This paper has been accepted at the ECOOP 2001 workshop on Adaptative Objet-Models and Metamodeling Techniques.

Meta-modeling Design Patterns: application to pattern detection and code synthesis

Hervé Albin-Amiot [1], Yann-Gaël Guéhéneuc [2]

École des Mines de Nantes
4 rue Alfred Kastler
BP 20 722
44 307 Nantes cedex 3
France

[1] Soft-Maint
4, rue du Château de l'Éraudière
BP 72 438
44 324 Nantes cedex 3
France

[2] Object Technology International Inc.
2670 Queensview Drive
Ottawa, Ontario, K2B 8K1
Canada

{Herve.Albin-Amiot, Yann-Gael.Gueheneuc}@emn.fr

ABSTRACT: Design Patterns have been quickly adopted by the *object-oriented* community, in particular since the publication of "Design Patterns: Elements of Reusable Object-Oriented Software". They offer elegant and reusable solutions to recurring problems of design. Their use increases productivity and development quality. However, these solutions, at the boundary of programming languages and design models, suffer from a lack of formalism. For this reason, their application remains empirical and *manually* performed. This position paper presents how a meta-model can be used to obtain a representation of design patterns and how this representation allows both automatic code generation and design patterns detection.

KEYWORDS: Design Patterns, code generation, pattern detection

1. Introduction

Since their emergence [1, 16], design patterns have been widely accepted by software practitioners. Their contribution covers definition, design, and documentation of class libraries and frameworks. Even if there is no consensus about the way to support the application of design patterns (using tools, languages, … [15]), we strongly believe that this task should be automated or at least, be assisted. Moreover, manual application is tedious and error prone [20]. In addition, as mentioned in [12], the loss of traceability remains an essential drawback of this *by-hand* (and sometimes hard) coding task. In other words, when a pattern is applied, the resulting implementation does not provide a mean to go back to the pattern for which it was required, the pattern code being mixed within the user's application code.

The application of a design pattern can be decomposed in three distinct activities: 1) the choice of the right pattern, which fulfils the user requirements, 2) its adaptation to these requirements (the term instantiation is commonly used to identify this task), and, 3) the production of the code required for its implementation. Automating these two last tasks is a very challenging issue for the *Design Patterns* community and galvanizes a lot of research works.

Two kinds of works can be distinguished: those using a pattern representation for a given implementation language and those using a representation dedicated to a given modeling language [14]. In both cases, the first issue concerns the necessary extension of an existing base language or the definition of a new one. If patterns are supported at the implementation language level [7, 10], the traceability between the applied patterns and the resulting code is *de facto* ensured [3]. However, this approach suffers from the use of a non-standard implementation language that impedes portability of the applications developed with it. The second approach, based on a modeling language, does not have this limitation but, in most case, does not address either code generation or traceability [4, 6, 11, 19]. In an iterative development, where source code and model are not synchronously manipulated, it is important to provide a univocal link between them. Traceability must absolutely be enforced.

In this paper we present a solution based on a modeling language where code generation and traceability are addressed. Patterns are formalized using a set of basic bricks called *entities* and *elements*. These bricks are defined in the core of a meta-model dedicated to the representation of patterns. In order to evaluate it more rapidly, we have not defined it as an extension of an existing one (like *UML* for example). The proposed meta-model, experimented in *Java*, provides a mean to describe structural and behavioral aspects of design patterns. From this description, it gives the required machinery to produce code and to detect instantiated patterns in code.

The intended contribution of our approach is the reification of design patterns as first-class modeling entities. We use such reified design patterns to produce their associated code implementation, according to the context of their application, and to detect their occurrences in user's code. We deduce the way to apply a design pattern solely from its declaration, not from external hints or specifications. The meta-model we use handles uniformly instantiation and detection of design patterns.

The rest of this paper is organized as follow: Section 2 proposes a short introduction to the meta-modeling technique applied to the patterns representation. Section 3 introduces our meta-model and illustrates, through the Composite pattern example, how we use it to describe a pattern, to instantiate it, to produce its associated code, and to detect it. Section 4 discusses limitations of our approach and finally, section 5 concludes on the use of the meta-modeling technique.

