A Rewriting Logic-based Meta-Model for Design Patterns Formalization

Halima Douibi, Kamel Boukhelfa, Faiza Belala
LIRE Laboratory. University Mentouri of Constantine
Constantine, Algeria
{douibi_halima, boukhelfakamel}@yahoo.fr, belalafaiza@hotmail.com

Abstract— **Informal description of design patterns is adopted to facilitate their understanding by software developers. However, these descriptions lead to ambiguities limiting their correct usage in support tools. Hence, there is a need for formal specification of the design patterns to ensure their successful application. In this paper, we propose a new formalization of design pattern while using a meta-model, based on rewriting logic. The meta-model is encoded in Maude to provide an executable framework allowing experimentation of design patterns models and their formal analysis. Indeed, the relevant elements that constitute a design pattern solution are formally deduced from this formalization.**

Keywords-Design patterns; Meta-model; Rewriting logic; Maude.

I. INTRODUCTION

A design pattern expresses solution of a known and recurrent problem in a particular context. It is applied in object programming software to improve the quality of the expected system [3].

The design patterns are generally described by using a combination of textual descriptions, object oriented graphical notations such as UML's diagrams and sample code fragments. This informal description is adopted to facilitate their understanding by software developers. However, these descriptions lead to ambiguities limiting their correct usage in support tools. Hence, there is a need for formal specification of the design patterns to ensure their successful application. Indeed, this precise and rigorous description permits to achieve the following goals:

- a full understanding of the patterns semantics
- a formal analysis to resolve some issues such as patterns duplication, refinement, disjunction and composition
- a development of the patterns integration into CASE tools.

The formal approaches to design pattern specifications are not intended to replace existing informal approaches, but to complement them.

In this work, we propose a rewriting logic-based meta-model to formalize design pattern solutions and their instantiations. Our proposed meta-model includes all the common elements of design patterns, so any design pattern can be expressed in terms of this meta-model. It provides a high level of abstraction that will cover all the features of design patterns, it allows a generic representation that is used to produce automatically any design pattern specification. Rewriting logic is identified as a semantic basis of our approach since it constitutes an unified semantic framework for many concurrent models. Besides, it has an important property which is reflection allowing powerful meta-programming uses. Intuitively, a logic is reflective if it allows to express at object (or data) level a meta-level (or type level) description. It is used extensively in our meta-model implementation represented as a rewrite logic theory [5]. Hence, we show how this meta-model can be implemented in Maude language. This implementation exploits fully the flexible parser of Maude language and its facilities to define the concrete syntax of the relevant features of our meta-model and its possible analysis.

The rest of this paper is organized as follows: in Section 2, a synthesis on related work for design patterns formalization is given. Then, we present in Section 3 the key concepts of the rewriting logic and its practical Maude language. Section 4 describes on one hand, the proposed meta-model of design patterns, and on the other hand, its encoding in Maude language. Furthermore, an illustrative example is given to elucidate the main idea of our approach. Section 5 concludes this work and presents its perspectives.

II. RELATED WORK

Several attempts to formalize design patterns have been proposed. In this section, we present a brief survey about them focusing especially on the specification formalisms dealing with structural and behaviour aspects of design patterns.

In [9], BPSL (Balanced Pattern Specification Language) language is proposed. This language uses a subset of first-order logic (FOL) to formalize structural aspect of patterns, while the behavioural aspect is formalized in TLA (Temporal Logic of Actions). The first-order logic is justified by its simplicity to express relations between pattern participants as predicates.

In [4], the author presents a use of formal language LePUS (LanguagE for Patterns Uniform Specification) to describe design patterns. LePUS is a fragment of the monadic high-level order logic using a limited vocabulary of entities and relations. A LePUS instruction is formed by a list of participants (classes, functions or hierarchies) and a list of relations between these participants. A program is represented by a model M which is a pair <P, R> where P is the universe of the basic entities (classes and functions) and R=R1,…, Rn is the set of the relations between these entities. These relations are deduced by generalization of all the

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-158-8

existing basic relations between participating entities in the GOF patterns (GOF for "Gang Of Four") [3]. Hence, a design pattern is described by HOL formulae which are accompanied by a graphic representation in order to facilitate its understanding.

