

# Verification of Aspect-UML models using Alloy

Farida Mostefaoui  
DIRO, University of Montreal  
Quebec, Canada  
mostefaf@iro.umontreal.ca

Julie Vachon  
DIRO, University of Montreal  
Quebec, Canada  
vachon@iro.umontreal.ca

## ABSTRACT

Aspect-oriented (A-O) programming has emerged as a promising paradigm to improve modularity by providing mechanisms to capture and execute crosscutting concerns in software applications. Among others, A-O allows developers to incrementally modify the behavior of a base program, by introducing *aspects* which implement crosscutting concerns having effects at various points throughout a program. Hence, despite the clean separation of concerns in aspect-oriented systems, it remains difficult to predict the effect of a given aspect on this base program. Once woven, does an aspect still achieve what it was intended for? Does it violate base program properties that should be preserved? Does it interfere with the properties of other aspects? We propose to address these questions through the formal analysis and verification of A-O system model. More precisely, this work considers A-O models written in Aspect-UML (our UML profile). Having no regards to A-O language specific features, these models might just as well be the result of a forward as of a backward engineering process. In particular, this article explains how Aspect-UML models can be specified within the Alloy model analyzer and how aspect interactions can therefore be verified.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*Assertion checkers, formal methods, model checking programming by contract*; D.2.1 [Software Engineering]: Requirements/Specifications—*Languages*

## General Terms

Model verification

## Keywords

Aspect oriented modeling, UML, Alloy, pre/post conditions, invariants, verification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop AOM '07, March 12-13, 2007 Vancouver, British Columbia, Canada Copyright 2007 ACM 1-59593-658-5/07/03... \$5.00

## 1. INTRODUCTION

The aspect-oriented paradigm [10] allows core functionalities and crosscutting concerns (i.e. aspects) composing a system to be programmed independently in separate modules. This is possible thanks to the A-O compiler that later "weaves" aspect behaviors at specified join points within the base program. Although improving system modularity, reuse and maintainability, A-O still faces an important criticism (as discussed in [12]) concerning the difficulty to reason about aspect interactions once they are woven into compiled code. The aspect community has recently shown an increased interest for this issue. To tackle aspect composition and conflict detection, we advocate an approach dealing with aspect interactions at model-level and relying on formal verification techniques. Being model-based, our approach remains independent from language specific features, contrarily to other solutions based on static analysis of programs. Using formal models, we can count on a meta-model having a well-defined semantics, a key element for automatic analysis. We consider models which can easily be enhanced with additional declarative semantic specifications contributing to a finer analysis of aspect interactions. Most program analysis approaches are missing this kind of information when time comes to reason about aspect and object behavior. Compared to programs or requirement specifications, models present a certain flexibility: they can be more or less abstract. They can be refined and henceforth reveal conflicts which could not have been detected by a requirement analysis (which usually offers little insights on concrete conflicts due to design decisions).

The approach described in this paper considers the analysis of aspect interactions in A-O models written in Aspect-UML [20, 13]. Aspect-UML is a simple profile extending UML with fundamental A-O concepts (aspects, advices, pointcuts, joint points and crosscutting dependencies). It also allows for formal annotations, such as pre and post conditions, to accurately specify the behavior of sensitive elements such as join points and advices as well as context passing at pointcuts. Thanks to these annotations, Aspect-UML models provide additional information to analyze aspect interactions from a semantic point of view. This work therefore goes a step further than traditional approaches based on program static analysis which often fail to decide about semantic conflicts between aspects. With no regards to A-O language specific features, Aspect-UML models might just as well be produced within the context of a forward as of a backward engineering process (model extraction).

Aspect-UML models can be checked for conflicting aspect interactions. One way to automate the verification process is to translate Aspect-UML models into an Alloy specification. Alloy provides a simple model specification language based on first order logic as well as a model analysis and simulation tool [7]. An Alloy model is composed of a set of signatures defining objects<sup>1</sup> sets and relations over them. This model can be further constrained by predicates and assertions. A model is an abstraction which actually defines a set (possibly infinite) of finite model instances. Alloy implements model verification by searching for model instances satisfying some specified property. A model can be checked to be valid or satisfiable within model instance size constraints. Indeed, the Alloy analyzer limits the search to model instances whose size (in terms of objects) is inferior to some bound fixed by the user. Alloy justifies its verification approach by putting forward the *small scope hypothesis* according to which counterexamples invalidating a model tend to occur in small models instances already.

To verify Aspect-UML models, we first assume that the base system and the aspects have both been proved to be individually correct. By translating Aspect-UML models into Alloy, our formal verification process aims to reveal the following kinds of aspect interactions problems: (1) violation of local properties: an advice or a join point's pre/post condition is violated due to the weaving of an aspect.; (2) violation of a class, aspect or system invariant due to the addition of an aspect.