2. Meta-modeling and patterns

Techniques based on meta-modeling consist in defining a set of meta-entities from which a design pattern description is obtained by composition of these entities using an instanciation (pseudo-) link. This composition follows semantic rules, fixed by the relations among meta-entities. From this point of view, meta-modeling is a mean to formalize patterns.

A pattern meta-model does not capture what a pattern is *in general*, but how it is used in one, or several specific cases, for example, application, validation, structural representation, etc. Each kind of use implies the definition of a dedicated meta-model. For example, a fragment-based meta-model for representing patterns structure [4], or a meta-entity-based meta-model for patterns instantiation and validation [19].

A pattern meta-model never "produces" patterns. Instead of, it produces models of patterns. These resulting models are approximations of patterns in the considered use-case. A pattern meta-model ensures that patterns exist as first-class entities. However, it does not guarantee the traceability *per-se*. Nevertheless, it is possible to propose a mechanism of detection that solves the traceability problem.

There exists several meta-models for representing design patterns but none is specifically designed toward code generation and detection. [6] introduces meta-model for design patterns instantiation and validation but without support for code generation. In the *PatternGen* tool [19], the meta-model does not support source code production (another module handles this feature) and it offers no patterns detection. In [4], the fragment-based system allows only design patterns representation and composition.

3. Presentation of the meta-model using a toy example

Throughout this section, we will focus on the Composite pattern. This pattern is relatively simple and is often used as example [14] either for code production [2] or for detection [8, 9]. It composes *objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of object uniformly* [1]. We follow the description presented in the "Implementation" section of [1], in paragraph "declaring the child management operations". This is the most common use of this pattern (for instance, see classes Component and Container of the *JAVA* AWT[1]).

3.1. The meta-model in a nutshell

The meta-model embodies a set of entities and the interaction rules between them. All the entities needed to describe structure and behavior of design patterns introduced in [1] are present. Figure 1 shows a fragment of the meta-model.

A model of pattern is reified as an instance of a subclass of class `Pattern`. It consists in a collection of entities (instances of `PEntity`), representing the notion of participants as defined in [1]. Each entity contains a collection of elements (instances of `PElement`), representing the different relationships among entities. If needed, new entities or elements can be added by specialization of the `PEntity` or `PElement` classes.

[1] Abstract Window Toolkit

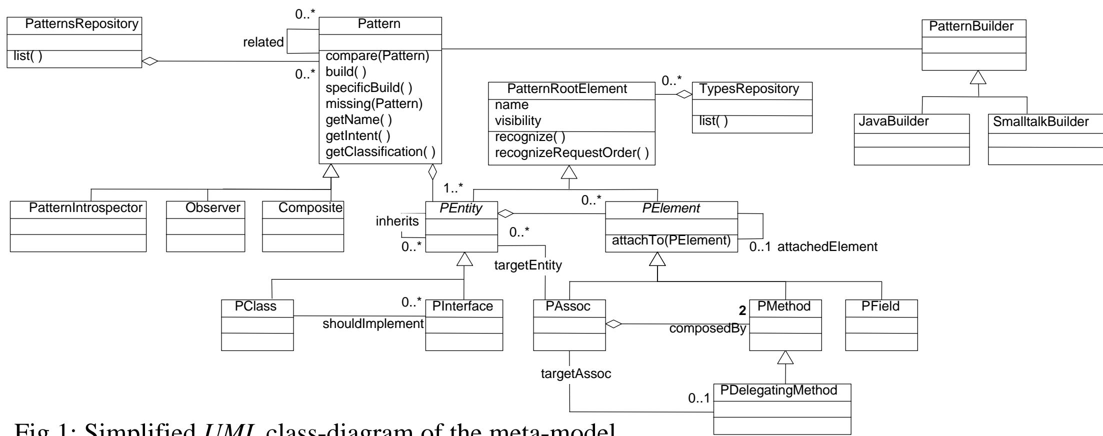

Fig 1: Simplified *UML* class-diagram of the meta-model

The meta-model defines the semantic of patterns. A pattern is composed of one or more classes or interfaces, instances of `PClass` and `PInterface` and subclasses of `PEntity`. An instance of `PEntity` contains methods and fields, instances of `PMethod` and `PField`. The association and delegation relationships are expressed as elements of `PEntity`.