Other research works deal with the issues of design patterns integration in CASE tools. We can cite [2] about DPML (Design Pattern Modeling Language) which defines a meta-model and a notation for specifying design pattern solutions and solution instances within object models.
The meta-model defines a logical structure of objects DPML which can be used to create models of design pattern solutions and design pattern solution instances, while the notation describes the diagrammatic notations used to represent visually the models.

At present, DPML allows only specifying the structural aspect of pattern design and no indications are mentioned about the composition and the verification of design patterns. However, instantiation is achieved by mapping from the pattern specification to its realization in a UML design model. The most interesting element of DPML is that it uses a simple set of visual abstractions and readily lends itself to tool support.

Unlike our approach, the most emerging ones are founded on hybrid models and tackle formalization of only some concepts of design patterns which are closed to the object level.

In the present work, we aim to formalize design patterns using a new meta-modeling approach, in which the meta-model represents a part of the global standardized UML meta-model as described by [1]. This meta-model is then integrated in rewriting logic framework.

Our approach differs mainly from the above cited works by the use of a common formalism to specify both the structural and behaviour aspects of design patterns. Moreover, the use of the meta-model concept in design patterns formalization permits to describe all the design pattern features at a same high level of abstraction. In addition, with the encoding of our meta-model in Maude, we obtain executable programs that can be subject of several analysis and verifications.

III. REWRITING LOGIC AND MAUDE

Rewriting logic is known as being logic of concurrent change taking into account the state and the calculus of the concurrent systems. It was shown as a unifying semantic framework of several concurrent systems and models [5][6]. In this context, we can cite without being exhaustive, the labelled transitions systems, Petri nets, CCS, etc.

In rewriting logic, a dynamic system is represented by a rewriting theory $\Re=(\Sigma, E, R, L)$ describing the complex structure of its states and the various possible transitions between them. In rewriting theory definition, (Σ, E) represents an equational membership theory, L is a set of labels and R is a set of labelled conditional rewriting rules. These rewriting rules can be of the following form:

$$(\forall X)\, r{:}t \rightarrow t' \,if \bigwedge_{i\in I} p_i = q_i \wedge \bigwedge_{j\in J} w_j : s_j \wedge \bigwedge_{l\in L} t_l \rightarrow t'_l$$

where r is a labeled rule, all the terms (p_i ,q_i ,w_j ,s_j ,t_l ,t_l') are Σ-terms and the conditions can be rewriting rules, membership equations in (Σ, E), or any combination of both. Given a rewriting theory, we say that $\mathcal{R}$ implies a formula [t]→[t'] if and only if, it is obtained by a finite application of the following deduction rules :

1. Reflexivity:
For each term [t] ∈ $T_{\Sigma,E}(X)$, [t] → [t]
where $T_{\Sigma,E}(X)$, is the set of Σ-terms with variables.

2. Congruence:
For each operator $f \in \sum_n$, n ∈ N,

$$\frac{[t_1] \rightarrow [t_1'] \cdots [t_n] \rightarrow [t'_n]}{[f(t_1,\ldots,t_n)] \rightarrow [f(t'_1,\ldots,t'_n)]}$$

3. Replacement:
For each rewriting rule,
r :[t(x1,…,xn)] → [t'(x1,…,xn)] in R :

$$\frac{[w_1] \rightarrow [w'_1] \ldots [w_n] \rightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \rightarrow [t'(\overline{w'}/\overline{x})]}$$

4. Transitivity:

$$\frac{[t_1] \rightarrow [t_2]\,[t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

Rewriting logic is also a reflexive logic, i.e., aspects of its meta-theory can be represented in a consistent way, namely there is a universal theory U in which any finitely presented rewrite theory R (including U itself) can be presented as a term $\overline{R}$, any terms t; t' in R as terms $\overline{t}$, $\overline{t'}$ and any pair (R,t) as a term $\langle \overline{R}, \overline{t} \rangle$, so that the following equivalence is established : $R \vdash t \rightarrow t' \Leftrightarrow U \vdash \langle \overline{R}, \overline{t} \rangle \rightarrow \langle \overline{R}, \overline{t'} \rangle$