The organization of the paper is the following. Section 2 presents the Aspect-UML profile, and illustrates, by means of a simple example, the various concepts introduced by the profile. Section 3 introduces the Alloy analyzer and describes how Aspect-UML models can be translated into Alloy models. Section 4 outlines the formal verification of these models, using Alloy's model analyzer, to detect conflicting interactions between aspects. The article ends with a short discussion of related contributions and gives an overview of future work.

## 2. ASPECT-UML PROFILE

UML [16] is a general purpose modeling notation for specifying and visualizing software systems. It has emerged as the standard modeling language endorsed by Object Management Group (OMG). To fulfill modeling needs of specific domains, UML provides extension mechanisms such as stereotypes, tagged values and constraints. Extensions defined to model the particular elements of a domain are gathered into a UML profile.

To model A-O systems at an early stage of the development life-cycle, we proposed in [20] a UML profile called "Aspect-UML". This profile is a very natural extension of UML, which introduces the basic concepts of the aspect paradigm, within both class and use case diagrams. Concerned with the verification of aspect interactions, this profile is enhanced with formal annotations, such as pre and post conditions, to accurately specify the behavior of sensitive elements such as join points, advices and pointcuts [13].

<sup>1</sup>Alloy is not object-oriented. Objects are similar to records. They are defined by ADT signatures and have no implicit identity.

To illustrate our modeling approach, we use an application example inspired from the one given in [3] describing an aspect implementation of a telephony application which handles phone calls. The example application is a simple simulation of a telephony system in which customers *initiate*, *accept*, *drop* and *merge* both local and long distance calls. The basic system provides core functionalities to simulate customers and connections. To these basic functionalities, one would like to add the timing, billing and interrupting features described below.

- The timing feature is concerned with timing the connections and keeping the total connection time per customer. It intervenes at the beginning and the end of a connection.
- The billing feature is concerned with charging customers for the calls they make. A charge per connection is calculated and, upon termination of a connection, it is added to the appropriate customer's bill.
- The interrupting feature is used for handling busy lines by interrupting the callee. It intervenes at the beginning of a connection, by checking if the destination is busy, in which case the callee is interrupted.

Figure 1 shows how the timing, the billing and the interrupting callee requirements are integrated into the UML class diagram, using Aspect-UML conveniences. These crosscutting requirements are respectively captured by aspects *Timing*, *Billing* and *Interrupting* which are depicted as UML classifiers decorated with stereotype  $\ll Aspect \gg$ . These aspects crosscut the basic application via two pointcuts modeled as special interfaces named *OpComplete* and *OpDrop* (which are also depicted by classifiers accordingly stereotyped). A  $\ll crosscuts \gg$  dependency relationship is then used to related each pointcut to the join points it denotes.

The *OpComplete* pointcut interface contains an abstract operation named *opComplete(c:Connection)* to be executed when one of its join points is reached. Similarly, the *OpDrop* interface contains an abstract operation *opDrop(c:Connection)*. Let's consider the *Timing* aspect for example (The *Billing* and *Interrupting* aspects are modeled similarly). It implements both the *OpComplete* and *OpDrop* pointcut interfaces and thus provides a corresponding advice to implement each of them. Each advice implementing a pointcut interface is to be executed at its crosscutted join points. A realization relationship therefore relates each aspect to the pointcuts it implements. As for advices, they are annotated with either one of the stereotypes  $\ll before \gg$ ,  $\ll after \gg$  or  $\ll around \gg$ , depending on whether they must be executed before, after or in place of the join points referenced by the pointcut.

As mentioned, Aspect-UML models are to be enhanced with annotations and constraints which formally specify model fragments such as join points, advices and pointcuts. Indeed, the semantics of these elements is required for the verification of aspect interactions.

Aspect-UML constraints are specified directly on the class diagram using UML notes (shown as rectangles with down-