An association (class `PAssoc`) belongs to `PEntity` and references another `PEntity` (this is a simplification that represents only simple relationships like binary and mono-directional associations found in [1]). For example, an association that links a class `B` to a class `A` is defined using two instances of class `PClass`, `A` and `B`, and one instance of class `PAssoc`. The instance of class `PAssoc` belongs to `A` and references `B`.

Delegation is expressed in a similar way using the class `PDelegatingMethod`. For example, the delegation of the behavior of a method `foo` of `A` to a method `bar` of `B` is realized using an instance of class `PDelegatingMethod`. The instance of class `PDelegatinMethod` belongs to `A` and references the method `bar` of `B`. The `PDelegatinMethod` object also references the association between `A` and `B` to deduce from its cardinality nature of the message sent: simple or "multicast".

3.2. Instantiation of the Composite pattern

We now further present the meta-model and its use through the Composite pattern example. Figure 2 presents the general procedure to instantiate a pattern.

① The first step consists in specializing the meta-model to add all the structural and behavioral needed constituents. In the case of the Composite pattern, the meta-model previously presented is sufficient. However, it would be necessary, for example, to add a new `PEntity`, called `ImmutablePClass`, to implement a Composite pattern which leaves are immutable. (In this case, a new subclass `ImmutablePClass` to the class `PClass` would be added.)

② The second step consists in the *instantiation* of the Composite pattern meta-model, built according to the generic meta-model semantic. We call the resulting model as an abstract model because it retains no information about the user's application context. This abstract model corresponds to a reification of the Composite pattern and holds all the needed information related to the pattern.

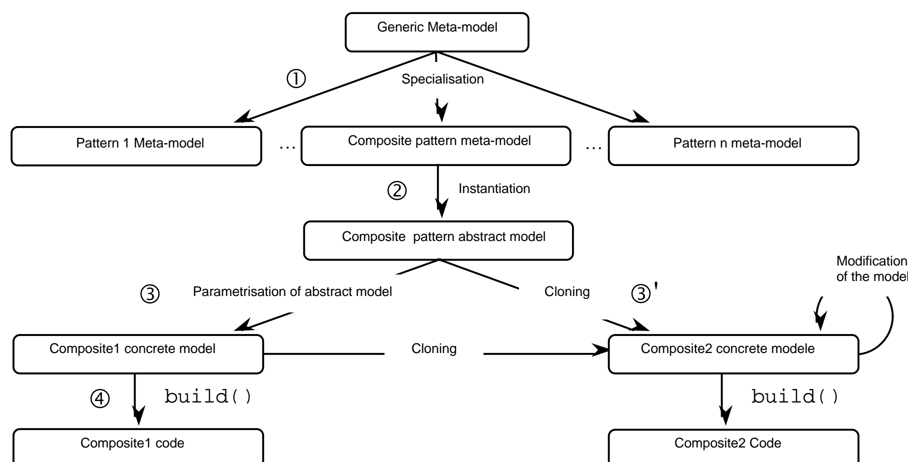

Fig 2: Process of instantiation of the Composite pattern

Thus, the Composite pattern "takes reality" and becomes a first-class object. This differs from [2, 7, 11] and is an important gain of our approach. Because pattern abstract models are first-class objects, it becomes possible to reason and act on them as regular objects. Thus, it is possible to introspect them and modify their structure and behavior both statically and dynamically.

Figure 3a shows the structure of the Composite pattern as defined in [1]. From the structure and the notes, we obtain a new class diagram where all informal indications (*UML Notes*) are explicit. This class-diagram, shown on Figure 3b, is the abstract model of the Composite pattern.

