A Design Rule Language for Aspect-Oriented Programming

Alberto Costa Neto[1], Arthur Marques[2], Rohit Gheyi[2], Paulo Borba[1], Fernando Castor Filho[1]

[1] Informatics Center – Federal University of Pernambuco
PO Box 7851, 50740-540 – Recife – PE – Brazil

[2]Department of Computing Systems – Federal University of Campina Grande
PO Box 10.106, 58.109-970 – Campina Grande – PB – Brazil

{acn,phmb,castor}@cin.ufpe.br

{arthursm,rohit}@dsc.ufcg.edu.br

Abstract. *Aspect-Oriented Programming is known as a technique for modularizing crosscutting concerns. However, constructs aimed to support crosscutting modularity might actually break class modularity. This can be mitigated by using adequate Design Rules between classes and aspects. We present a language that supports most of the Design Rules found in AO Systems, making easy to express and verify them automatically. We discuss how our language improves crosscutting modularity without breaking class modularity. Also, we give some details about the language semantics expressed in Alloy.*

1. Introduction

Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] has been proposed as a technique for modularizing crosscutting concerns. Logging, distribution, tracing, security, and transactional management are accepted as examples of crosscutting concerns well addressed by AOP. Yet, AOP seems to produce modularity problems. In this context, several authors argue that in order to reason about classes it is necessary to consider all aspect implementations [Sullivan et al. 2005, Clifton and Leavens 2002, Steimann 2006].

In the presence of aspects, class modularity is compromised because, when evolving a class, it might be necessary to analyze the implementation of existing aspects, which can refer to classes implementation, instead of analyzing only the class and the interface of other referred classes. In fact, by referring to classes implementation details in aspects, one can inhibit modular reasoning and compromise the changeability, requiring class modifications to be fully aware of the aspects affecting the class. Therefore, constructs aimed to support **crosscutting modularity** might actually break **class modularity** [Ribeiro et al. 2007].

This weakness can be mitigated by using adequate Design Rules (Section 2) between classes and aspects, as discussed in other works [Sullivan et al. 2005, Lopes and Bajracharya 2006]. Design rules are necessary to reduce such new dependencies in Aspect-Oriented (AO) systems. They are not just guidelines and recommendations: they generalize the notion of information hiding interfaces and must be rigorously obeyed. Besides being useful for verification purposes, they serve as a guideline for developers since the initial phases of the development process. None of the previous approaches [Sullivan et al. 2005, Lopes and Bajracharya 2006] propose a language with the specific purpose of describing design rules.

In this paper we present a Language for Specifying Design Rules (LSD) that improves the modularity of AO systems. This is achieved by specifying the essential structure and behavior that each developed component must provide aiming to establish the minimum requirements necessary to work in parallel. LSD supports the description of Design Rules in a declarative manner, making easier the development of automatic verification mechanisms of these rules in the written code. We discuss some evolution scenarios (Section 5) aiming to show how the existence of design rules can help to identify problems and also help to solve them.

The main contributions of this paper are:

- Present a language for specifying design rules, improving the modularity of AO systems, in a declarative manner using a syntax similar to Java and AspectJ and with a defined semantics (Section 3).
- Describes the language semantics in Alloy [Jackson 2006] which, besides being useful for that purpose, is also helping to refine the language semantics through its automatic verification (Section 4).

2. Modularity Issues in AOP

We discuss in this section the modularity concept adopted in this work, how AOP contributes to improve (or not) it and how Design Rules can be useful to improve modularity.

Parnas [Parnas 1972] proposes a criteria to consider a design modular, and the **information hiding** principle, which is used as the criteria to decompose a system into a set of modules, hiding the parts of a system that are more likely to changes into modules with stable interfaces. The following quality attributes are expected in a modular design:

Comprehensibility: A modular design allows developers to understand a module looking only at: (1) the implementation of the module itself; and (2) the interfaces of the other modules referenced by it.

Changeability: A modular design enables local changes. If the hidden implementation of a module **A** changes, other modules that depend exclusively on **A's interface** will not need to change, since there is no modification in the module interface.

Parallel development: After specifying module interfaces, a modular design enables the parallel development of modules. Different teams might focus on developing different modules, reducing the time-to-market and the need of communication.

In AO software, crosscutting concerns are implemented with aspects that intercept join points in classes and change their behavior. One type of problem that denotes lack of modularity is called **unanticipated change in base code**. It occurs for example when a extract method refactoring is applied, removing join points (method call and execution, for example) and creating others like the call and execution of the extracted method. Any aspect dependent on that join points will not work adequately, unless the pointcuts are changed to match the new join points. This problem occurs because the aspect is dependent on something that can be changed, thus the design is not modular.