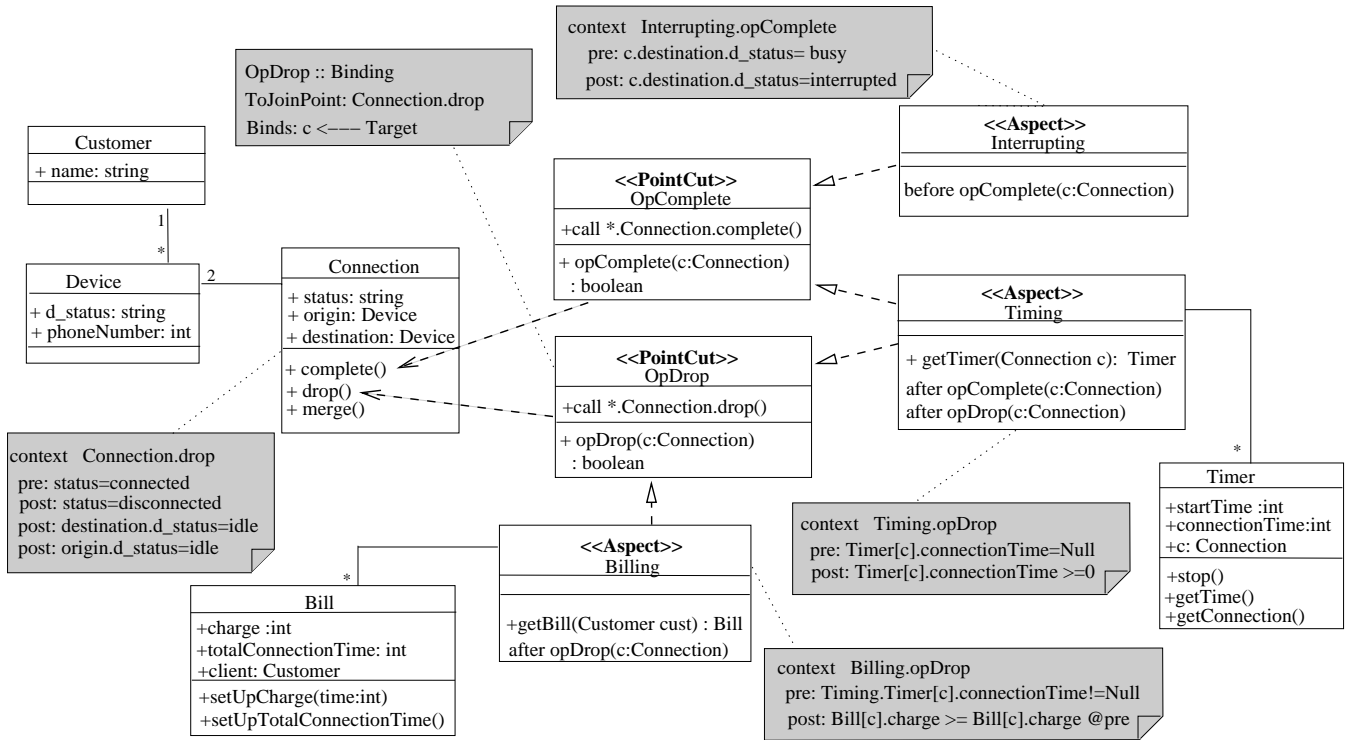


Figure 1: Class diagram for the Telecom system

right corner bent over on Figure 1). Alternatively, these constraints can be listed in some independent text file. More details about the notation used for annotations and constraints can be found in [13].

- **Specification of join points and advices.** Join points and advices are both associated to operations. The behavior of these operations is specified declaratively using pre and post conditions (e.g. see the annotation attached to the *drop* operation in class *Connection* on Fig. 1). Conditions are formulated using a notation similar to OCL.
- **Specification of pointcuts.** To adapt to the context where they are woven, advices often need to capture the contextual data captured by the join point. Pointcuts indeed allow to expose the context of join points. This context typically contains the identity of the object invoked by the method called/executed at the join point and the actual parameters of this method call. Pointcut specification defines how the execution context is passed from join points to related advices. Aspect-UML proposes a simple notation for pointcut specification (e.g. see the annotation attached to the *OpDrop* pointcut on Fig. 1).
- **Precedence constraints.** If several aspects crosscut a base model at the same join point and offer advices of the same type (i.e. *before* or *after*), the developer can clear up execution ordering ambiguities between advices by defining a precedence relationship between the conflicting aspects and by adding it to the class diagram. (e.g. A priority annotation *Timing < Billing*

could have been added to disambiguate the execution order of respective aspects at join points).

### 3. SPECIFYING ASPECT-UML MODELS IN ALLOY

#### 3.1 Overview of Alloy

Alloy [7, 2] is a structural modeling language, based on first-order logics and designed for the specification of object models through graphical and textual structures. It is based on ideas inspired from Z [17] and the many attempts to formalize object modeling<sup>2</sup>. An Alloy model is a structured specification composed of the following elements.

- **Signatures.** A signature define a data domain and its structure. Similar to abstract data types (ADT), a signature introduces a basic type (a typed set of objects) and a collection of relations (called the signature's fields) over defined sets of objects. Signature extension (similar to ADT subsorting) is a powerful feature used to support hierarchical specification.
- **Facts.** Facts are explicit constraints on relations and types defined by signatures. Facts always hold. They are therefore verified by all the instances of the model.
- **Predicates.** Predicates are expression intended to express constraints on objects defined by a model. They can be applied when needed. Instances of a model can be checked to satisfy a given predicate.

<sup>2</sup>Alloy is not object-oriented but simulates some O-O concepts using ADTs.

- **Assertions.** Assertions follows from the model’s facts. Assertions therefore specify constraints intended to be valid i.e. true for all the model’s instances. A counterexample is generated by the Alloy analyzer when an assertion does not hold.