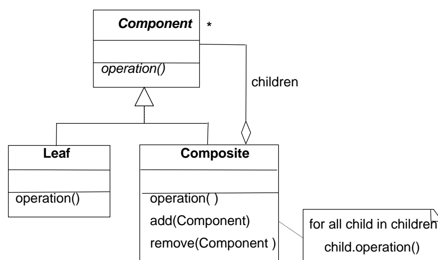

Fig 3a: Structure of the Composite pattern

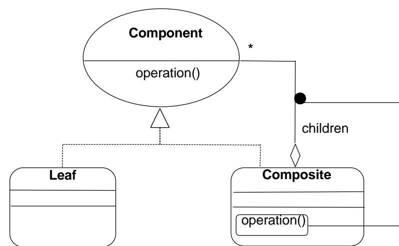

Fig 3b: Composite pattern abstract model

Rounded squares represent instances of `PClass`, *oval, instance of* `PInterface` *and boxed method, instance of* `PDelegetingMethod`. *Instance of* `PAssoc` *and realization link use the UML notation.*

The abstract model is expressed in a declarative manner in the *Java* class `Composite`, subclass of the class `Pattern`. The next table shows a fragment of this declaration dual of the diagram Fig 3b.

Composite pattern abstract model definition
`class Composite extends Pattern {`
Declaration takes place in the Composite class constructor
`Composite(…) {` …
Declaration of the "Component" actor

`component = new PInterface("Component")`
`operation = new Pmethod("operation")`
`component.addPElement(operation)`
`this.addPEntity(component)`

Declaration of the association "children" targeting "Component" actor with cardinality n
`children = new PAssoc("children", component, n)`
Declaration of the "Component" actor

`composite = new PClass("Composite")`
`composite.addShouldImplement(component)`
`composite.addPElement(children)`

The method "operation" defined into "Composite" actor implements the method operation of "Component" actor and is linked to its through the association "children"

`aMethod = new PdelegatingMethod(`
`"operation", children)`
`aMethod.attachTo(operation)`
`composite.addPElement(aMethod)`
`this.addPEntity(composite)`

Declaration of the "Leaf" actor

`leaf = new PClass("Leaf")`
`leaf.addShouldImplement(component)`
`leaf.assumeAllInterfaces()`
`this.addPEntity(leaf)`
}

Declaration of specific services dedicated to the Composite pattern
…
For example, the service "addLeaf" to dynamically adds "Leaf" actor to the current instance of the Composite pattern

`void addLeaf(String leafName) {`
 	
`PClass newPClass = new PClass(leafName)`
`newPClass.addShouldImplement(`
`(PInterface)getActor("Component"))`
`newPClass.assumeAllInterfaces()`
`newPClass.setName(leafName)`
`this.addPEntity(newPCclass)`
 	
}

We can relate such a declarative description of design patterns to the work on *tricks* by Eden, Yehudai, and Gil [18]. With the purpose of automating design patterns application in mind, they define the notion of tricks, language independent constructs that can be seen as links among programming language idioms and design patterns. In our approach, a design pattern is described using low-level design constructs (such as association, delegation…) that are close to tricks.

Abstract models are stored inside a pattern repository (class `PatternsRepository`, Figure 1). This repository helps to access easily to the previously defined design patterns and to assess the relevance of the solution they represent.

③ The third step consists in the instantiation of the abstract model into a concrete model. The concrete model represents the pattern applied to fit user application requirements. To illustrate the instantiation procedure we use an example that has been introduced in [1] and reused in [6]. This example defines a hierarchy of graphical components as shown Figure 4.

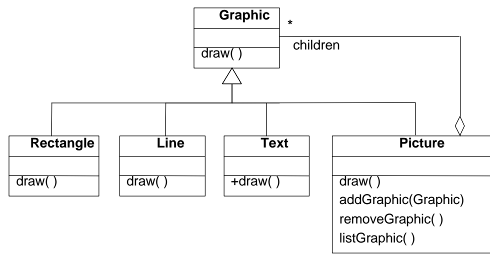

Fig 4: *UML* diagram of the concrete model

The abstract model instanciation is realized by instantiating the class `Composite` defined in ②. Then, this instance is parameterized (each participant or actor of the pattern is named for example) to match the concrete application (Figure 4). An alternative (③' on Figure 2) to this parameterization is a cloning operation. Cloning allows modifications of the manipulated model (abstract or concrete) that shortcuts its normal behavior, while ensuring its integrity by providing a mean to come back to a coherent model.