Another problem that frequently exists in AO software is the **lack of support to parallel development**. This occurs because many aspects require that some details about classes are defined and codified before aspects programming. For example, it is necessary to know which methods change an object state in order to write an aspect that intercepts

these methods and implements persistence. Thus, developers must agree on some rule to name these methods so that aspects can be developed in parallel.

As mentioned in Section 1, these problems can be mitigated by establishing design rules. Intuitively, many AO developers accord these rules but they do not document them. Others write them as comments within programs or design models. A small number of developers try to enforce these rules automatically (writing programs to check them). Consequently, the design rules are frequently forgotten by programmers and must be discovered every time a developer needs to change the software.

It is important to notice that design rules are not so necessary if aspects are developed after classes and these classes never change. Since software is in constant change, we do not consider this as a feasible approach, preferring the design rule based approach.

Our approach, discussed in the next section, address many of modularity problems through the use of a language to express and enforce design rules to both classes, interfaces and aspects. Section 4 presents language semantics.

3. A Language for Specifying AO Design Rules

In this section, we present a **Language for Specifying Design Rules (LSD)**. Some LSD concepts were sketched in a previous work [Dósea et al. 2007]. The main objective of LSD is to decouple classes and aspects, improving modularity and maximizing independent development opportunities. Through the definition of Design Rules we argue that both class and aspect developers can work independently if a minimum set of constraints is defined and respected. LSD was defined as a mechanism for expressing and checking design rules during all development phases, specially during software design.

3.1. LSD Overview

Many Object-Oriented (OO) programming languages provide the concept of interface which specifies a public set of methods and constants expected as being provided by any class that implements it. This interface notion supports the creation of separated and narrow interfaces to different clients of the same class, limiting the coupling between them.

Our concept of interface, which we call a **Design Rule**, is wider, which means that it involves more than public methods and constants. It can include, for example required join points, private members, inter-type declarations, inheritance and implements declarations. A Design Rule contains a set of constraints that must be followed by components that declare to implement them. These constraints are automatically verified by a tool (static analyzer) that points out when some constraint is disrespected.

Aiming to explain how design rules can be created, we describe major steps that can be followed when building an AO system with design rules.

Discuss Design Rules: Developers, based on previous experience, discuss how classes and aspects will interact and agree on some design rules (Section 3.2).

Write Design Rules: The agreed design rules are written in some form that developers can understand. LSD supports this task by providing a new language construct that is called **dr** (Section 3.2). With LSD, it is possible to express them in a declarative fashion and verify them during codification.

Develop OO and AO components: Each class, interface or aspect is implemented and automatically checked against the appropriate design rules (Sections 3.3 and 3.4).

Determine the DR instance: This task involves the binding of design rules, classes, interfaces and aspects. It is necessary because classes, interfaces and aspects are not coupled and make no reference to each other. LSD determines this binding through a Design Rule Instance (Section 3.5).

In the following sections, it is presented an example that demonstrates LSD constructs and their details. The Display Update example was chosen because it is used in several AO works. Basically it is part of a simple tool for editing drawings comprising figure elements like points and lines that are depicted in a display.

3.2. Discussing and Writing the Design Rule

As a result of the first step (discussing the design rules), developers could agree on the following rules for figure elements, display and display update:

1. **FigureElement** methods called **set*** (starting with **set**, like **setX**) and **moveBy** must be **public** and return **void**. Also, all constructors must be public.
2. All **FigureElement** constructors and methods called **set*** or **moveBy** are the only possible points of state change in figure elements.
3. Methods called **set*** or **moveBy** and **constructors** must change some attribute of the figure element.
4. Methods called **set*** or **moveBy** cannot call any method called **set*** or **moveBy** from a **FigureElement**.
5. A **Display** class must have a **public void update()** method.
6. The aspect responsible for updating the display must declare a pointcut called **stateChange** that intercepts calls to the methods/constructors that change figure elements state based on their names (predetermined).
7. The aspect must also contain an advice that calls **Display.update()**. This method cannot be called from any other place in the system.