More than a specification language, Alloy provides an analyzer which can, among other things, check the validity of assertions over a model, modulo the size of model instances being considered. Indeed, to check if a model is valid or satisfiable, the analyzer requires the user to specify upper bounds on the number of objects of each signature which can be contained in a model instance. This therefore entails the search through model instances to be finite. Instance exploration can be controlled. Simple and flexible, Alloy globally appeared to be an adequate tool to achieve our goal, that is to analyze aspect interactions in Aspect-UML models.

### 3.2 Specification of an Aspect-UML model in Alloy

This section explains how Aspect-UML models can be translated into Alloy specifications so they can later be analyzed using this tool. Aspect-UML models provides both structural and semantic information. The translation shall therefore handle both type of information.

#### Translation of Aspect-UML structural elements into Alloy.

Table 1 summarizes the correspondence which can be established between the structural elements of an Aspect-UML model and the elements of an Alloy specification.

**Table 1: Translation of Aspect-UML structural elements into Alloy concepts**

Aspect-UML	Alloy
-Class	-Signature Declaration
-Aspect	-Signature Declaration
-Attribute, association	-Relation defined by a signature field

Classes and aspects are translated into signature declarations in Alloy. Attributes of a class or of an aspect are translated into relations within the corresponding signature. Similarly, associations between classes and/or aspects are also translated into relations. As for user-defined enumeration data types <sup>3</sup>, which are sets of values with no identity in UML (e.g. `connected` and `disconnected` in our case study), enumeration literals are each translated into a signature extending the signature defined for the enumeration type. For instance, the following Alloy code fragment specifies the class `Connection` appearing in the class diagram of the Telecom application (Figure 1).

```
sig Connection{
  status: Status,
  origin, destination: Device}
```

```
abstract sig Status{}
```

<sup>3</sup>Alloy offers basic support for only two predefined data types i.e. for booleans and integers.

```
sig connected, disconnected extends Status{}
```

As for the `Timing` aspect, it is specified by the following Alloy code fragment:

```
sig Timing{
  effectiveConnections: set Connection,
  getTimer: effectiveConnections → some Timer}
```

```
sig Timer{
  startTime, connectionTime: Int,
  c : Connection}
```

#### Translation of Aspect-UML semantic annotations into Alloy.

To deal with the semantics of aspect interactions at join points, Aspect-UML models have been extended with annotations specifying the individual contract that each method/-advice operation (executing at the join point) must respect. These contracts are declared in terms of pre and post conditions. The Alloy model must therefore take into account these constraints.

Fortunately, the Alloy language allows us to go beyond the representation of static structure. We can model evolving systems through the use of predicates capturing the constraints that operations (methods and advices at join points) must satisfied before and after their execution. Let *op* be an operation (with parameter *s* of type *T*) whose behavior is specified by pre and post conditions respectively constraining parameter *s* to be greater than 0 before executing and to be less than 100 afterwards. The following predicate can be used to describe, in Alloy, the constraining effects of this operation. (Let’s recall that Alloy uses logic variables i.e. with single assignment).

```
pred p(s, s':T) {
  // precondition
  s > 0
  // postcondition
  s' < 100 }
```

In a correct Aspect-UML model, aspect interaction at join points must never cause the violation of operations’ individual pre and post conditions, whatever execution scenario is being considered. In the Alloy model, this means the weaving of aspects will require the composition of the advice’s constraints with the related join point’s constraints. If there is no conflict, all model instances should satisfy the composite constraint thus induced by aspect weaving. To illustrate this, let’s consider the Alloy predicates that we have defined to constrain the woven execution of methods and advices at the `Drop` join point in the Telecom application. In the following steps, we will explain how to specify (1) the constraints applying to the `drop()` method, (2) the constraints applying to advices `opDropTiming` and `opDropBilling`, (3) the constraints imposed by the composition of those advices, and finally (4) the constraints induced by the weaving of composed advice at the `drop()` join point.

1. The behavior of method `drop()` must satisfy the constraints imposed by its pre and post conditions: preconditions must be satisfied by the model for the method to execute and the method execution must leave the model in a state satisfying the postconditions. In this case, the status of the connection should be set to *connected* when `drop()` is called, while its execution should afterwards leave it to *disconnected*. The before and after states of the connection are respectively captured by variables *c* and *c'*. These constraints can be specified in Alloy using a predicate definition, as shown below.

```
pred drop(c, c': Connection) {
  c.status=connected &&
  c'.status=disconnected &&
  c'.origin.d_status=idle &&
  c'.destination.d_status=idle }
```