The rewriting logic theoretical concepts are implemented through the Maude language [5][6]. Maude objective is to extend the use of the declarative programming and the formal methods to specify and verify critical and concurrent systems. Maude program is simple and easy to understand. It represents a rewriting theory, i.e., a signature and a set of rewriting rules. The computation in this language corresponds to the deduction in rewriting logic. Maude integrates also equational and object oriented programming, which are used in our formalisation to describe our proposed meta-model, in a convenient way. Its logical basis facilitates a clear definition of the object oriented semantics and makes it good choice for the formal specification of object oriented systems. In this case, a concurrent system is modeled by a multi-set of objects and juxtaposed messages. Concurrent interactions between objects are governed by rewriting rules.

An object is represented by the term < O : C | a1 : v1 , … , an : vn >, where O is the object instance name of class C, ai, I ∈ 1..n, the object attributes names, and vi, their respective values.

The class declaration follows this syntax:
class C | a1 : s1 , … , an : sn .
Where C is the name of the class and si is the sort of the attribute ai. It is also possible to declare sub-classes to benefit from the class inheritance.

Messages are declared using the keyword "msg". The general form of a rewriting rule in the Maude's object-oriented syntax is:

crl [r] : M1 … Mn < O1 : F1 | at1> … < Om : Fm | atm > => < Oi1 : F'i1 | at'i1 > … <Oik : F'ik | at'ik > M1' … Mp' if Cond .

r is the rule label, Ms, s ∈ 1..n, and M'u, u ∈1..p, are messages, Oi, i ∈ 1..m, and Oil, l ∈ 1..k, are objects, Cond is the rule condition. If the rule is not conditional, we replace the keyword crl by rl and we remove the clause if Cond.

Another important aspect which favors the use of Maude language is its implementation through a running environment, allowing prototyping and formal analysis of concurrent and complex systems.

IV. THE META-MODEL FORMALIZATION APPROACH

In this section, we present our design patterns formalization approach based on the rewriting logic formalism. This executable logic is intended to specify both the structural and behaviour aspects of design patterns contrary to the other formalization approaches which use at least two distinct formalisms. Besides, its reflective feature allows us to reason on design patterns meta models instead of their formal specifications only. Thus, the use of the meta-model concept in design patterns formalization permits to describe all the design pattern features at a same high level of abstraction. In the following, we first define our meta-model for specifying design pattern solutions. Then, we show how to encode it in Maude to obtain executable design pattern models that can be subject of several analysis and formal verifications.

A. *The proposed Meta-Model*

To formalize design pattern models and their solutions, we suggest to use a generic notation based on the following meta-model (see Figure 1) as it was done by authors of [1]. Our meta-model defines a logical structure of elements involved in the models conception of the design pattern solutions.

We consider a design pattern solution as a collection of elements and constraints on these elements. An element represents a structural significant part of a design pattern solution. In the object oriented context, this can express a class, an operation or an attribute. Constraints represent conditions that must be verified by an element (i.e., class, operation or attribute). Also, they may represent OCL constraints in object oriented framework.

Thus, the core concept of our meta-model, design pattern model (DP Meta-Model), serves to generate a design pattern solution. The other elements are joined with a set of specific relationship.