Structural Rules are design rules that describe constraints about classes and aspects members. Their format is similar to the interface description in Java, but it is possible to include additional constraints (beyond required public methods and constants), like attributes that must be declared, required private or protected methods and expected inter-type declarations. Besides, there are specific constraints about aspects structure, like requiring a specific pointcut declaration and advice declaration. Listing 1 shows a design rule called **DisplayUpdateDR** that contains three structural rules: **FigureElement** (Lines 2-13), **DisplayUpdate** (Lines 14-23) and **Display** (Lines 24-26).

Behavioral Rules provide a mechanism for specifying constraints about components behavior. Examples of behavioral rules are required method calls (call/xcall) and attribute access (get/xget) or change (set/xset), as shown in Listing 1 (Lines 9-11 and 21).

The first rule is expressed in Listing 1 within the **FigureElement** Structural Rule (Lines 3-5). Any modifier or return type that appears associated to a member (**public** and **void** in the example), must match with the classes implementing **FigureElement** structural rule. When something is not informed it is not considered.

Rules 2-3 are expressed in Listing 1 (Line 9) by the behavioral rule **xset**, which requires an assignment to some **FigureElement** attribute within change methods (specified with an **or** in Line 7) and prohibits changes elsewhere. If Rule 2 were suppressed, we could have used **set** instead of **xset**, because **set** does not prohibit assignments from other places of the program, like **xset** does. Both **set/xset** define the places where an assignment to an attribute must occur, but only **xset** restricts the assignments to the scope in which it is present. Equivalent notion is used in **get/xget** and **call/xcall** behavioral rules.

Rule 4 is expressed in Listing 1 (Lines 10-11) by negating calls to change methods within their own. It is important to notice that LSD does not impose any order between gets, sets and calls within the method. It only check if they are present in the scope.

Listing 1. Design Rule for Display Update.

```
1   dr DisplayUpdateDR [FigureElement, DisplayUpdate, Display] {
2      class FigureElement {
3         all( * set *(..) )     then public void set *(..);
4         all( * moveBy(..) )  then public void moveBy(..);
5         all( new(..) )        then public new(..);
6
7         all( * set *(..) || * moveBy(..) || new(..) ) then
8            *(..) {
9                xset(* FigureElement.*);
10               !call(* FigureElement.set *(..));
11               !call(* FigureElement.moveBy(..));
12           }
13      }
14      aspect DisplayUpdate {
15         public pointcut stateChange(FigureElement fe): target(fe) &&
16            (call(* FigureElement+.set *(..))  ||
17             call(* FigureElement+.moveBy(..)  ||
18             call(FigureElement+.new(..))));
19
20         after(): stateChange() {
21            xcall(* Display.update());
22         }
23      }
24      class Display {
25         public void update();
26      }
27  }
```

Also, in Listing 1 (Lines 24-26) the fifth rule is expressed, forcing classes that implement the **Structural Rule Display** to provide a **public void update()** method.

The remaining rules (6 and 7) are expressed, respectively, by Lines (15-18) and (20-22), within the **Structural Rule DisplayUpdate** (Lines 14-23) and basically express that an aspect implementing **DisplayUpdate**, must provide the pointcut **stateChange** and associate this to an advice that obligatorily and exclusively calls **Display.update()**.

Although design rules should be as stable as possible, we believe that it is easier to change a software with clear design rules than one with no explicit rules. In order to execute the change, the necessary changes to design rules must also be done.

3.3. Implementing Classes

Considering that the design rules are defined, class developers can concentrate on the details of classes implementation. They are free to change anything except what is established on the design rule. This is equivalent to what happens when a class implements an interface: all methods present in the interface must be implemented on the class. The

main differences are more types of constraints and also that some of them involve more than one component, like an inter-type declaration. But a common idea was preserved: clients depend only on the component interface.

Listing 2 shows a class **Point** that implements **DisplayUpdateDR** as **FigureElement**. This means that all constraints (structural and behavioral rules) associated to **FigureElement** must be respected by **Point**. Other figure elements like **Rectangle**, **Line** and **Circle** could be presented, but to keep the example short we omitted them.

Listing 2. Classes implementing the DisplayUpdateDR.

```
1   public class Point implements DisplayUpdateDR(FigureElement) {
2       protected int x, y;
3       public Point(int x, int y) {
4           this.x = x;
5           this.y = y;
6       }
7       public void setX(int x) { this.x = x; }
8       public void setY(int y) { this.y = y; }
9       public moveBy(int x, int y) {
10          this.x = x;
11          this.y = y;
12      }
13  }
14
15  public class Screen implements DisplayUpdateDR(Display) {
16      public void update() { /* Updates Screen */ }
17  }
```

DisplayUpdateDR is also implemented by **Screen** but as **Display**. This implies that **Screen** must provide an **update** method. Based on the design rule we can detect when some constraint is disrespected, avoiding to affect other components that depend on these constraints. This is important because components can be developed independently.

