Declarative Reflection and its Application as a Pattern Language

A. Herranz [1] J.J. Moreno [2] N. Maya [3]

School of Computer Science
Technical University of Madrid
Campus de Montegancedo s/n
Boadilla del Monte 28660
Spain

Abstract

The paper presents the reflection facilities of the specification language Slam-sl. Slam-sl is an object oriented specification language where class methods are specified by pre and postconditions. The reflection capabilities allow managing these pre and postconditions in specifications what means that semantic reflection is possible. The range of interesting applications is very wide: formal specification of interfaces and abstract classes, specification of component based software, formalization of design pattern, using Slam-sl as a pattern language, etc. The paper discusses the last two advantages in some detail.

1 Motivation

We have recently presented The SLAM Project [13,20], a system which includes an object oriented specification language, Slam-sl, that is supported by a development environment that, among other features, is able to generate readable code in a high level object oriented language. The Slam-sl language is a formal specification language that *integrates algebraic specifications* (like those proposed by the OBJ language family [10,4] or Larch-LSL [11]), and *model-based specifications*(as Z [21], VDM [15], or Larch interfaces languages [11]). Specification of class methods uses two predicates (pre/post-conditions) that describe the relationship between the input and the output by means of logic formulas.

In this paper we present the reflective features of the language which can be used for some interesting applications. An object-oriented reflective system

[1] email: aherranz@fi.upm.es
[2] email: jjmoreno@fi.upm.es
[3] email: noelia@lml.ls.fi.upm.es

is one that is itself built out of programmable, first-class objects. In Slam-sl it is possible to dynamically inspect and manipulate classes, objects, methods, pre and postconditions, and other semantic language elements.

Reflection capabilities can be used in many useful applications: formalization of design patterns in term of class operators. component based software, compiler construction, formalization and manipulation of modelling languages (i.e. UML, [7]), debugging, module operations, etc.

The reflective features of a language can be categorized as *linguistic*, *structural*, and *behavioral*. Linguistic and structural reflection is basically syntactical and is present in a number of programming languages either in the object-oriented imperative community (Smalltalk [8], Java, C# [3]) and in the declarative programming community (Lisp, Prolog, Maude [4]). However, behavioral reflection allows managing the semantics of the elements of the language (Maude).

We call *declarative reflection* those behavioral reflection characteristics that rely on declarative models (logics, functions, etc.) Declarative reflection not only allows manipulating the semantics of classes and objects, but also to formally reason about them. In the previous list, only the specification language Maude, based on rewriting logic what in turns is a reflective logic, can be considered to have declarative reflective features. However, they are focused mainly on module operations. Slam-sl is equipped with behavioral/declarative reflection: (1) It is possible to inspect and manipulate object oriented elements: classes, methods, etc. (2) It is possible to manage the declarative behaviour of such elements, as class invariants, or methods pre and postconditions. (3) Declarative properties of those elements are included and formal reasoning is enforced. For instance, inheritance is only permitted under certain semantical conditions and these conditions are preserved even in dynamically generated classes. (4) It is possible to declaratively model object oriented features, as interfaces or abstract classes.

We are going to present in detail a pair of interesting application: the formalization of design patterns as class operators and, consequently, the use of a formal method as a pattern language.

The paper is organized as follows: an introduction to Slam-sl in section 2, reflective properties of Slam-sl in section 3, and composite and decorator pattern specifications in section 4.1. We present some conclusions in section 5.

2 Object oriented specifications. Slam-sl

This section presents the main constructions of the language focused on reflective features that will used in the following sections.

Slam-sl is part of the SLAM project [13,14,12], a software construction development environment that is able to synthesize (reasonable) efficient and readable code in different high level object oriented target languages like C++

or Java. Among other features, the user can write specifications in a friendly way, track her hand-coded optimizations, or check in debug mode those optimizations through automatically synthesized assertions. To our knowledge this novel feature is not present in any existing system.

In order to facilitate the understanding of Slam-sl we will show its elements with a concrete syntax that does not necessarily correspond neither with an internal representation nor the environment presentation [4], so the reader should not pay attention to the concrete syntax but to the abstract one.

A Slam-sl program is a collection of specifications that defines classes and class properties. The specification of method behaviour is given through preconditions and postconditions but with a functional flavour as we will see.