The following source code is given as an example. We deduce it from the pattern abstract model and the context. A visual manipulation tool (such as ***PatternsBox***[2] not further described therein) dynamically produce this code using user inputs.

Declaration of a new Composite concrete model

```
Composite p = new Composite()
p.getActor("Component").setName("Graphic")
p.getActor("Component").
     getActor("operation").setName("draw")
p.getActor("Leaf").setName("Text")
p.getActor("Composite").setName("Picture")
p.addLeaf("Line")
p.addLeaf("Rectangle")
```

④ The final step consists in code generation. It is automatically performed once the concrete model currently manipulated receives the "`build()`" message (see Figure 1). The *Java* source code obtained from the concrete model diagram Figure 4 is presented below:

```
/* Graphic.java */
public interface Graphic {
  public abstract void draw();
}

/* Picture.java */
public class Picture implements Graphic {
 // Association: children
 private Vector cildren = new Vector();

 public void addGraphic(Graphic aGraphic)
 {children.addElement(aGraphic);}

 public void removeGraphic(Graphic
                          aGraphic)
 {children.removeElement(aGraphic);}
```

```
  // Method linked to: children
  public void draw()
  {for(Enumeration enum =
        children.elements();
        enum.hasMoreElements();
   ((Graphic)enum.nextElement()).draw());
  }
}

/* Text.java */
public class Text implements Graphic {
  public void draw(){}
}

/* Line.java */
public class Line implements Graphic {
  public void draw(){}
}

/* Rectangle.java */
public class Rectangle implements Graphic
{
  public void draw(){}
}
```

3.3. Detection of the Composite pattern

The detection system is designed to work on code produced by instantiation of abstract models and on design patterns implemented *by hand*. It does not use any marking system and does not require any detection-specific information to be present in the code. The detection is based principally on structural information and may be extended to include other information The detection system uses a repository of all the constituents of a pattern (*i.e.* elements and entities). This repository, instance of class `TypesRepository` (Figure 1) contains all the `PEntities` and `PElements` currently defined in the meta-model.

Detection is decomposed in two steps:

① The class `PatternIntrospector`, in charge of the detection, submits all the syntactic elements (classes, interfaces, methods…) found in the user's source code to the types in the `TypesRepository`.

Each type through its `recognize` method detects which constituents match it. Then it passes the remaining not recognized constituents to the following type. The general algorithm is depicted in the table presented on the next page.

Then, the `PatternIntrospector` builds a concrete model that represents the given user's code using only constituents defined into the meta-model. Finally, it solicits each abstract model found in the `PatternsRepository` to determine which pattern(s) has been detected.

[2] Available at http://www.emn.fr/albin

Reconstruction algorithm, class `PatternIntrospector`

Let `C`: list of the user's classes
Let `P`: pattern being recognized
Let `E`: list of existing entities (`Pentity`)
Let `L`: list of existing elements (`PElement`)
Let `S`: list of the syntactic elements (classes, interfaces…) found on the user code.
Let `T`: list of instanced of `Pelement`
Let `U`: class being examined

```
S = C
T = ∅
For each e of E, while size(S) > 0
       S = e.recognize(S, P)
T = P.listPEntities()
For each t of T
       U = Class.forName(t.getName())
       S = U.getDeclaredConstructors() + U.getDeclaredMethods() + U.getDeclaredFields()
       For each l of L, while size(S) > 0
              S = l.recognize(S, P)
```

Method `recognize` of `Pentity` (`e.recognize(S,P)`)

Let `N`: list of non-recognized entities

```
N = S
For each s of S
    If s = e Then
        P.addPEntity(new(s))
        N = N – {s}
Return N
```

Method `recognize` of `Pelement` (`l.recognize(S,P)`)

Let `N`: list of non-recognized elements

```
N = S
For each s de S
    If s = l Then
        P.getActor(s.getDeclaringClass().
                getName()).addPElement(new(s))
        N = N – {s}
Return N
```

② Each abstract model examines and determines which entities (instances of `PEntity`) of the submitted model can be associated to its different roles. The following criteria are applied:

- ❶ A role of the abstract model must be fulfilled by a constituent of same type in the model built from the user's code.
- ❷ The model built from the user's code must contain (at least) as many entities as the abstract model.
- ❸ For each role attributed from the abstract model, the corresponding entities in the model being built must contain (at least) as many elements as the entity from the abstract model.
- ❹ Inheritance links must be present.
- ❺ Realization links must be present.
- ❻ Association links must be present.