3.4. Implementing Aspect

The **ScreenUpdate** aspect (Listing 3) implements the design rule **DisplayUpdateDR** as **DisplayUpdate**. In conformity with the structural rule **DisplayUpdate**, **ScreenUpdate** defines a pointcut **stateChange** and an after advice that calls **Display.update()**.

Listing 3. Aspects implementing the DisplayUpdateDR.

```
1   public aspect ScreenUpdate
2       implements DisplayUpdateDR(DisplayUpdate) {
3
4       private Display display;
5       public pointcut stateChange(FigureElement fe): target(fe) &&
6               (call(* FigureElement+.set*(..))   ||
7                call(* FigureElement+.moveBy(..)  ||
8                call(FigureElement+.new(..)));
9       after(): stateChange() {
10          display.update());
11      }
12  }
```

It is important to notice that **ScreenUpdate** does not make reference to **Point** or **Screen**, what contributes to decouple aspects from them. Considering that, we can understand how it is possible to develop the aspect without classes implementations, achieving the parallel and independent development of classes and aspects discussed in Section 2.

3.5. Defining a Design Rule Instance

One important question that arises when classes and aspects are completely decoupled is where the binding between the structural rules of a design rule and their implementing elements (classes, interfaces and aspects) will occur. In LSD, this binding is performed at design rule instantiation, which requires the list of classes, interfaces, and aspects that will play the roles (structural rules) of that design rule. For example, the design rule of Listing 1 has three parameters (Line 1): **FigureElement**, **DisplayUpdate** and **Display**. Listing 4 shows the **DisplayUpdateDR** instantiation by assigning a name to the instance **DispUpd** and associating class **Point** to **FigureElement**, aspect **ScreenUpdate** to **DisplayUpdate** and class **Screen** to **Display**. It is possible to bind multiples components to a parameter, like a **Point** and **Line** to a **FigureElement**, by providing a comma-separated list of these components in the place of **Point**. The components **implements** clause is used to check if the bindings are correct, which means that each structural rule associated to the component in the design rule instance must be in its **implements** clause.

Listing 4. Design Rule instance.

```
1  dri DispUpd = new DisplayUpdateDR(FigureElement = Point;
2                                    DisplayUpdate = ScreenUpdate;
3                                    Display = Screen);
```

The chosen approach is flexible enough to support any number of instances of the same design rule with different parameters in a single system. Also, the same component can be bound to different structural rules in different systems. Finally, in an AO Software Product Line (SPL), design rule instantiation can be used as a configuration mechanism that supports any number of product instances.

3.6. Additional Language Constructs

In this section we briefly discuss other language constructs present in LSD that were not shown in other sections, but are also important to express design rules.

Inter-Type Declarations (ITD). ITDs enable aspects to introduce attributes and methods in classes. At the same time that ITD is a useful mechanism, it may create dependencies between classes and aspects. This mechanism is frequently used in AO software product lines to introduce variable method implementations or attribute values in classes. Design rules can be used to explicitly create a contract between classes and aspects introducing product line variabilities in classes.

Inheritance/Implementation Another type of constraint is to require that a class inherit from a specific class or implements a certain interface. This can be useful to write pointcuts in aspects based on a common superclass/interface. LSD provides a mechanism to enforce this constraint by declaring the superclass/interface in the class extends/implements clause within the corresponding structural rule.

Member Expressions. LSD supports expressions involving structural rule members, as requiring a method call to methods **m1** or **m2** (**or**). The class implementing this structural rule will be correct if it calls at least one of them. It is also possible to require that a method **m1** exists and other **m2** does not exist (**not**). Requiring two or more members to exist at the same time (**and**) is possible too and in fact is the default behavior for methods listed individually in a structural rule. These operators can be combined, providing more expressivity.

Quantification. Some degree of quantification is supported by LSD, enabling to select the domain to which it is applied a design rule. Listing 1 (Lines 3-5 and 7-12) shows examples of quantification over change methods. If necessary, quantification can be applied to all types of members.

Exceptions. Besides the **declare soft** construct, LSD allows the inclusion of **throws** clause in structural rules methods. The idea is forcing developers to follow constraints related to exceptions that must be included or not in method declarations.