2.1 Slam-sl toolkit

As many others specification languages Slam-sl has a powerful toolkit with predefined types representing booleans, numbers, characters and strings, records and tuples, collections (sequences, sets, etc.), dictionaries (maps, relations, etc.). Slam-sl type syntax reflects value syntax, for instance, the type 'sequence of integers' is written as `[Integer]` and its values are written as `[1,2]`, a tuple type can be written as `(Char, Integer)` and its values as `('a',32)`, etc.

Records and tuples. The *type* `(x : Float, y : Float)` defines the Cartesian product $Float \times Float$ but, in order to avoid 'boilerplate' in specifications, field selectors are defined. Let `p` be `(x -> 0.1, y -> 0.0)`, `p` represents the tuple $(0.1, 0)$ – another allowed equivalent syntax – and `p.x` is the projection of the first component of the tuple. Some syntactic sugar for modifying a record have been added to Slam-sl through the *except* operator (\). For instance, the formula `q = p \ y -> 1.0` states that `q` represents `(x -> p.x, y -> 1.0)`.

Sets and sequences. The set type expression `{String}` groups together values like `{"Hello","world"}` or `{}`. Usual mathematical operations over sets are predefined (union +, intersection $*$, etc.).

Sequences like `["Hello","world"]` or `[]` belongs to the type `[String]`. Sequences are indexed data collections. If `s` is a sequence, then `s(i)` with is the i-th element, 1 is the first index, of the sequence (if exists), and
`t = s \ 2 -> "Earth"` establishes that `t` represents `["Hello","Earth"]`. Usual operations over sequences are predefined (append +, insert, dom, rng, **in**, etc).

Both sets and sequences inherit the properties of Collection. Collection is a class over which quantifiers are allowed.

Collections and 'quantifiers'. Set and list comprehension, *restricted* quantifiers, and *iteration* follow a common abstract scheme of 'traversing' collections and Slam-sl introduces the following expressive syntax:

[4] In fact, Slam-sl programs are stored in XML format and its presentation in the environment can be customized.

Q x **in** d [**where** $F(x)$] **with** $E(x)$

The above Slam-sl expression is a quantified [5] expression. Q is the quantifier symbol that indicates the meaning of the quantification by a binary operation and a starting value. d is an object of the class Collection. x is the variable the quantifier ranges over. F is an optional boolean expression that filters elements in the collection. Finally E represents the function previously applied to elements in the collection.

Some predefined quantifiers appears in the following table with an informal description:

Symbol	Generalizes
exists	$\vee$ with false
exists1	as **exists** but limiting the count to 1
forall	$\wedge$ with true
sum	$+$ with 0
prod	$\times$ with 1
count	*inc* with 0 (counting!)
select	searching
max	*max*
filter	filters
map	apply a function to every element in a collection

2.2 Classes and class properties

In Slam-sl, a class is defined by specifying its properties: name, relationships with other classes, and method specifications. We will specify the class Stack for representing stacks of objects:

class Stack **inherits** Collection
state Empty
state NomEmpty (top : Object, rest : Stack)

The first line declares a new class called *Stack* and establishes that class Stack inherits properties from Collection. Lines starting with **state** define attributes that are the internal representation of the class instances. Slam-sl permits defining *algebraic types* to indicate that a syntactical construction represents class instances, in our example, the values Empty and NonEmpty (5, Empty)

[5] We have maintained the term 'quantifier' because it is a generalization of quantification in logic

represent the state of an empty stack and the state of a stack with a unique object (the constant 5) repectively.

Class relationships. Slam-sl can be considerated as a programming language. As in any other (OO) programming languages, we cannot distinguish every kind of relationships between classes as UML allows. For instance, *aggregation* cannot be distinguished from *composition*, and some *associations* are implicit through the semantics of methods. Anyway, we can list the following relationships that can be caught staticaly:

Aggregation: the state specification of a class defines an *aggregation* or *composition* between class instances and instances of other classes.

Inheritance: class properties can be defined from scratch or by inheriting them from already defined classes. Overriding of such properties are constrained in Slam-sl, not only the signatures but also the meaning (see below).

Polymorphism: generic polymorphism is introduced by permitting introducing arguments in types. Slam-sl allows deferring classes in the style of Eiffel but adding some features from theories (in OBJ terminology [10]) as well as type classes (à la Haskell [16]) playing a more powerful role than C++ templates.