2. The behavior of advice `opDrop` in the **Timing** aspect is specified in a similar way. Its precondition constrains the timer associated to the connection (being dropped) not to have yet a registered connection time value. On the opposite, its postcondition forces the timer to have a registered connection time value greater or equal to 0. These constraints compel the timer to change state, which consequently forces the **Timing** aspect to also change state since it must update its field containing the list of timers. Parameters *t* and *t'* are used to denote the state of the **Timing** aspect before and after the execution of the `opDrop` advice. As for the connection, its state (modeled by parameter *c*) remains the same. The following Alloy predicate is used to specify the constraints imposed by the `opDrop` advice.

```
pred OpDropTiming(t,t':Timing, c:Connection) {
  t.useTimer[c].connectionTime=NULL &&
  t'.useTimer[c].connectionTime ≥ 0 }
```

An Alloy predicate must be defined to specify the constraints imposed by the pre and post conditions of each advice and joint point method appearing in the Aspect-UML model. In the case of **Billing's** `opDrop` advice, pre/post conditions force the total charge on the caller's bill to augment if the timer has recorded a connection time greater or equal to 0. Consequently, only the bill changes state. This state change can be observed thanks to parameters *b* and *b'* in the `OpDropBilling` predicate below. This Alloy predicate specifies the constraints compelled by the execution of **Billing's** `opDrop` advice.

```
pred OpDropBilling(b,b':Billing, t:Timing,c:Connection) {
  (t.useTimer[c].connectionTime ≥ 0) &&
  (b.useBill[c.origin.owner].charge ≤
  b'.useBill[c.origin.owner].charge) }
```

Some may have remarked that the **Billing** aspect implicitly requires the services of the **Timing** aspect. This way of designing aspects is not recommended: aspects should be defined independently from each other. Nevertheless, if the **Timing** aspect is retracted, our analysis will be able to detect **Billing's** failure at the `drop` join point.

3. When more than one advice of a same type (i.e. before or after) are executed at a same joint point, these *concurrent* advices need first to be composed together before being

weaved at this joint point. Concurrent advices having priorities are composed sequentially following the order specified by the precedence constraints. Sequential composition forces postconditions of an operation in the sequence to imply the preconditions of the following operation. If concurrent advices are not prioritized, they are composed sequentially but in a non deterministic order. All possible ordering will therefore need to be checked. Again postconditions of an operation will be constrained to imply the preconditions of the following operation.

For example, **Timing** and **Billing** are non prioritized concurrent aspects interacting at the `drop` join point. Their respective `opDrop` advice need to be composed in a non deterministic order. To force Alloy to go through all possible advice ordering, we define a predicate called `executeOpDrop` which let's Alloy the free choice to let predicate `OpDropBilling's` postconditions imply the preconditions of predicate `OpDropTiming`, or the opposite. When checking validity of an assertion, the Alloy analyzer will thus go through all possible executions of this predicate, therefore trying all possible ordering of these two advices pre and post constraints.

```
pred executeOpDrop(t,t':Timing, b,b':Billing,c:Connection){
  OpDropTiming(t,t',c) && OpDropBilling(b,b',t',c)
  ||
  OpDropBilling(b,b',t,c) && OpDropTiming(t,t',c) }
```

4. The final weaving of the advice resulting from the composition of `OpDropTiming` and `OpDropBilling` is to occur just after the execution of the `drop()` join point. Predicate `weaveAtOpDrop`, given below, defines the constraints compelled by the weaving. This predicate clearly forces the context passing constraint  $c \leftarrow target$  (defined by `pointcut OpDrop` in the Aspect-UML model) to be respected thanks to parameter *c'*. This parameter represents the state of the connection just after executing the `drop` method. It is passed to predicate `executeOpDrop` defining the set of constraints induced by the composite advice to be executed precisely at this point. Finally, since the effect of this advice must not contradict initial postconditions of the `drop` join point, we force the last state of the connection to be disconnected, as initially expected.

```
pred weaveAtOpDrop(t,t': Timing, b,b':Billing, c,c': Connection)
{ drop(c,c') &&
  executeOpDrop(t,t',b,b',c') &&
  (c'.status=disconnected)}
```

## 4. VERIFICATION OF ASPECT-UML MODELS USING ALLOY

The verification approach we propose, using the Alloy analyzer, is known as Model-Based Verification "MBV". MBV is a systematic approach to find errors in software requirements, designs or code [6]. The approach incorporates mathematical formalism, in the form of models, to provide a disciplined and logical analysis practice. MBV involves creating formal models of system behavior and analyzing these models against formal representations of expected properties. Figure 2 shows our verification process using the Alloy analyzer into the development flow of our aspect-oriented approach.