4. Semantics

In this section, we present a translational semantics for our language. We map all constructions to a theory specified in Alloy [Jackson 2006], a formal object oriented modeling language. Alloy uses **signatures** to describe the elements presented on its model and **facts** to describe the relationship between these signatures or elements that belongs to them. We chose Alloy due to its simplicity in expressing first-order logic constraints and its tool support to perform analysis in specifications. In order to explain the translational semantics first we present our theory (Section 4.1), where we discuss the abstract syntax used in Alloy, then we give an intuition of **DisplayUpdateDR** semantics (Section 4.2) and finally we present the translational semantics (Section 4.3) where we map each design rule to its counterpart in Alloy according to this theory. We also discuss the benefits and drawbacks of our approach (Section 4.4).

4.1. Theory

We specified the abstract syntax of all elements (classes, methods, fields, aspects, advices) in our Alloy theory (specification). For example, Listing 5 presents two Alloy signatures representing a class and an aspect respectively. An Alloy signature denotes a set of objects.

Listing 5. Class and Aspect Representations

```
1   abstract sig Class extends Type {
2       vis: one VisibilityQualifier ,
3       imp: set Interface , ...
4   }
5   abstract sig Aspect {
6       attr: set Field ,
7       meth: set Method ,
8       advice: set Advice ,
9       pcut: set PointCut ,
10      decl: set InterTypeDeclaration , ...
11  }
```

A signature may introduce some relations, such as **imp**. They relate objects in one signature to another one. For instance, **imp** denotes the set of interfaces that a class may implement. The **set** Alloy keyword denotes that each object of **Class** is related to a number of objects of **Interface**. The **one** Alloy keyword denotes that each object of **Class** is related to exactly one element of **VisibilityQualifier**. Similarly, we have defined other relations in the **Class** signature specifying the attributes, constructors and whether the class is final, abstract. An Alloy signature may extend other signatures. In the previous listing, **Class** is a **Type**, which represents a type (class or interface). The previous Alloy signature **Class** is abstract. Only its subsignatures may have concrete elements. Each class is declared with a visibility qualifier (public, protected, friendly and private). We represent them by an Alloy signature (**VisibilityQualifier**).

Following the same approach the Alloy signature representing an aspect is shown in Listing 5. An aspect may declare a set of attributes, methods, advices, pointcuts and inter-type declarations. Similarly, we defined an Alloy signature for each element of our language, such as attributes, constructors, interfaces, methods, advices.

4.2. Example

In this section, we specify the display update example using our theory. For each element presented in Listing 1, we create a singleton signature in Alloy. For example, Listing 6 declares part of the **Display** class and the **update** method (Lines 24-26 of Listing 1). The **one** Alloy keyword denotes that the signature contains exactly one object.

Listing 6. DisplayUpdateDR Semantics (Part 1)

```
1  one sig Display extends Class {}{}
2  one sig update extends Method {} {
3      return = void
4      no update.param
5      update in Display.meth
6  }
```

The **Display.meth** expression denotes the set of methods declared in **Display**. Notice that there are invariants attached to **update**. It is a signature attached fact. It states some constraints about **update**. For instance, it is a void method and it is declared in the **Display** class. We add all constraints declared in all elements. The **in** Alloy keyword denotes the subset operator. It is important to observe that the **update** method cannot have parameters. If you would like to allow it to include any kind of parameters, you should add the following declaration: **update(..)**. It is important to observe that we do not include any constraint about the qualifiers of **update**. For example, since it is not presented whether the method is static in the declaration of **update**, this method can be static or non-static.