Method specifications. The standard methods in stack objects permit creating an empty stack, decide if a stack is empty, read the top of the stack, and push and pop elements. Slam-sl helps the user to *classify* different kinds of methods: *constructors*, *modifiers* and *observers*. Let us complete the stack class specification with the definition of its methods:

constructor empty
pre true
empty
post result = Empty

observer isEmpty : Bool
pre true
isEmpty
post result = (self = Empty)

observer top : Object
pre not self .isEmpty
top
post result = self .top

modifier push(Object)
pre true
push(x)
post result = NonEmpty (self, x)

modifier pop
pre not self .isEmpty
pop
post result = self . rest

In Slam-sl an operation is specified by a set of *rules*, every rule involves a *guard* or *precondition* that indicates if the rule can be triggered, an *operation call scheme*, and a *postcondition* that relates input state and output state. The general form of a rule is the following:

pre $P(x$,self)

op (x)
post $Q(x$,self,result$)$

where $P(x, \mathsf{self})$ is a Slam-sl formula involving variables in the argument (x) and the recipient of the message (self) in case of the operation to be either an observer or a modifier. $Q(x, \mathsf{self}, \mathsf{result})$ is another formula involving variables in the argument, the reserved symbol result that represents the computed value of the function and self that represents the state of the receipt of the message before the method invocation.

Some 'shorthands' help the user to write formulas concisely and readablely: self can be ommited for record fields, as in VDM, explicit function definitions are allowed and unconditionally true preconditions can be skipped.

Let us explain in detail how Slam-sl handles method overriding. Suppose you have a class C with a method m with precondition P and postcondition Q. Now, a subclass C' of C is declared supplying a new specification for m: precondition P' and postcondition Q'. As Slam-sl is a formal specification language, it forces that the following statement holds:

$$\textbf{Inheritance Property} : (P \rightarrow P') \wedge (P \wedge Q' \rightarrow Q)$$

Encapsulation. Encapsulation is an important distinctive in programming languages which permits the user to control coupling and maximize cohesion.In general, encapsulation is not always managed in formal methods. In Slam-sl, as in other object oriented programming languages, the user can indicate the visibility scope of each method: **public**, **protected** or **private**. If an attribute is indicated as public the user get for free an observer, for instance, in the stack example the definition of the observer top could have been avoided in this way:

state Empty
state NomEmpty (**public** top : Object, rest : Stack)

The language introduce a broad notion of inheritance via aggregation. Let us see an example, the following Slam-sl spec defines a *read only wrapper* for stacks:

class ROStack
state (target : Stack **accept** top, isEmpty)

constructor wrap (Stack)
wrap (s) = (target −> s)

Now, the user can return a wrapper instead of the stack if she does not want stack instances to be modifier by clients. This is a pretty unexplored feature.

Applying the 'shorthands' introduced through the section, we could redefine the stack example in a more concise way:

```
class Stack(T) inherits Collection
state Empty
state NomEmpty (
  public top : T,
         rest : Stack)

constructor empty
empty = Empty

observer isEmpty
isEmpty = (self = Empty)

modifier push(T)
push(x) = NonEmpty (rest -> self,
                    top -> x)

modifier pop
pre not self.isEmpty
pop = self.rest
```

Abstract classes. In Slam-sl it is quite easy to declare interfaces, i.e. classes with no state and methods that must be redefined in the subclasses. The way to declare such methods is to indicate that the precondition is false. This means that this method is not applicable in any case. Notice that it is still possible to supply an adequate postcondition. This postcondition must be preserved in all derived classes. Those methods that have no definition are implicitly considered to have precondition false and postcondition true. For the practical use of Slam-sl as an specification language a dedicated syntax for interfaces can be introduced. We just want to stress the point that they can be specified in a declarative way.

2.3 Semantics

There is a difference between the semantics one need to supply for an specification language and that for a programming language. The latter is designed for an *expert*: i.e. programmers or automatic tools for program manipulation. However, an specification language need to be equipped with a very intuitive semantics because its specifications need to be read by non-experts, i.e. customers. In this sense, the intuitive semantics for Slam-sl is (ordered sorted first order) logic and every function f defined by:

function f (...)
pre $P(x$,self)
f (x)
post $Q(x$,self,result)

can be undertood by this *simple* formula scheme:

$$\forall x, s.\ (P(x, s) \rightarrow Q(x, s, f(x, s))$$

The same idea underlines the W logic for the specification language Z [17].

One of the additional advantages of Slam-sl declarative reflection is that all the elements of the language are specified (see next section) by logic formulae. Therefore every Slam-sl component can be understood in an intuitive way.

On the other hand more elaborated semantics can be developed in order to support (automatic) Slam-sl specifications manipulation.

3 Reflective features

In this section we will present some Slam-sl reflective constructions that we will use in the following.