We have chosen class diagram notation to represent graphically our meta-model (see Figure 1). Classes and UML relationship (agregation, inheritance, composition, etc.) are used to represent respectively elements and relationship of the meta-model. Thus, the transcription of a class diagram in Maude is easily made thanks to the strong correspondence between UML class concept and the one in Maude [8]. We divided the classes of the solution part of design patterns in several parts (elements, operations, attributes, and constraints). Element class in our meta model represent classes in the solution part, operations of a class are represented by the class operation, the attributes of a class are also represented by a class called attribute. We can see in

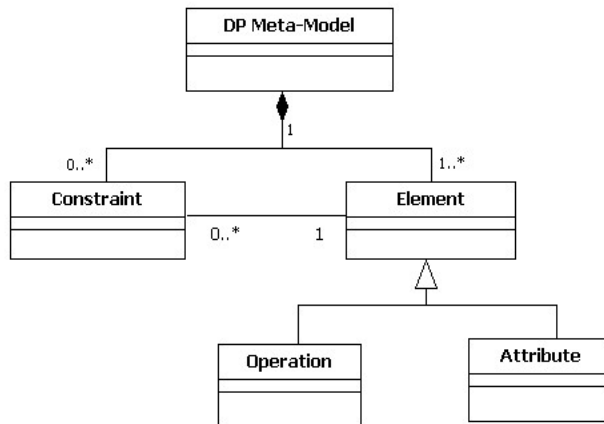

Figure 1. A Meta-model of design patterns

(Figure 1) that the two classes operation and attributes are subclasses of the element class, so each pattern may be composed of a set of elements, operations, attributes and possibly a set of constraints on these elements.

Example: Let us consider as a simple example the Singleton Pattern from [3] (see Figure 2). This pattern is used to ensure that a class has only one instance, and provides a global point of access to it. The solution part of Singleton consists of a single class which contains one attribute called instance and one operation called getinstance().

This pattern can be expressed using our meta-model as it is shown in (Figure 3). In this meta-model there are one main element called singleton which corresponds to singleton class in the solution part of the singleton design pattern, we have also getinstance and instance elements which correspond respectively to the getinstance() operation and instance attribute in the solution part of the singleton design pattern. Finally, we find Return-unique-instance element which represent the constraint imposed by the design pattern.

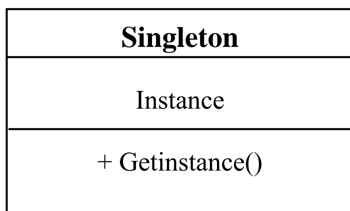

Figure 2. Singleton design pattern

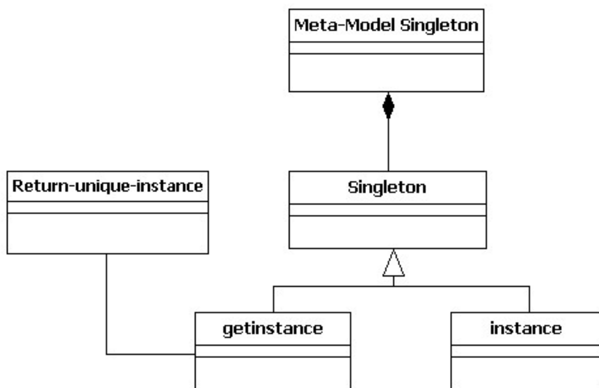

Figure 3. Applying DP Meta-Model to Singleton

B. Encoding the DP Meta-Model in Maude

Our proposed meta-model is formalized using rewriting logic model, we adopt object oriented concepts of this logic to define all the components of the meta-model "DP Meta-Model". The proposed model is described in this case, as being a multi-set of juxtaposed objects and messages, where the concurrent interactions between the objects are governed by rewrite rules. Objects represent elements (participants) of the proposed meta-model that may have some behaviours. Then, the semantics of these object interactions is materialized by the defined messages.

We exploit rewriting logic as a unique semantic formalism for specifying and checking design patterns and their solutions. Thanks to this formalization we lean on the category model to give precise and sufficient semantics to behaviour aspects in design patterns. Besides, this high level specification constitutes an executable one, it allows formal analysis using a particular well-founded language Maude having a proof and prototyping environment.

