` attribute receives the `nilTermList` value (in order to stabilize the system) and the `QL` error message is transferred to the `output` attribute of the current state (for feedback).

Sometimes the parsing rule calls `metaParse` directly. In this case the dual rule must check for the operator's failure directly. This is what happens for the particular case in which the user input does not correspond with the system grammar (see Section 4.1). The dual of the input parsing rule `[in]` able to handle errors is:

```
crl [in-error] :
  [ Q QL,
    < o : SystemState | output : nil, Atts >,
    QL' ] =>
  [ nil,
    < o : SystemState |
      output : ('Parsing 'error: printSyntaxError(RP, Q QL)), Atts >,
    QL' ]
  if RP := metaParse(SYSTEM-GRAMMAR, Q QL, '@Input@)) ∧
     not(RP :: ResultPair) .
```

The TOPO system checks for errors when parsing the left hand side and right hand side terms of a relation element:

```
op errorDeclRelation : QidList -> [Module] .
eq parseDeclRelation('__[TL], errorDeclRelation(QL)) = errorDeclRelation(QL).
ceq parseDeclRelation('rel_<_.['token[Q], 'token[Q']], M) =
  errorDeclRelation('Wrong 'LHS: printSyntaxError(RP, downTerm(Q, 'nil)))
  if RP := metaParse(upModule('TOPO-DESCRIPTION, false),
              downTerm(Q, 'nil), 'Obj) ∧ not RP :: ResultPair .
... --- same for the right hand side parsing error handling
```

The rule handling the error is:

```
crl [parsePOSet-error] :
  < O : X@Database | input : ('poset_is_end[T,T']), output : nil, Atts > =>
  < O : X@Database | input : nilTermList, output : (QL), Atts >
  if errorDeclRelation(QL) := parsePOSet(T, T') .
```

Result. A more stable system, able to detect and handle bad user input, making use of a freshly added error handling rule and some related error propagation operators.

Known uses. Most of the Maude tools referred to in Section 1 apply this pattern. ITP implements its own way of handling errors, but the idea of using kinds and implementing partial operations is the same.

CRC and RTM use the error handling operators provided by Full Maude. The operators are used internally in the same manner described in this pattern. SRT fully applies this pattern. The tool implements parsing rules able to detect errors. They make use of error handling operators for each top and partial input.

All the tools check if the user input corresponds with the system grammar. For CRC the check is made directly by the initial `[in]` rule. MTT and STR make this check in the same way described in the pattern. ITP and RTM add two new error checking rules - one for syntax errors and the other for ambiguous input (when the input can be parsed in more ways).

5 Pattern-based Iterations Used in the Development of Maude Metalanguage Applications

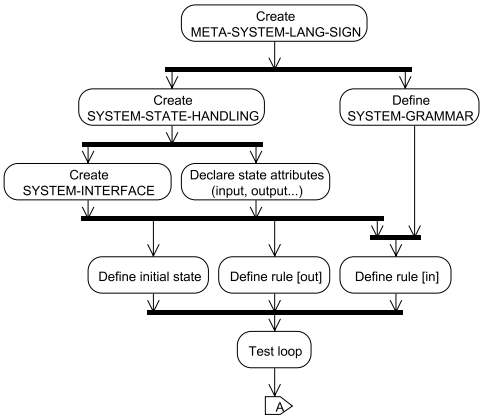
Maude metalanguage applications can be developed by using an iterative strategy. The idea is to build the base version of the system to be implemented and then, at each iteration to add new capabilities to that system. Every time an iteration is performed, the enriched system has to be tested for errors.

The base version of the system is a version the user can interact with. This goal is achieved by applying the *User Interface* pattern (see Section 4.1). At this point a minimal system state structure and the `[in]` and `[out]` rules are defined. These are basic rules that help creating the user interface. This version of the system can be tested by using the "`loop init .`" command. The actions performed during this step are illustrated in Fig. 2a).