Informally, a reflective language is a language in which interesting aspects of its model can be represented and manipulated in the language itself. Reflection makes possible advanced meta-programming applications, like reification of Slam-sl or other languages, and development of interpreters and component based systems.

As in other reflective languages, reflection features and "standard" features can be combined in any consistent way. Although our examples will not use this combination it is clear that many other examples (like component specification as is shown in the applications of reflective capabilities in C#) can take advantage of it.

3.1 Classes and class relationships

Like many others object oriented languages, Slam-sl classes are represented as instances of a (meta)class called *Class*. The declaration of a class *introduces* an immutable instance of the class Class. Let us start with the definition of Class:

```
class Class
public state (name : String , inheritance : { Class },
              stat : State , inv : Formula, methods : {Method})
invariant forall m1 in methods, m2 in methods | m1 /= m2 with m1.differ(m2)
```

We have made a natural reading of 'what a class is': a name, inheritance relationships, aggegation relationships, and methods. The class name is a string, inheritance is a set of instances of Class, aggregation is represented by an instance of State plus an instance of Formula representing the invariant, and, finally, we have added a set of instances of Method.

The invariant in Class establishes that two methods of the class must differ in the signature[6]. In other words, method overloading is allowed, but there must be an argument of different type. Notice that thanks to this declarative specification Slam-sl is able to identify those properties that a class must fulfill what is much more powerful than the reflective features of Java or C# that are merely syntactic.

Let us see the definition of State plus auxiliar classes:

[6] In fact, the invariant should include some other needed properties related to the inheritance rules stated in section 2.2 but me omit them for the sake of simplicity.

```
class Declaration
state (public name : String,
       type : Class)

invariant name. isIdentifier

public constructor makeDec
  (String, Class)

observer isSubtype
  (Declaration ) : Boolean
isSubtype(d) =
  self .type.isSubtype(d.type)
```

```
class DecCollection
state ( decls : { Declaration})

constructor makeEmptyDec

public modifier add (Declaration)

----
class State inherits DecCollection
```

Among the interesting operations of classes, let us show a couple of them. First of all we will introduce a constructor to create a class. Slam-sl automatically generates the field observers[7]. When a class is just an interface is detected by checking if the state is empty and if all the precondition of the methods are false.

```
public constructor makeClass
  (String, { Class }, State, Formula, {Methods})
makeClass (name, inh, st, inv, methods) =
  (name, inh, st, inv, methods)

public observer isInterface : Bool
 isInterface = st.isEmpty and forall m in methods
                                with m.doNothing
```

3.2 Methods

The class modelling methods could be specified in the following way:

```
class Method
Method state (kind : MethodKind, visibility : Visibility ,
              name : String, sig : ArgSig, return : Class,
              prec : Formula, postc : Formula)

public constructor makeMethod
  (MethodKind, Visibility , String, ArgSig, Class,
   Formula, Formula)
makeMethod (k, v, n, sg, rt, pr, ps) = (k, v, n, sg, rt, pr, ps)

public observer typeSig : [ Class]
```

[7] only if the user states attributes are 'public', but we assume it along the rest of the paper.

```
typeSig = map d in sig with d.type

public observer invokation : [ String ]
invokation = map d in sig with d.name
```

with the following previous definitions:

```
class ArgSig inherits DecCollection
```

We have introduced a couple of useful operations: constructing a method, abstracting the type signature just using the argument types (the names are almost irrelevant except for the pre and postconditions), and composing a method call with the argument names.

On top of them, we can describe a number of interesting operations on methods. The first one (isCompatible) indicates when two methods are equivalent (same name, types and equivalent pre and postconditions). The second one (canInherit) specifies when a method can override another definition. They must have a coherent definition (same name and arguments/return type) and the inheritance property must hold.

```
public observer isCompatible (Method) : Bool
isCompatible (m) =
  kind = m.kind and name = m.name and
  typesig = m.typesig and return = m.return and
  (prec implies m.prec[m.invokation/invokation]) and
  postc (implies m.postc[m.invokation/invokation])

public observer canInherit (Method) : Bool
canInherit (m) =
  kind = m.kind and name = m.name and
  sig.length = m.sig.length and
  (forall i in sig.dom
       with sig(i).isSubtype(m.sig (i))) and
  m.return.isSubtype(m) and
  (m.prec implies prec[invokation/m.invokation]) and
  (postc implies m.postc[m.invokation/invokation])
```

Finally, we specify operations to decide when two methods are really different (up to argument names) and when a method implements an interface method (i.e. precondition false):

```
public observer differ (Method) : Bool
differ (m) =
  name /= m.name or
    (name = m.name and
     (sig.length /= m.sig.length or
      (exists i in sig.dom
           with sig(i).type /= m.sig(i).type)))
```

```
public observer doNothing : Bool
doNothing = (prec = false and postc = true)
```

For the sake of simplicity, we assume that all record components of classes Method and Class are public. In fact, good object oriented methodologies recommend to make them private and to declare adequate methods to access them. We omit such definitions to avoid an overloaded specification.