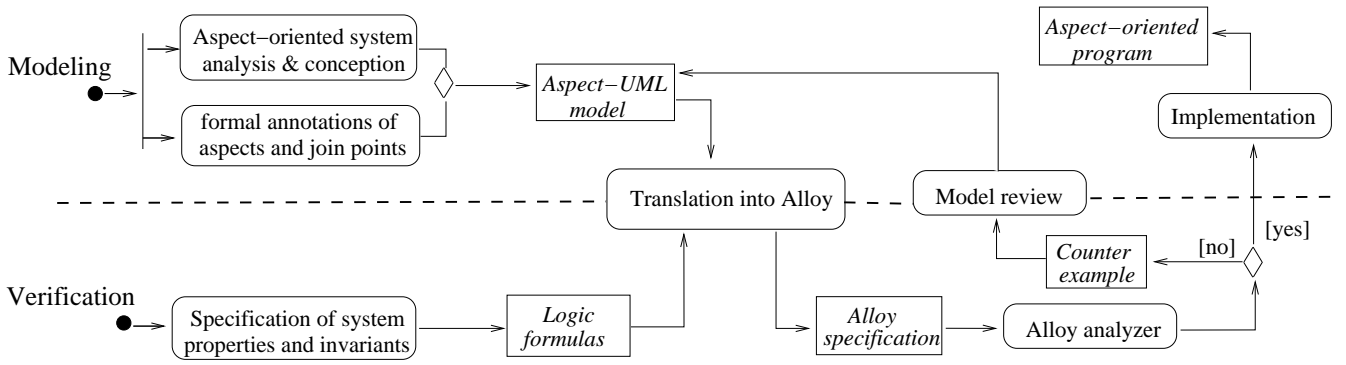


Figure 2: Aspect-oriented methodology

Assuming the base system and the aspects are both individually correct, the formal verification process we propose focuses on the discovery of errors due to aspect interactions (either with the base program or with other aspects). Aspect-UML allows us to define formal micromodels of A-O systems, therefore capturing the essence of the weaving mechanism, without having to take into account the whole set of system requirements or design decisions. Retaining solely critical parts (join points, advices and weaving indications), we can focus on the verification of aspect integration within the base system as well as on interactions (especially implicit ones) between aspects. More precisely, our verification approach aims to reveal important interference problems such as:

1. **Violation of local properties.** Violation of the base program specification (pre and post conditions) induced by woven aspects; or violation of an advices local specification (pre and post conditions) induced by the base program or some other woven aspect;
2. **Violation of global properties.** Violation of a system properties, such as invariants, following the introduction of new aspects.

#### 4.1 Alloy analysis of Aspect-UML models

The Alloy analyzer is a tool, based on algorithms of SAT solvers, for analyzing small model instances. The D. Jackson’s observation called *the small scope hypothesis* [8] is the fundamental premise that underlies Alloy’s analysis. This hypothesis states that negative answers tend to occur in small model instances already. Absence of errors in small instances can thus boost the confidence we may have in a positive answer for the whole verification. In other words, most flaws in models can be observed in small instances, since they arise from some shape being handled incorrectly. Whether the shape belongs to a large or small instance makes no difference. So if the analysis considers all small instances, most flaws will be revealed. So a good way to use Alloy analyzer is to start with a small scope analysis, and increase it gradually until a fault is found or until you are satisfied with the approximation.

Alloy supports two kinds of automatic analysis: simulation and checking. The first one is used to demonstrate the consistency of a given predicate by generating a state of the

model that satisfies the predicate. The second one is used to prove the validity of assertions by attempting to generate a counterexample. The analysis is always carried within a user-defined bounded scope, thus limiting the size of model instances being explored. When the Alloy analyzer finds a counterexample the assertion is necessarily false (violated), otherwise no conclusion can be formulated about its validity.

Once Aspect-UML models are translated into Alloy, the analysis of aspect interactions can be carried as follows:

1. **Verification of local properties.** At a given join point, an aspect advices may happen to interfere with the base system or the other aspects by violating their pre/post conditions. The corresponding Alloy models is composed of a set of predicates defining the constraints which the model must satisfy in all cases. It therefore suffices to formulate an assertion describing the composition of constraints entailed by weaving and to run the Alloy analyzer over it. Alloy will try to find a counterexample invalidating the assertion i.e. proving that at least one instance of the model doesn’t satisfy the set of specified constraints. Considering the verification of aspect interactions at a given join point in the Telecom application, we must formulate an assertion which 1) takes into account our initial hypothesis advocating the correctness of the base program and of individual aspects; and 2) specifies all the constraints which must be satisfied by the methods and the advices to be executed a the join point (as explained in the previous section) For example, in the Telecom application, to check if aspects **Timing** and **Billing** are interacting correctly at join point **drop** (i.e. they don’t violate the specification of the base system nor do they interfere with the specification of the other aspect), the Alloy analyzer was given the following assertion to check:

```

assert testWeaveAtOpDrop {
  all c: Connection, t:Timing, b: Billing |
  some ct: Connection, tt:Timing, bt: Billing |
  (c!=ct)&&(t!=tt)&&(b!=bt) => weaveAtOpDrop(t,tt, b,bt, c,ct)
}