Listing 7. DisplayUpdateDR Semantics (Part 2)

```
1  one sig DisplayUpdate extends Aspect {...}{...}
2  one sig adv1 extends Advice {}{
3      adv1 in DisplayUpdate.advice
4  }
5  fact {
6      update in adv1.call
7      all m: Role − adv1 | update not in m.call
8  }
```
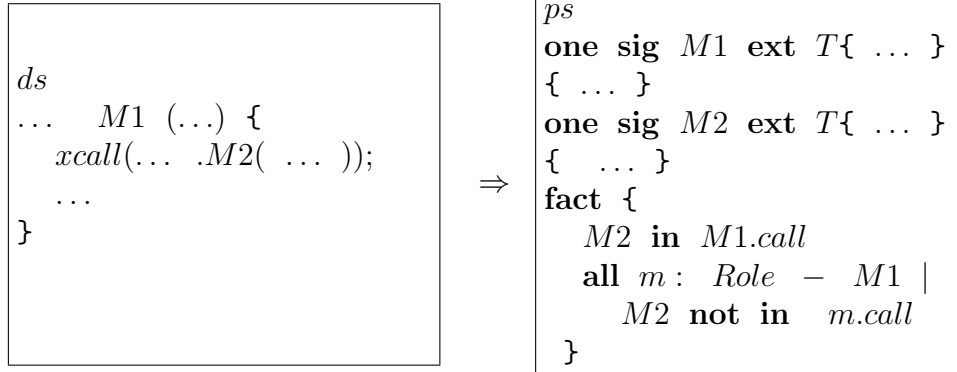
Listing 7 expresses the semantics of Listing 1 (Lines 20-22). It states that there is an advice declared in the **DisplayUpdate** aspect. The **all** and **not** Alloy keywords denote the universal quantifier and negation, respectively. Similar to a signature attached fact, an Alloy fact declares a set of invariants about the model. In the previous fact, we state that the **update** method cannot be called by any method, constructor or advice but the advice declared in the **DisplayUpdate** aspect. In our theory, the **Method**, **Constructor** and **Advice** signatures, which extend the signature **Role**, declare the relation **call**. This relation specifies all methods calls that are syntactically presented in the declaration of a method, constructor or advice. We map the other elements of Listing 1 similarly.

4.3. Translation

In this section, we present some of our general translations used in the example (Listing 1) to translate a specific design rule to its counterpart in Alloy. Translation 1 shows how the

xcall is mapped to its counterpart in Alloy. Each translation contains two templates. The left hand side (LHS) template contains a design rule in LSD. The right hand side (RHS) template shows an Alloy model specified using our theory.
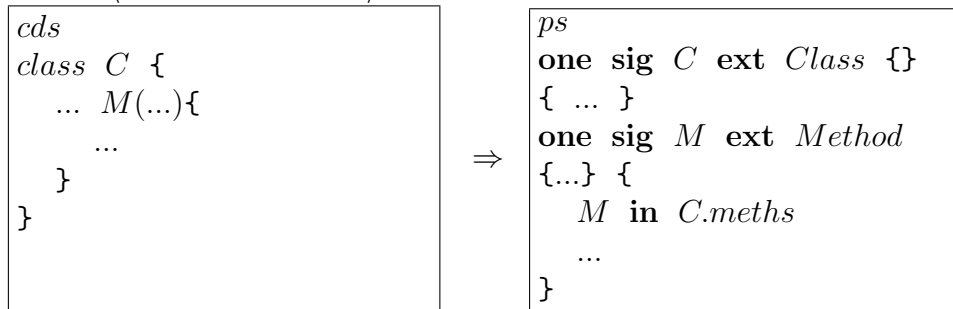
Translation 1 ⟨xcall⟩

```
ds
...    M1 (...) {
   xcall(...  .M2( ... ));
   ...
}
```

$\Rightarrow$

```
ps
one sig M1 ext T{ ... }
{ ... }
one sig M2 ext T{ ... }
{   ... }
fact {
   M2 in M1.call
   all m : Role − M1 |
      M2 not in  m.call
}
```

The ds and ps meta-variables denote a set of design rule declarations and Alloy signatures, respectively. On the RHS of Translation 1 there is one fact stating that $M2$ must be called in $M1$. Additionally, it cannot be called by any other constructor, method or advice. This translation is used in the example presented in Section 4.2 in order to map **xcall** in Listing 7.

Translation 2 presents how the rules introduced by a method declaration is expressed in our language. This translation is used in the example presented in Section 4.2 in order to map the **update** method in Listing 6. We have 20 translations for classes, aspects, advices, behavior rules. Each translation deals with one construct. Therefore, Translation 2 just presents the parts of the language translated to our theory in Alloy. For example, it does not show how the parameters and qualifiers are mapped. We have other translations for them.

Translation 2 ⟨method declaration⟩

```
cds
class C {
   ... M(...){
      ...
   }
}
```

$\Rightarrow$

```
ps
one sig C ext Class {}
{ ... }
one sig M ext Method
{...} {
   M in C.meths
   ...
}
```

4.4. Discussion

The first-order logic constraints can be easily mapped to Alloy. When specifying design rules, it may happen to introduce an inconsistency, which can be detected by the Alloy Analyzer. For example, consider that in the **DisplayUpdateDR** example (Listing 8) we have a **call** and a **xcall** to the same method **update** in different parts (**FigureElement.set*** and in the **DisplayUpdate** advice). Using our translations, we yield the constraints that are contradictory (Listing 9).