So, we first give the Meta-model class in our global object oriented Maude module Meta-Model-DP with a specific attribute called Name-ID of sort QID.

```
omod Meta-Model-DP is
  Including QID
  Class Meta-model | NameID : QID .
  ...
endom
```

This will define configurations of concurrent models of possibly several design patterns. Rewriting theory describes static and dynamic aspects of these models. Elements and constraints components in the Meta-model are respectively declared as classes called D-element and D-constraint, having one attribute Meta-Model to relate all elements and also constraints of a given Meta-model. The second attribute of D-constraint class is defined to indicate which element is concerned with this constraint:

```
Class D-element | Meta-Model : oid .
Class D-constraint | Meta-Model : oid,
Element : oid .
```

Elements of type Operation or Attribute in the design patterns meta-model are defined as sub classes of D-element class with an attribute called Element to indicate the operation or the attribute to which element they are attached:

```
Subclass D-operation | Element : oid <  D-
element .
Subclass  D-attribute | Element : oid < D-
element .
```

Thus, we have defined the essential parts of elements or constraints structures involved in the meta-model of design patterns. In addition, we are able to provide an object-message fair rewriting strategy that is well suited for executing objects of Meta-Model DP configurations. For lack of paper space, we have limited our work to the following set of messages which have as role to deal only with the different constructs of the meta-model. For instance, the first message is called get-element, it has one argument of type oid (of class D-element):

```
msg _get-elements :    Oid → Msg .
msg _get-constraints : Oid → Msg .
msg _get-operations_:  Oid Oid → Msg .
msg _get-attributes_:  Oid Oid → Msg .

rl [get-elements]:
< O1 : Meta-model | NameID : S1 >
< O2 : D-element | Meta-Model : O1 > O1get-
elements   =>
< O2 : D-element | Meta-Model : O1 > .

rl [get-constraints]:
< O1 : Meta-model | NameID : S1 >      <
O2 : D-element | Meta-Model: O1 >    < O3 :
D-contraint | Meta-Model : O1 , Element: O2>
O1get-contraints   =>
 < O3 : D-contraint | Meta-Model : O1 ,
Element: O2> .

rl [get-operations]:
< O1 : Meta-model | NameID : S1 > < O2 : D-
element | Meta-Model: O1 >  < O3 : D-
operation | Meta-Model : O1 , Element: O2>
 O1get-operations O2
=> < O3 : D-operation | Meta-Model : O1 ,
Element: O2> .
rl [get- attributes]:
```

```
< O1 : Meta-model | NameID : S1 > < O2 : D-
element | Meta-Model: O1 >  < O3 : D-
attribute | Meta-Model : O1 , Element: O2>
 O1get-attributes O2
=> < O3 : D-attribute | Meta-Model : O1 ,
Element: O2> .
```

These rules are examples of synchronous message passing rules involving one or more objects and one message on the left-hand side. In these examples new objects are created but no new messages are sent. Another rule type involving the joint participation of at least two meta-models of design patterns may be naturally defined in the same way. Note that the multiset structure of the configuration provides the top-level distributed structure of the design patterns meta-model and allows concurrent application of these rules

The proposed model is generic enough, it may be easily enriched to take into account specific patterns as for example those of [3]. We add to the Meta-Model-DP module some constants (operators and equations) to instantiate our meta-model to a given design patterns. We take the same example of Singleton pattern to show what we need to declare in this new module called for this case DP-SINGLETON (see Figure 4). The constant singleton designate a given configuration specifying the design pattern. This configuration is defined through an equation which contains a set of objects that form the pattern participants: Sing, S-class, instance, get-state and S-constraint .