```

This assertion is of the form (*condition* => *predicate*), where *condition* specifies our initial hypotheses stating that the base system and the aspects are individually correct at the join point **Drop**. In this case, the base system is assumed to be correct if the method **drop()** executes individually correctly, that means that the connection’s status is

effectively changed (i.e.  $c!=c'$ ) by this operation. Similarly, aspects are assumed to be individually correct if they ensure the correct individual execution of their respective advices `OpDropTiming` and `OpDropBilling`. In this case, this means we are considering models in which `Timing` and `Billing` do changed when execute ( $t! = t'$ ) and ( $b! = b'$ ). Under these assumptions, the Alloy analyzer is asked to check if predicate `weaveAtOpDrop` always hold. We recall that this predicate specifies the set of constraints induced by the weaving of aspect at the join point `drop`.

Thus, if the initial conditions hold than the weaving predicate must hold. Otherwise, if a counterexample is found while checking the assertion, this means the Aspect-UML model present some aspect interaction problem. When analyzing the Alloy model describing our Telecom Aspect-UML model, the Alloy analyzer found no counterexample while checking the assertion `testWeaveAtOpDrop`. We still can't be sure it is valid, but we do, at least, have better confidence it is.

**2. Verification of global properties.** System invariants can easily be specified by additional predicates in Alloy models. In the Telecom application, two significant invariants can be specified:

- For each state in the system, the number of connected connections must always equal or less than 50.
- The status of all devices (origins and destinations) engaged in connected connections is *not idle*.

These invariants can be specified by the following predicates:

```
pred limitedConnectedConnections {
  # {all c:Connection | c.status=connected} ≤ 50 }
```

```
pred deviceIdle {
  all d:Device, some c:Connection |
  (d in c.(origin + destination)) && (c.status=connected)
  => (d.d_status!=idle)}
```

Checking that these predicates hold over all states of our model requires checking the execution traces of the model. Alloy provides a library to handle ordered lists of states. We shall use this facility to extend the verification of our model to invariants. Since invariants are assumed to already hold on the base system, it suffices to check that they locally hold on the execution trace produced by woven advices at join points.

So far, we have only considered global properties which are invariants, i.e. properties that hold globally at all the states of the system. As for other safety and liveness properties, their verification using our approach is more complex, since we only have a partial view of the system, where only the behaviors of join points and advices are modeled. The verification of these properties will require compositional proofs to take into account individual proofs of the other parts of the system not being modeled. This is a problem we should soon tackle.

## 5. RELATED WORK AND CONCLUSIONS

This paper introduced our modeling and verification approach for aspect-oriented systems. High-level modeling of aspect-oriented systems can be achieved using our UML profile Aspect-UML. These models can be translated into equivalent Alloy specifications therefore enabling the verification of significant non-interference properties of aspect-oriented systems, using the Alloy analyzer. Of course, one may fear the use of automatic verification for its well-known state explosion problem. Our verification approach shall evolve such as to remain scalable and to master state explosion. By reserving verification to the critical weaving parts of the system, our approach already limits verification to a subset of the system's states. Compositional verification is also being considered to reuse computed verification results and thus reduce new proof obligations. The use of Alloy in our methodology also favors incremental checking of large systems: by bounding the size and the number of model instances being explored, partial verification results can be obtained even for very large systems.

Considerable research has been done in the area of Aspect Oriented Modeling to deal with aspect-orientation of software at an abstraction level. Most of these proposals suggest the use of UML for the modeling of A-O concepts. In particular [1] defines a UML profile to aspect-oriented software development where both static and dynamic views are handled, and [18] proposes a UML based design language to support AspectJ. These two approaches rest on the same extension mechanisms, where aspects are modeled as stereotype of UML classifiers and join points and pointcuts are modeled as stereotype of UML operations. Our modeling approach also relies on UML extensions using stereotypes. However, in our opinion, our definition of pointcuts as interfaces (1) provides a better modularization of Aspect-UML models, and (2) helps to identify potential aspects interactions on an Aspect-UML class diagram, by spotting aspects implementing a common pointcut interface. Moreover, our profile is enhanced with annotations, such as pre and post conditions to specify the behavior of pointcuts, join points and advices. Although, the idea of using semantic annotations has already been suggested in Aspect-oriented models, as in [9] that uses quality of service contracts to specify non functional properties; these annotations have never been really developed and used to improve model verification.

As for the detection of aspect interactions, to this day, few proposals have rigorously addressed aspect interactions problem and taken into account their semantics. Among those who did [4, 5, 19] most concentrate their analysis on the detection of potential conflicts revealed by the syntax. This type of verification is more limited regarding coverage and precision. To increase the accuracy and the significance of the verification process, our approach goes beyond syntax and takes into account the semantic of aspect-oriented systems.