3.3 Formulas

Possibly, the most interesting Slam-sl reflective properties are those related to *formula* management. Slam-sl runtime environment can manage formulas in the same way the compiler does, this means formulas can be created and compiled at runtime so the user can specify programs that manage classes and class behaviors. The following specification of formulas reflects its abstract syntax in Slam-sl:

```
class Formula
state Constant (Bool)
state Variable (String)
state And (Formula, Formula)
state Or (Formula, Formula)

state Implies (Formula, Formula)
state Equiv (Formula, Formula)
state Expression (Expression)

public constructor makeTrue
makeTrue = Constant (true)
public constructor makeFalse
makeFalse = Constant (false)
public constructor makeVariable (String)
makeVariable (s) = Variable (s)

public constructor makeAnd
  (Formula, Formula)
makeAnd (f1,f2) = $f1 and f2$
...

public modifier substitute (String, Expression)
substitute (var,expr) =
  result =
    case self
      Constant (c) -> self
    | Variable (v) -> if v = var then Expression (expr)
                      else self
    | And (f1,f2) -> And (f1.substitute(var,expr),
                          f2. substitute (var,expr))
    | Or (f1,f2) -> Or (f1. substitute (var,expr),
                        f2. substitute (var,expr))
    | Implies (f1,f2) -> Implies (f1. substitute (var,expr),
                                  f2. substitute (var,expr))
    | Equiv (f1,f2) -> Equiv (f1. substitute (var,expr),
                              f2. substitute (var,expr))
    | Expression (e) -> e[x/expr]
```

Writing formulas with the above interface would produce unreadable specifications so we write instances of Formula using the Slam-sl own notation between

$ ... $ symbols and permitting the compiler to parse the sentence and generate the expression. See the definition of the constructor makeAnd.

Syntactic sugar for the substitution operation have been introduced: f[x/e] is the formula f replacing all the references to the variable x by the expression e. Notice that we have used that syntax for expressions in the last line of the definition of substitution operation.

4 Application

4.1 Design Patterns as Class Operators

As an application of the reflective features of Slam-sl let us show how design patterns [9] can be formalized as class operators. A given (preliminary) design is the input of a design pattern. This design is modeled as a collection of classes. The result of the operation is another design obtained by modifying the old classes and/or creating new ones, taking into account the description of the design pattern.

For instance, consider you have a collection of classes *leafs* (e.g. Line, Circle, Rectangle, ...) that share some operations (e.g. draw, rotate, resize, ...) and you want to compose all of them in a wider object that either has all of them as particular cases and also can collect some of them inside (e.g. a Figure). The *Composite* pattern considered as an operator accepts classes (*leafs*) as input and returns two new classes Component (merely an interface) and Composite (for the collection of components) with the common operations as methods, and modifying classes in *leafs* to inherit from Component.

More specifically, design patterns are modeled as a class with a single function apply that is a class operator. This precondition for this function collects the logical conditions required to use the pattern with success. Basically, this means that the pattern precondition establishes the *applicability* of the pattern, talking in terms of the sections in the pattern description. For instance, in the *Composite* pattern we mentioned above, the precondition needs to ensure that all the classes in *leafs* define the common methods with the same signature.

On the other hand the postcondition encompasses most of the elements of the *intent* and *consequences* sections of the pattern description. In the *Composite* pattern, the postcondition establishes that input classes *leafs* now inherit from Component and classes Composite and Component are introduced, the first one inheriting from the second one. The Composite state is a collection of Components and its methods are described by iterative calls to the corresponding *leafs* methods.