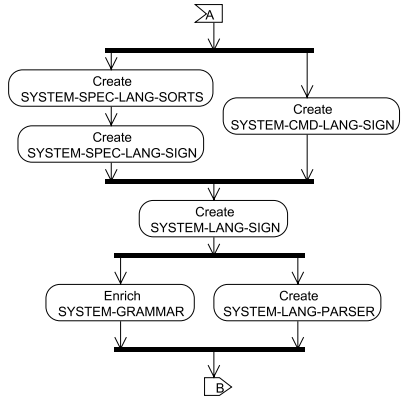
The next step is to create the modules that will contain the system language signature. The structure of these modules is presented in the *System Language Signature* pattern (see Section 4.2). Also the module that will contain parser definitions is created according to the *System Language Parser* pattern (see Section 4.3). Fig. 2b) illustrates the activity diagram corresponding to this iteration.

The system development continues with the signature specification, according to a predefined grammar. Every time the system needs to be enhanced so that it can accept a new command or unit, the *System Language Signature* pattern must be applied. Handling the new input is achieved by enriching the system using the *System Language Parser* and *Error Handling* (see Section 4.4) patterns. The system

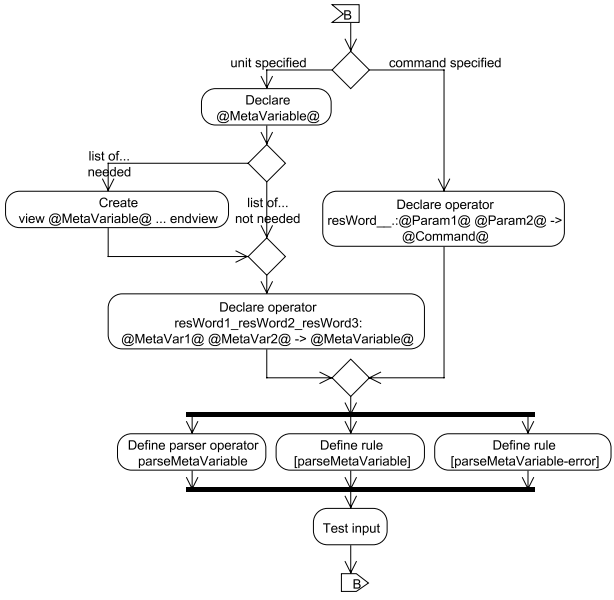
is tested by providing many instances of the freshly defined input and observing whether the response is the expected one or not. The actions performed during this iteration are illustrated in Fig. 2c).



a) User interface



b) System language signature



c) System input handling

Fig. 2. Activity diagrams for Metalanguage Applications development

6 Conclusions

This paper introduces four software patterns useful for developing Maude metalanguage applications for specifying and analyzing systems. The principles that guided us through defining a Maude pattern are:

- it solves a problem (captures solutions),

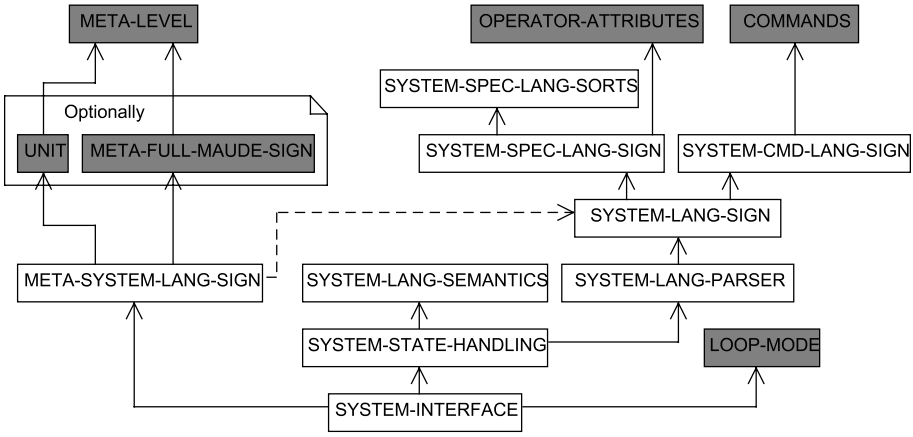


Fig. 3. Overview

- it is a proved concept, not theories or speculation,
- the solution is not obvious (it generates a solution to a problem indirectly),
- it describes a relationship (deep system structure and mechanisms),
- the pattern has a significant human component (minimize human intervention).