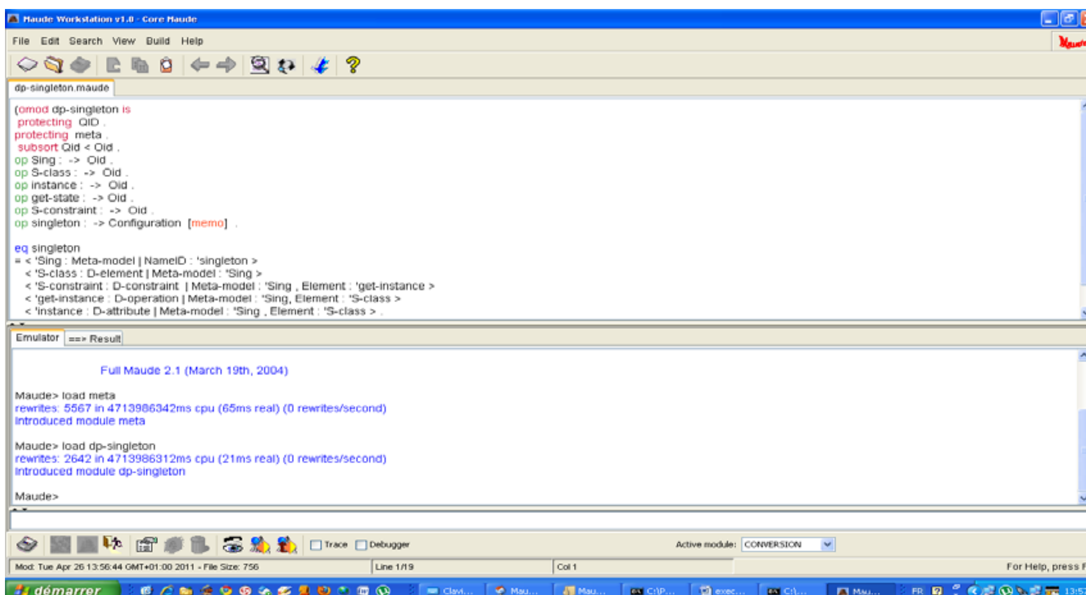

Figure 4. Meta-model module and DP-singleton module

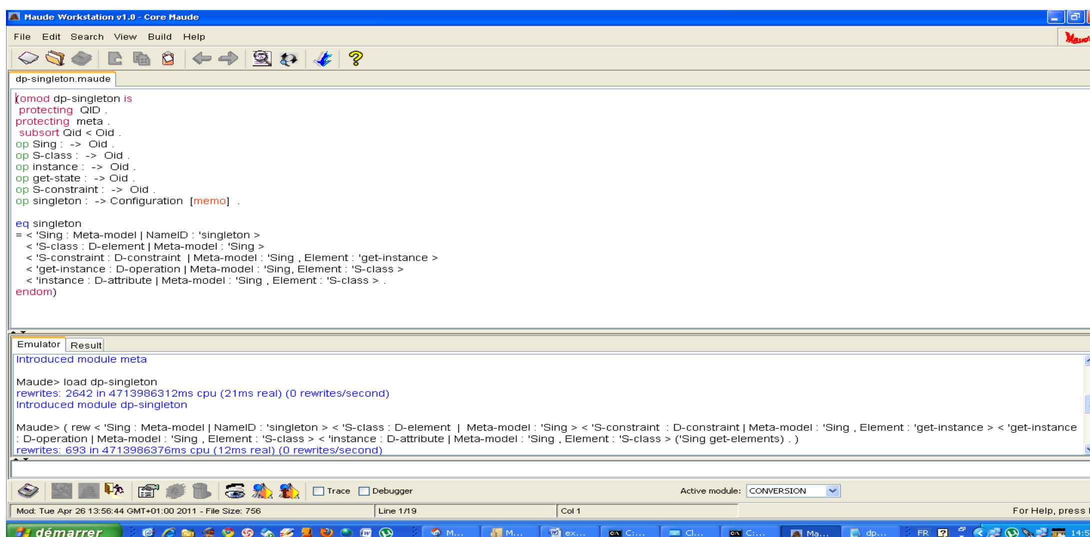

Figure 5. Rewriting configuration example

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-158-8

C. Formal Analysis

The successful implementation of the different modules, previously proposed has been done in Maude workstation 1.0 environment implemented in JAVA. The syntactic verification is implicitly done when the module is loaded (see Figure 4).

We can for instance rewrite any configuration consisting of participants (elements and constraints) of a given design pattern model (or several ones) and some messages with the rewrite command of Maude engine. (Figure 5) presents an example of rewriting a simple configuration to get the elements of the Sing design pattern (Singleton).