In order to describe all this elements, the reflective features play a significant role because they allow inspecting argument classes and to describe new classes as result. Design patterns can be described by a (polymorphic) class DPattern. The method apply describes the full behaviour of the pattern

by accepting a collection of classes as arguments (the previous design) and returning a new collection of classes. The class argument (coming from the polumorphic definition) is occasionally needed to instruct the pattern about the selection of classes, methods, etc. that take part in the pattern. This argument is stored in the pattern by a dedicated constructor.

```
class DPattern (T)
state (arg: T)

public constructor instantiate (T)
 instantiate (x)
post result.arg = x

public function apply ([Class]): [Class]
```

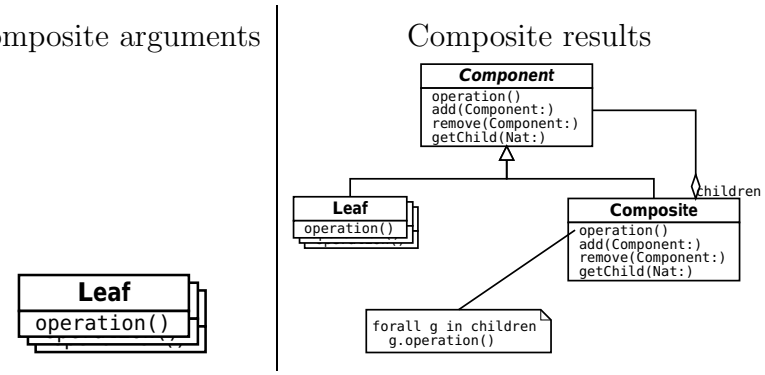
Inheritance is used to derive concrete design patterns. It is also needed to instantiate the type argument and supplying a value for the state. Notice that design pattern variants are easily supported in our model.

Let us describe the method by some examples taken from [9]. We have chosen one pattern for each component of the classification: creational, structural, and behavioral patterns. A graphical description complements the formal definition using an OMT-based notation taken again from [9]. A deeper discusion as well as more examples can be found in [19].

4.2 Composite pattern

The *Composite* pattern is part of the object structural patterns. It is used to compose objects intro tree structures to represent part-whole hierarchies. Using the pattern the clients treat individual objects and compositions of object uniformly.

When we treat it as a class operator, we have the collection of basic objects as argument (called the *leafs*). The result "invents´´ two new classes Component and Composite. Component is just an interface for all the common methods in all the leaf classes plus some methods to add, remove and consult internal objects. Composite inherits from Component and stores collection of components. The result also collects all the classes in *leafs* that are modified by inheriting from Component. The methods in Composite can be grouped in two parts. On one hand, we have methods to add and remove a component, and also to consult the ith element in the component collection (getChild). On the other hand, we have all the common methods of the *leafs* that have a very simple specification by iterative calling the same operation in all the components. See figure 1 for the complete Slam-sl specification.



class Composite **inherits** DPattern ()

public function apply ([Class]): [Class]
let m **in** commonMethods **equiv** (**forall** cl **in** leafs **with** m **in** cl.methods)
pre (**not** leaf.isEmpty) **and** (**not** commonMethods.isEmpty)
apply (leafs)
post result = [component, composite] +
map c **in** leafs **with** c \ inh. insert (component)
where
component =
makeClass ("Component", {}, makeEmptyDec, $true$,
map m **in** (commonMethods + [create, add, remove, getChild])
with m \ prec = $false$ **and** postc = $true$)
composite =
makeClass ("Composite", {}, [makeDec (children, [component])],
$true$, [create , add, remove, getChild]
+ **map** m **in** commonMethods gen (m))
create =
makeMethod($constructor$, $public$, "create", makeEmptyDec,
$true$, $(result = []) $)
add =
makeMethod($modifier$, $public$, "add", [makeDec("c", component)],
$true$, $(result = children. insert (c))$)
remove =
makeMethod($modifier$, $public$, "remove", [makeDec("c", component)],
$true$, $(result = children.remove (c))$)
getChild =
makeMethod($observer$, $public$, "getChild", [makeDec("i", Nat)],
$true$, $(result = children[i]))$
gen (m) = m \ prec = $(forall c in children with m.prec [this/c])$ **and**
postc = $(result = map c in children
with makeCall (m.name,
[c] + m.invokation))$

Fig. 1. Composite pattern specification.

4.3 Decorator pattern

The *Decorator* pattern is classified as object structural and it is used to attach additional responsibility to an object dynamically. It can be seen as the following class operator: A collection of concrete components and a collection of decorators are used as arguments. They share some operations that the pattern abstracts in two steps. First of all, a new Decorator class abstracts the operation of the decorators. Then another newly created class Component abstracts the operation either for the concrete components and for the decorator.