The concrete model we submit to the Composite abstract model is presented Figure 6.

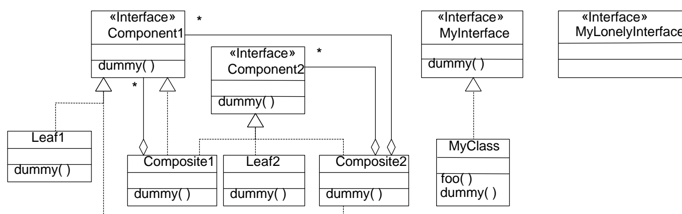

Fig 6: Simplified *UML* class-diagram of the given user code

The abstract model finds the following (partial) results after having applied the three first criteria:

« Component »	« Composite »	« Leaf »

`MyInterface`
`Component1`
`Component2`
	`MyClass`
	`Composite1`
	`Composite2`
		`MyClass`
		`Composite1`
		`Composite2`
		`Leaf1`
		`Leaf2`

Once the three first criteria have been applied to reduce the search space, the algorithm verifies for each remaining criterion (binary constraint) which entity (value), for a given role (variable), has a supporting entity verifying this constraint.

This is similar to the arc-consistency problem in CSP. In our case, if we obtain, after filtering, arc-consistency, at least one valid instance of the pattern represented by the abstract model used for examination has been detected.

We use for filtering the *AC-1* algorithm proposed by Waltz in 1972. Its complexity is $O(n \times e \times d^3)$ with n the number of roles, e the number of criteria and d the maximum number of potential `PEntity` for a role. This algorithm is very costly but very easy to implement and sufficient for evaluation purposes.

After computations, as expected, the algorithm found two instances of the Composite pattern:

```
* Pattern: Composite: 2 instance(s)
Component: Component1
Composite: Composite1, Composite2
Leaf: Leaf1
---
Component: Component2
Composite: Composite2
Leaf: Leaf2, Composite1
```

4. Limitations and future

4.1. Instantiation

The main limitation of our approach concerns the integration of the generated code with the user's code. A solution would be to transform the wanted implementation to fit the user's code. Instantiation would include the generation of the strict implementation of the pattern, and then this implementation would be integrated into the user's source code using source to source transformation. We are currently investigating the definition of a transformation engine (***JavaXL***) able to automatically transform user's source code according to a pattern declaration.

4.2. Detection

Another problem of this approach concerns the integration of the user's code specificity into the detection mechanism. How to distinguish variants of a design pattern? The current approach (based on a constraint programming AC[3]-like algorithm) should be able to make such distinctions. However, we experienced limitations with such a strict constraint system. The solution we are currently investigating (***Ptidej***) is based on an explanation-based constraint system, with dynamic constraint relaxation capabilities. From a strict design pattern declaration and according to some priorities, the constraint system is able to automatically relax unsatisfied constraints to find more solutions.

5. Conclusion

As stated Section 2, several approaches have been proposed to instantiate or to detect design patterns. Those approaches are either at the implementation level or at the design level.

At the implementation level, design patterns are represented through syntactic constructions. Therefore, instantiation and detection are straightforward. The traceability is *de facto* ensured. However, this approach requires the use of a dedicated implementation language, which may be too much a constraint.

At the design level, design patterns are represented through high-level constructs (such as classes, methods…) independently from a specific implementation language. However, instantiation (code generation) is not always supported and there is no insurance for a 1-to-1 correspondence between design constituents and implementation constructs (a unique pattern, at design level, may be spread out into several classes at the implementation level). Thus, there is a need for a mechanism ensuring the traceability.

In this paper, we described a meta-model that offers a way to define patterns at the design level. The structure and properties of a design pattern are defined using the constituents defined in the meta-model. The same abstract model can be used both for instantiation and detection. There is no dissemination of design pattern-related information over the design or the implementation. An abstract model contains all the information related to its instantiation and detection, thus ensuring its traceability.