Each pattern tackles a different problem that occurs during the implementation of a system. The *User Interface* pattern is applied when defining a specialized *loop*, the *System Language Signature* pattern is used for creating a new input language, the *System Language Parser* pattern is used for parsing units and commands written in the created language, and the *Error Handling* pattern is applied when checking for errors. An overview of the interaction between the modules created during the development of a metalanguage application using these patterns is illustrated in Fig. 3.

The patterns have been tested by a group of three undergraduate students with basic knowledge regarding the Maude system. In a matter of few hours they managed to implement a new system able to receive, parse and interpret user input corresponding to a minimal grammar.

These patterns compose a minimal set that can be extended if more complex systems need to be developed.

Acknowledgement

The authors would like to thank Professor Roberto Bruni and the anonymous reviewers for their valuable suggestions and comments. Many thanks addressed to Elena Naum, Ramona Dunca and Alexandru Ştefan for their assistance and suggestions.

References

[1] Alexander, C., S. Ishikawa and M. Silverstein, "A Pattern Language," Center for Environmental Structure Series **2**, Oxford University Press, New York, NY, 1977.

[2] Andrei, O., G. Ciobanu and D. Lucanu, *A rewriting logic framework for operational semantics of membrane systems.*, Theoretical Computer Science **373** (2007), pp. 163 – 181.

[3] Andrei, O. and D. Lucanu, *Strategy-based proof calculus for membrane systems*, in: *7th International Workshop on Rewriting Logic and its Applications (ETAPS 2008)*, 2007, p. this volume.

[4] Binder, R. V., "Testing Object-Oriented Systems: Models, Patterns, and Tools," Object Technology Series, Addison Wesley, 2000.

[5] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, "Pattern-Oriented Software Architecture - A System of Patterns," Wiley and Sons Ltd., 1996.

[6] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. L. Talcott, "All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic," Lecture Notes in Computer Science **4350**, Springer, 2007.

[7] Clavel, M., F. Durán, J. Hendrix, S. Lucas, J. Meseguer and P. Ölveczky, *The Maude Formal Tool Environment*, in: T. Mossakowski, U. Montanari and M. Haveraaen, editors, *CALCO*, Lecture Notes in Computer Science **4624** (2007), pp. 173–178.

[8] Eker, S., N. Martí-Oliet, J. Meseguer and A. Verdejo, *Deduction, Strategies, and Rewriting*, Electronic Notes in Theoretical Computer Science **174** (2007), pp. 3–25.

[9] Gamma, E., R. Helm, R. Johnson and J. M. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Professional Computing Series, Addison-Wesley, 1994.

[10] Lucanu, D. and G. Roşu, *CIRC : A Circular Coinductive Prover*, in: T. Mossakowski, U. Montanari and M. Haveraaen, editors, *CALCO*, Lecture Notes in Computer Science **4624** (2007), pp. 372–378.

[11] Martí-Oliet, N., J. Meseguer and A. Verdejo, *Towards a Strategy Language for Maude.*, Electronic Notes in Theoretical Computer Science **117** (2005), pp. 417–441.

[12] Meseguer, J., *A logical theory of concurrent objects and its realization in the maude language*, in: G. Agha, P. Wegner and A. Yonezawa, editors, *Reserch Directions in Concurrent Object-Oriented Programming*, The MIT Press, 1992 pp. 314–389.

[13] Paun, G., "Membrane Computing. An Introduction," Springer, 2002.