The class argument is used to split the sequence of classes into the concrete components and the decorators. Concrete components are forced to inherit from Component, while decorators inherit from Decorator and modifies the common methods to add a call to the decorator operation. The Decorator class contains a Component in the state and offers the common methods as public. They are implemented as simple calls to the equivalent operations in the stored component. Finally, Component is merely an interface for the common methods.

Due to lack of space this pattern specification have been removed from the final version of the paper.

4.4 Design Pattern Composition

Viewing design patterns as operator over classes allows us to create *new* design patterns by *operator composition*. For instance, the composite design pattern can be applied to a collection of *leafs* and then a decorator can be applied to the new design.

In the case study presented in chapter two in [9], the design of a document editor is guided by the application of several design patterns. Some of those design patterns are applied to (a part of) the result of a previous one. Because design patterns have been modelled as class operators, we can specify the composition of them:

```
composite = instance (Empty);
glyph = composite.apply ([border, scroll, character,
                          rectangle, polygon]);
decorator = instance (3);
mono_glyph = decorator.apply(glyph. prefix (3))
```

4.5 Slam-sl as a pattern language

The formalization of design patterns in terms on class operators and using the (declarative) reflective features of an specification language have a number of advantages:
(1) Coherent specifications of patterns are essential to improve their comprehension and to reason about their properties.

(2) It is possible to develop tools for supporting design patterns. In fact we are interested in introducing them in existing development environments (as Visual Studio, Visual Age, etc.). The tool can allow applying a design pattern to the project you are working on. The project should be modified according to our description adopting the design pattern. In this way, we can apply design patterns to already existing code and with "every day´´ existing CASE environments.
(3) Patterns can be combined by simply applying function composition.
(4) The functional semantics of Slam-sl can be modified to support functional-logic semantics. Functional-logic languages amalgamate the main features of functional languages and Prolog-like languages, in such a way that inverse functions can be computed. This means that, in principle, we can identify a concrete design patterns into an existing design/specification.

In the literature we can find some other formalizations of design patterns. The work in [1] is focused on the formalization of architectural design patterns based on an object oriented model integrated with a process oriented method to describe the patterns. [18] presents a way to formalize temporal behaviors of design patterns, so communication between objects is the main goal of the specification that uses primitives of a process algebra. Although both use an specification language for the formalization they do not propose any supporting tool and reflection is not used. The project proposed in [5,6] are more focused on providing tools that interact with existing code. They use a metaprogramming language based on a (limited form of) verbal specification. A tool can read it and produce what they call a *trick*, basically an algorithm to manipulate programs. They have also designed a visual language for specifying patterns (LePus). They share some of our goals and even more, but we claim that we can get a similar power with a simpler approach.

In fact, thanks to its declarative reflection features, Slam-sl can be considered as a pattern language. Once you can model a patterns as a class operator, Slam-sl can be used to specify it and this specification can be used to instruct the associated tool to apply the pattern to existing designs and programs.

4.6 Component-based software specification

One of the most promising new topics in software construction is component based software in which programs are composed of several components, provided either by the development team, the development environment, or by third part providers. A component is a software compositional unit having a collection of interfaces and fulfilling requirements. A component can be developed and integrated with other components independently of time and space, to produce a new software application.

One of the important problems when dealing with software components is how to "ask" a component about the services it offers. Current systems just offer the possibility to consult the name and the signature of such these

services. The "semantical" behaviour of a component can be identified by the programmer (but not by the program itself) using the documentation provided by the component designer. This documentation is usually very informal and cannot be consulted automatically by another component.

For these reasons, Slam-sl can be used to specify software components. On one hand we have a formal definition of the different services of the components if we specify them as Slam-sl classes. On the other hand, this information can be consulted by another Slam-sl component by using the reflective capabilities. Remember that given a class we can inspect its methods, the signature of them as well as the pre and postcondition. Following our small example, imagine we have a component to provide stack operations with our Stack class. Then we can write another component that can consult the methods of Stack by the expression Stack.methods, consult the signature of one of these methods by, for instance, push.sig, or even check the postcondition of one of them as done in top.postc. Notice that it need to be done by introducing into the "standard" Slam-sl specification reflective features combining the language and the metalanguage in a transparent way.

However, there are some other aspects like the communication protocols between components that are also needed for component specification. The extension of Slam-sl with concurrency is a matter of future work and should be used for the complete specification of software components.