However, techniques we have presented are valuable for representing patterns essentially from structure-based elements. The main reason is that each meta-entity needs to have code equivalence and ability to detect in code its corresponding syntactic construct. To reach this goal, each description must be structure-based in order to generate code (generation always provides structure and not behaviour) and prevent us from an uncertain dynamical source analysis.

The problem of describing behaviour using structural elements can be addressed using a strict separation between patterns representation and the code producer-detector system using two meta-models. A first meta-model could be used to instantiate pattern using structural and behavioural bricks without code equivalence and a second one would provide bricks to produce and detect architecture parcels. Moreover, this separation would allow us to represent patterns using formalisms such as contracts [17] or constraints applied on role and actor, even if this formalism does not retain sufficient information to produce code. The main drawback of this solution is its complexity: it requires definition of several interacting meta-models.

[3] Arc consistency

6. Bibliography

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides ; Design Patterns : Elements of Reusable Object-Oriented Software ; Addison-Wesley Professional Computing Series, 1995.

[2] F. J. Budinsky, M. A. Finnie, J. M. Vlisside, P. S. Yu ; Automatic code generation from design patterns ; IBM Systems Journal, vol. 35, no. 2, 1996.

[3] G. Baumgartner, K. Läufer, V.F. Russo ; On the Interaction of Object-Oriented Design Patterns and Programming Languages ; Technical Report CSD-TR-96-020, Purdue University (USA), 1996.

[4] Gert Florijn, Marco Meijers, Pieter van Winsen ; Tool Support for Object-Oriented Design Patterns ; ECOOP'97, pages 472-495, 1997.

[5] C. Krämer, L. Prechelt ; Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software ; WCRE'96, 1996.

[6] B.-U. Pagel, M. Winter ; Towards Pattern-Based Tools ; EuropLop'96, 1996.

[7] J. Bosh ; Design Patterns & Frameworks : On the Issue of Language Support ; LSDPF'97 : Workshop in ECOOP'97, 1997.

[8] Roel Wuyts ; Declarative Reasoning about the Structure of Object-Oriented Systems ; TOOLS-USA'98, 1998

[9] K. Brown ; Design reverse-engineering and automated design pattern detection in Smalltalk ; Technical Journal, 1995.

[10] M. Tatsubori, S. Chiba ; Programming Support of Design Patterns with Compile-time Reflection ; OOPSLA'98 Reflection Workshop, 1998.

[11] P. Desfray ; Automation of Design Pattern : Concepts, Tools and Practices ; UML'98 : Beyond the Notation, Bézivin - Muller (Eds.), Lecture Notes in Computer Science, vol. 1618, Springer-Verlag, 1999.

[12] J. Soukup ; Implementing Patterns ; Pattern Languages of Program Design, pages 395-412, Coplien - Shmidt (Eds), Addison-Wesley, 1995.

[13] W. Zimmer ; Relationships Between Design Patterns ; Pattern Languages of Program Design, pages 345-364, Coplien - Shmidt (Eds), Addison-Wesley, 1995.

[14] G. Sunyé, A. Le Guennec, J.-M. Jézéquel ; Design Pattern Application in UML ; ECOOP'00, pages 44-62, 2000.

[15] C. Chambers, B. Harrison, J. Vlissides ; A Debate on Language and Tool Support for Design Patterns ; POPL'00, pages 277-289, 2000.

[16] W. Pree ; Design Patterns for Object-Oriented Software Development ; ACM Press, Addison-Wesley, 1995.

[17] J.-M. Jézéquel, M. Train, C. Mingins ; Design Patterns and Contracts ; Addison-Wesley, 2000.

[18] A. H. Eden, Y. Gil ; Precise Specification and Automatic Application of Design Pattern ; ASE'97, 1997.

[19] G. Sunyé ; Génération de code à l'aide de patrons de conception ; LMO'99, pp. 163-178, 1999. In French.

[20] Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brössler, and Lawrence G. Votta ; A controlled experiment in maintenance comparing design patterns to simpler solutions ; IEEE Transactions on Software Engineering, 2000.