Listing 8. Inconsistent Design Rule

```
1   dr DisplayUpdateDR [FigureElement, DisplayUpdate, Display] {
2       class FigureElement {
3           public void set*(..) {
4               call(* Display.update());
5           }
6       }
7       aspect DisplayUpdate {
8           public pointcut stateChange(FigureElement fe): target(fe) &&
9               (call(* FigureElement+.set*(..))    ||
10               call(* FigureElement+.moveBy(..))   ||
11               call(FigureElement+.new(..)));
12
13           after(): stateChange() {
14               xcall(* Display.update());
15           }
16       }
17       ...
18  }
```

Listing 9. Inconsistent Alloy Constraints

```
1   update in methSet.call
2   all a: Role − adv1 | update not in a.call ...
```

As explained before, the **Role** signature represents all methods, constructors or advices. The variables **adv1** and **methSet** represent the advice declared in **DisplayUpdate** and the methods **set*** declared in **FigureElement** in Listing 8, respectively. This inconsistency may be difficult to detect in a larger specification. Using our approach, the Alloy Analyzer can detect that the design rule is inconsistent since we map each design rule to a specific Alloy model. In this case, the Alloy Analyzer performs a complete analysis, since we know all elements involved. Additionally, the tool contains a functionality (the unsat core) that allows us to extract the minimum set of constraints that is making the model inconsistent. In the previous example, the Alloy Analyzer highlights the problem in the Alloy specification. As a future work, we intend to build a tool that highlights the problem in LSD.

We have focused on the semantics of the language. As a future work, we intend to formalize the type system and the well-formedness rules of the language. We have identified a number of well-formedness rules. Mostly are based on the AspectJ and Java static semantics (well-formedness rules and type system) since we have closely followed them in order to be easier to understand by developers. We also have some rules specific to our language. We are aiming at choosing another language to specify the type system and well-formedness rules. Based on our experience, since the specification will increase, it will be difficult to have the benefits of the Alloy Analyzer analysis. Moreover, the current version of Alloy does not allow recursive definitions directly. This may turn the specification difficult to read and maintain.

We have a tool support that implements many of the translations proposed from LSD to Alloy. We have implemented them in parallel when specifying the translations rules. The case studies and some toy examples specified in our language were mapped to their counterparts in our theory in Alloy using the tool support. Moreover, most translations are simple and deal with one construction each time. These facts increase our confidence that the set of translations is complete.

5. Evolution Scenarios

In this section we discuss some evolution scenarios (used in other work [Griswold et al. 2006]) that could occur in the system presented in Listing 1. The main objective is showing how the existence of design rules can help to identify problems and also help to solve them.

Adding Color to Figure Element: One possible evolution to the system is supporting colors. Considering that color is an attribute common to all figure elements, it is natural to add it to the **FigureElement** class. As occurs to other **FigureElement** attributes, its value will be changed by invoking the **setColor** method. Thus the design rule will not be changed, but it will help developer to understand the change constraints. If the developer does not respect the **naming convention**, which establishes that all methods that change **FigureElement** state must be called **moveBy** or **start with set**, and tries to define a method called **changeColor**, the developer will receive an error message.

Modifying moveBy to Call set Methods: Another possible evolution is changing **moveBy** method replacing direct assignment to attributes by calls to corresponding **set methods**. This change would not be allowed by the design rule and will generate an error because, as shown in Listing 1 (Line 10), calls to methods with names starting with **set** are prohibited within change methods (methods with names starting with set, or exactly moveBy or any constructor). In this situation, the change must be aborted or the design rule must be adapted. This last option must involve the aspect developer because some assumptions, like "no call to change methods within change methods", will not be true anymore. In this case, developers must agree in a new set of rules and rewrite them in the design rule. The current design rule helps developers to understand the dependencies.

6. Related Work

Sullivan [Sullivan et al. 2005] presented a comparative analysis between an AO system developed following the widely cited oblivious approach and the same system developed with clear design rules that document interfaces between classes and aspects. This last approach promises benefits when relevant crosscutting behaviors are anticipated and when new code, anticipated or not, can be written against existing interfaces (design rules). Its main problem is expressing the design rules in natural language, leading to sometimes long and ambiguous interpretation.

Griswold [Griswold et al. 2006] showed how to express (using AspectJ constructs) part of the design rules into a set of **Crosscutting Programming Interfaces (XPIs)** that are useful to document and check part of the design rules (contracts). Although it was possible to check part of the design rules, the use of a language not designed to this purpose leads frequently to complex specifications (contracts imposed by aspects).