5 Conclusion

We have also presented the declarative reflection characteristics of SLAM. The main advantage with respect to other reflective languages is that the semantics of a class method can be inspected (by consulting pre and postconditions) what is very useful for a number of applications like specifying design properties, and the definition of grey box frameworks [2]. Our conclusion is that declarative reflection is a key feature for: (1) A simple formalization of design patterns in terms of class operators. (2) Supporting concrete tools that permit applying design patterns to existing code. (3) Specification and development of component-based software. (4) Formalization, modelling and manipulation of UML in the vein of [7] where the language Maude and its reflective capabilities are used to model UML, check properties of UML designs, and supporting UML extensions. Slam-sl can be used for the same purpose. (5) Providing logical semantics for every Slam-sl element.

The precise definition of software design patterns is a prerequisite for allowing tool support in their implementation. Thus comprenhensive specifications of patterns are essential not only to improve their understanding and property reasoning, but also for supporting an automatization of their use. Our proposal for fomalizing design patterns is to model them as class operators. We are not saying that design patterns *are* class operators. Our thesis is that most of them *can be seen* as class operators and this view offers interesting

advantages in terms of comprehension and automatization of use. Of course, our approach is not neccesarily "better" than others. In fact, different formalizations focused on a particular aspects yields to different tools. We only encourage the fact that it is very simple and easy to automatize in existing development tools. On the contrary, it is not already clear that all the design patterns can be modeled as class operators (for instance the *Factory Method* that can only be seen as a class operator in a very tricky way).

As a future work we plan to incorporate our ideas to a concrete tool for introducing design patterns in the software development process. We are also interesting in exploring further the use of SLAM for component specification and development.

References

[1] P. Alencar, D. Cowan, and C. Lucena. A Formal Approach to Architectural Design Patterns. In M. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, LNCS, pages 576–594. Springer Verlag, 1996.

[2] M. Buechi and W. Weck. The Greybox Approach: When Blackbox Specification Hide too much. Technical Report TUCS-TR-297a, Turku University, Findland, 1999.

[3] C# language specification. Draft standard ECMA/TC39/TG2/2000/3.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *A Maude Tutorial*. CSL, SRI International, March 2000.

[5] A. Eden, A. Yehudai, and J. Gil. Precise Specification and Automatic Application of Design Pattern. In *Proc. 12th Annual Conference on Automated Software Engineering*, 1997.

[6] A. Eden, A. Yehudai, and J. Gil. LePUs - a Declarative Pattern Specification Language. Technical Report 326/98, Department of Computer Science, Tel Aviv University, Israel, 1998.

[7] J. Fernández and A. Toval. Can intuition become rigorous? foundations for uml model verification tools. In F. M.Titsworth, editor, *International Symposium on Software Reliability Engineering*, pages 344–355, San Jose, California, USA, October 8-11 2000. IEEE Press.

[8] B. Foote and R. Johnson. Reflective Facilities in Smalltalk-80. In *Proc. OOPSLA '89*, pages 327–335, 1999.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[10] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical report, Oxford + SRI, October 1993.

[11] J. V. Guttag and J. J. Horning. A tutorial on Larch and LCL, a Larch/C interface language. In S. Prehn and W. J. Toetenel, editors, *VDM91: Formal Software Development Methods*. Springer-Verlag Lecture Notes in Computer Science 551, October 1991.

[12] A. Herranz and J. Maya, N. Moreno-Navarro. Specifying in the large: Object Oriented Specifications in the Software Development Process. In *Proceedings Sixth International Conference on Integrated Design and Process Technology (IDPT) - to appear*, 2002.

[13] A. Herranz and J. Moreno-Navarro. Towards Automating the Iterative Rapid Prototyping Process with the SLAM System. In *Proceedings VI Spanish Conference on Software Engineering*, pages 217–228, 2000.

[14] A. Herranz and J. Moreno-Navarro. On the Design of an Object-oriented Formal Notation. In *Proceedings Fourth Workshop on Rigorous Object-Oriented Methods ROOM 4 - to appear*, 2002.

[15] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.

[16] S. P. Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98. A non-strict Purely Functional Language*. February 1999.

[17] A. Martin. *Machine-Assisted Theorem-Proving for Software Engineering*. PhD thesis, Wolfson Building, Parks Road, Oxford, UK, 1995.

[18] T. Mikkonen. Formalizing Design Patterns. In *Proc. ICSE'98*, pages 115–124. IEEE Computer Society Press, 1998.

[19] J. Moreno Navarro and A. Herranz Nieva. Design Patterns as Class Operators. submitted to OOPSLA, 2001.

[20] The SLAM site. `http://lml.ls.fi.upm.es/slam`.

[21] J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.