This model will enable developers to detect erroneous behaviors and contradictions in the meta models of design patterns. Its main advantage is the clear distinction between the two concerns static and dynamic ones. Thus, firstly we describe the static aspects of the meta-model using Maude objects through their classes. Then, we proceed to the enrichment of this module to ensure its behaviour description. The execution semantic of a model composed of possibly some communicating design patterns models is naturally defined thanks to rewrite rules concurrent execution and Maude module operations. Besides, we can exploit the META-LEVEL predefined Maude module to get the meta representation of these both declared modules and terms (objects and configurations). Thus, we may check if a given design pattern solution respects its pattern or no thanks to metaReduce and metaApply commands.

V. CONCLUSION

The use of formal methods to design patterns is an effective means to improve the reliability and the quality of complex systems. The objective of this work was to adapt one of these methods, largely mastered due to its widespread use in our recent research works, to design patterns model specification and analysis, so that the system development depending of these design patterns solutions can benefit from it.

We have proposed a new approach to formalize design patterns. First we have suggested a meta-model for all design patterns [3]. We considered a design pattern solution as a collection of a participants and constraints on these participants. Our meta-model was defined as a oriented object Maude module, in which all the components of the meta-model are defined as classes. Design patterns can be instantiated from this module. Obtaining the different patterns as dynamic configurations. This will allow us to have multiple patterns in a common configuration.

In the related work section, we have discussed the originality of our proposed approach relatively to other ones that provide languages to formalize design patterns, we have shown through the paper how Rewriting logic (via Maude) offers an accurate way of specifying a meta-model for a generic design pattern and possibly its solutions and provides good tool support for reasoning about them. Our proposed Object Rewrite Theory based model takes benefits from the underlined formalism to consider both static and dynamic aspects of Design Patterns, so it inherits all theoretical advantages of rewriting logic formalism. We do not have need to prove that rewriting logic is better than other formalisms used in this context such monadic high-level order logic, first-order logic (FOL), temporal logic of actions, etc. Besides, rewriting logic has been shown as a logical framework in which several logic have been already integrated.

In future, we will be interested by the study of behavior associated to more complex operations performed on these patterns models such as the composition, the parameterization, etc. On the other hand, defining and encoding in Maude this design pattern meta-model will facilitate the integration of the design patterns into CASE tools. We will take advantage from our meta-modelling approach and the meta model of Maude to deal with Model-to-model transformations (Moment-MT tool, [7]) as technologies looking for reducing the gaps between platform-independent models (PIMs) and platform-specific models (PSMs).

REFERENCES

[1] A. Boronat. “MOMENT: a formal framework for MOdel manageMENT”. PhD in Computer Science, Universitat Polit`enica de Val`encia (UPV), Spain (2007),

[2] D. Mapelsden, J. Hosking, and J. Grundy, “Design Pattern Modelling and Instantiation using DPML”. The 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. “Design patterns: elements of reusable objectoriented systems”. Addison-Wesley, 1995.

[4] E. Gasparis. “LePUS: A Formal Language for Modeling Design Patterns”, Chapter XVI in Design Pattern Formalization Techniques, IGI Publishing (an imprint of IGI Global) (2007), pp. 357-372.

[5] J. Meseguer. “A Logical Theory of Concurrent Objects and its Realization in the Maude Language”. In G. Agha, P. Wegner, and A. Yonezawa, Editors, Research Directions in Object-Based Concurrency. MIT Press, 1992.

[6] J. Meseguer, “Software Specification and Verification in Rewriting Logic”. In M. Broy and M. Pizka, editors, Models, algebras and logic of engineering software, pp. 133-193. IOS Press, 2003. http://maude.cs.uiuc.edu. 7.12.2011

[7] http://moment.dsic.upv.es/. 7.12.2011

[8] K. Boukhelfa, F. Belala, A. Choutri, and H. Douibi, “For more understandable UML diagrams”. In IEEE/ACS International Conference on Computer Systems and Applications (AICCSA) 2010, pp. 1 – 7.

[9] T. Taibi and D. Ngo. “Formal specification of design patterns-A balanced approach”. In Journal of Object Technology, 2, 4. (2003), pp. 127-140.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-158-8