Program Description Logic (PDL) [Morgan et al. 2007], inspired by the pointcut language in AspectJ, allows succinct declarative definitions of programmatic structures which correspond to design rule violations. However, PDL is useful only in OO systems and does not support a clear definition of rules for parallel development.

Open Modules [Aldrich 2005] supports the definition of an interface composed by a set of pointcuts that can be advised by clients, introducing a form of encapsulating

join points occurring inside a module and protecting them from external advising. Although join point hiding is an important concern, it does not provide information to the aspect developer (beyond exported join points) that could be useful for establishing design rules that serve as interface between the OO and AO developers, like LSD provides.

7. Conclusion and Future Work

We discussed how LSD can improve modularity due its interface notion with a solution that eliminates the ambiguity and reduces the complexity found in other approaches, like XPIs [Griswold et al. 2006]. With LSD, it is possible to independently develop classes, interfaces and aspects, as long as the design rules are preestablished.

We noticed that design rules are important in many types of aspects, but there are some basic aspect implementations that are so general that make no reference to classes or interfaces (no coupling), like for example basic Tracing and Logging aspects, that do not demand design rules. LSD does not impose restrictions that constrain the development of these basic aspects, since it is not obligatory that aspects implement design rules.

LSD requires the creation of new artifacts (design rules) that demand enough experience from software designers. Additionally, developers must get used to new language constructs. In spite of that, explicitly expressing design rules, and specially being capable of verifying them, eases the task of developing new components and also adapting existent components.

Another important point is that although we consider defining design rules as a necessary step to modular AO development, it is possible to write classes and aspects, and later establish the design rules that were used. In this case, they will not help during development but will be useful for preventing errors and to give assistance to developers during software maintenance and evolution. This approach can be used whenever the best design is unknown (lack of experience) or in agile development processes.

Another interesting point was using Alloy to specify LSD semantics. It helped us to deeply understand some constraints that we wanted to express and lead us to think about some options and take decisions that influenced LSD expressivity and understandability.

As future work we plan to continue the formal semantics definition, specifying the type system and well formedness rules. Also, more constructs (specially more behavioral rules) are necessary to support other constraints. Additionally, we plan to evaluate the language in more real case studies, that include AO SPL. Finally, we are extending the AspectBench Compiler for AspectJ (abc) [Avgustinov et al. 2005] to support LSD constructs, but using the new frontend based on the AO meta-compiler JastAdd [Ekman and Hedin 2007], which supports a constrained form of static AOP (intertype declarations) that associated to demand-driven evaluation and attribute grammars, enable composable extensions and flexible modularization.

Acknowledgments

---

[1]http://www.ines.org.br

References

Aldrich, J. (2005). Open modules: Modular reasoning about advice. In *ECOOP '05: Proceedings of 19 th European Conference on Object-Oriented Programming*, pages 144–168. Springer.

Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, US. ACM.

Clifton, C. and Leavens, G. T. (2002). Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Foundations of Aspect Languages*, pages 33–44.

Dósea, M., Neto, A. C., Borba, P., and Soares, S. (2007). Specifying design rules in aspect-oriented systems. In *First LA-WASP*, João Pessoa, Brazil.

Ekman, T. and Hedin, G. (2007). The jastadd extensible java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, US. ACM.

Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., and Rajan, H. (2006). Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60.

Jackson, D. (2006). *Software Abstractions: Logic, Language and Analysis*. MIT press.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *ECOOP: Proceedings of the European Conference on Object-Oriented Programming*.

Lopes, C. and Bajracharya, S. (2006). Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value. *LNCS Transactions on Aspect-Oriented Software Development I*, pages 1–35.

Morgan, C., Volder, K. D., and Wohlstadter, E. (2007). A static aspect language for checking design rules. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 63–72, New York, NY, USA. ACM Press.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

Ribeiro, M., Dósea, M., Bonifácio, R., Neto, A. C., Borba, P., , and Soares, S. (2007). Analyzing class and crosscutting modularity with design structure matrixes. In *SBES '07: Proceedings of 21 th Brazilian Symposium on Software Engineering*, pages 167–181.

Steimann, F. (2006). The paradoxical success of aspect-oriented programming. *SIGPLAN Not.*, 41(10):481–497.

Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H. (2005). Information hiding interfaces for aspect-oriented design. In *ESEC/FSE'05*, pages 166–175, New York, NY, USA. ACM.