Regarding formal verification of aspect interaction, authors in [15] present an approach to verify the properties of systems composed of multiple crosscutting concerns. The approach models concerns as sets of concurrent processes. Concerns are specified as labeled transition systems and composed through merge and override operators. The prop-

erties of the composed system are then verified using the LTSA model checking tool. Contrarily to our approach that relies on modular verification, the proposition of [15] is concerned with the verification of augmented systems. Thus, because the entire augmented system is treated at once, there may be some scalability problems. Authors in [11] use model-checking to modularly verify aspect advices. In their proposed prototype, the verification is done only for the state machine generated by the aspect code. Invariants of the original system are checked if they are maintained in the part of the system modeling the aspects, without re-verifying the entire system. However this approach, doesn't consider the situation where an aspect can disable a transition of the system it modifies, i.e. the approach doesn't consider overriding aspects. We also rely on modular verification, but our approach, on the one hand, deals with the formalization of models rather than programs. On the other hand, our verification approach considers the case where the operations of the base system can be overridden by woven aspects. Moreover, most proposals for the verification of aspect interactions deal with source code. We rather propose to tackle the problem from a top-down perspective by addressing modeling before system implementation.

Alloy was also used in [14]. This article describes the adaptation of a role-based aspect-oriented modeling method and defines a notion of aspect weaving as role merging. Since the weaving (model transformation) is done manually, the authors propose the use of the Alloy analyzer to verify if a given invariant is satisfied before and after each transformation. This approach essentially deals with the verification of invariants. In this sense, our solution seems more general as it addresses the detection of various kinds of property violation due to aspect interactions.

Future work shall focus on the implementation of an automatic model-translator from Aspect-UML to Alloy. Verification shall be extended to deal with various kinds of properties including safety and liveness. In the meantime, our A-O modeling process is also being fine tuned to allow some easy transition from early aspect models to aspect design models.

## 6. REFERENCES

- [1] O. Aldawud, A. Elrad, and A. Bader. A uml profile for aspect oriented software development. In *Int. Workshop on AOP at Int. Conf. on AOSD'2003*, 2003.
- [2] Alloy Homepage. <http://alloy.mit.edu>.
- [3] AspectJ Team. The aspectj programming guide. Xerox Corporation, 2002.
- [4] C. C. Clifton. *A design discipline and language features for modular reasoning in aspect-oriented programs*. PhD thesis, Iowa State University, Ames, IA, USA, 2005.
- [5] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse and interaction analysis of stateful aspects. In *3<sup>rd</sup> Int. Conf. AOSD*, pages 141–150, 2004.
- [6] D. Gluch, S. Comella-Dorda, J. Hudak, G. Lewis, and C. Weinstock. Model-based verificationscope, formalism, and perspective guidelines. Technical Report CMU/SEI-2001-TN-024 ADA396628, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2001.
- [7] D. Jackson. Alloy: A lightweight object modelling notation. In *MIT Laboratory for Computer Science: Cambridge, MA*, 2000.
- [8] D. Jackson. Software abstractions: Logic, language, and analysis. In *The MIT Press*, 2006.
- [9] J. Jézéquel, N. Plouzeau, T. Weis, and K. Geihs. From contracts to aspects in uml. In *Aspect-Oriented Modeling with UML Workshop at AOSD'02*, 2002.
- [10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97, LNCS*, pages 220–242, 1997.
- [11] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT'04/FSE-12*, 2004.
- [12] G. Leavens and C. Clifton. eds.: Foundations of aspect-oriented languagesworkshop. In *In Leavens, G., Clifton, C., eds.: AOSD05. Volume 4*, 2005.
- [13] F. Mostefaoui and J. Vachon. Approche basée sur les réseaux de Petri pour la vérification de la composition dans les systèmes par aspects. *RSTI - L'Objet*, 12(2-3):157–182, September 2006.
- [14] S. Nakajima and T. Tamai. Lightweight formal analysis of aspect-oriented modelss. In *Workshop on Aspect-Oriented Modeling at UML'04*, 2004.
- [15] T. Nelson, D. Cowan, and P. Alencar. Supporting formal verification of crosscutting concerns. In *REFLECTION 2001, Lecture Notes in Computer Science, Springer, vol.2192, pp. 153-169*, 2001.
- [16] Object Management Group. Unified modeling language: Superstructure. version 2.0, August 2005.
- [17] J. Spivey. *The Z Notation*. Prentice-Hall, 1992.
- [18] D. Stein, S. Hanenberg, and R. Unland. A UML-based aspect-oriented design notation for aspectj. In *International Conference on Aspect-Oriented Software Development*, pages 106–112. ACM, 2002.
- [19] M. Storzer, J. Krinke, and S. Breu. Trace analysis for aspect application. In *AAOS'03*, 2003.
- [20] J. Vachon and F. Mostefaoui. Achieving supplementary requirements using aspect-oriented development. In *ICEIS*, pages 584–587, 2004.