JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 6, No. 7, Special Issue: Aspect-Oriented Modeling, August 2007

Design-level Detection of Interactions in Aspect-UML models using Alloy

Farida Mostefaoui and **Julie Vachon**,
DIRO, University of Montreal, Quebec, Canada

Aspect-oriented (AO) programming has emerged as a promising paradigm to improve modularity by providing mechanisms to capture and execute crosscutting concerns in software applications. Among others, AO 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? These questions address the well known *aspect interaction problem*, encountered within the AO paradigm. This article tackles the interaction problem in the context of formal AO system model analysis and verification. To be more precise, this work considers AO models written in Aspect-UML (our UML profile). Aspect-UML does not depend on any AO language specific features nor is it associated with any specific development process. This paper first explains how Aspect-UML models can be translated into Alloy, a simple structural first-order logic modeling language which can be formally analyzed. Given this translation, it then demonstrates how Alloy's model analyzer can be used to verify aspect interactions of an Aspect-UML model.

1 INTRODUCTION

The aspect-oriented paradigm [1] 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 AO compiler that later "weaves" aspect behaviors at specified join points within the base program. Although improving system modularity, reuse and maintainability, AO still faces an important criticism (as discussed in [2]) concerning the difficulty to reason about aspect interactions once they are woven into compiled code. In the AO community, these issues are commonly known as the *aspect-interaction problem*. The introduction of new aspects can indeed compromise the local or global integrity of the system due to undesirable interactions with other aspects or with other base modules composing it. 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? In the remainder of the paper, we shall concentrate on the detection of such undesirable interactions (also called conflicts). To deal with aspect composition and conflict de-

Cite this article as follows: Farida Mostefaoui and Julie Vachon: *Design-level Detection of Interactions in Aspect-UML models using Alloy*, in Journal of Object Technology, vol. 6, no. 7, Special Issue: Aspect-Oriented Modeling, August 2007, pages 137–165,
http://www.jot.fm/issues/issues_2007_08/article6

tection, 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 AO models written in Aspect-UML [3, 4]. Aspect-UML is a simple profile extending UML with fundamental AO 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 by detecting semantic conflicts between aspects. With no regards to AO 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 aspect interactions. One way to automate the verification process is to translate Aspect-UML models into an Alloy specification. Alloy provides a simple specification language based on first order logic as well as a model analysis and simulation tool [5]. An Alloy model is composed of a set of signatures defining objects[1] 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 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.

With the aim to formally verify aspect interactions in AO models, this paper proposes a generic and modular approach for the translation of Aspect-UML models into Alloy's analyzer. Our translation takes into account not only structural elements

[1] Alloy is not object-oriented. Objects are similar to records. They are defined by signatures and have no implicit identity.

of Aspect-UML models but also the descriptive specifications of their behavior. On the one hand, it is worth noting that, to this day, most work concerned with aspect conflict detection is based on some static analysis of AOP source code (or of some AOP operational notation). It however seems important to also take into account the semantics of operations to identify the "real conflicts" i.e. the set of interactions which violate the desired properties of a system or its initial assumptions. This is the spirit of our verification approach. On the other hand, achieving formal verification of aspect interactions requires a non trivial formalization and practical solutions to cope with tractability issues. Alloy seemed to be the most appropriate framework to achieve this. There were already a few attempts at translating UML/OCL into Alloy [6, 7], but none seems yet to have come with a generic process to translate object interactions into Alloy. In this sense, our modular and automatizable translation of Aspect-UML (advice composition and weaving) sort of provides the milestones of a systematic approach for the formalization and verification of object and aspect interactions using Alloy.

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 two 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 describes the case study used to present our modeling and verification approach. Section 3 presents the Aspect-UML profile, and illustrates by means of the case study, the various concepts introduced by the profile. Section 4 gives a short overview of the Alloy language and of its analyzer. Section 5 describes how Aspect-UML models are translated into Alloy models. The application presented in Section 2 is used to illustrate and to show how the translation works. Section 6 outlines the formal verification of Aspect-UML models, using Alloy's model analyzer, to detect conflicting interactions between aspects in the case study. Section 7 discusses related work, whereas comments and ideas on future work conclude this article.

2 CASE STUDY

To illustrate our approach, we use an application example describing an aspect implementation of a telephony application which handles phone calls. This application is a simple simulation of a telephony system in which customers can `initiate` and `drop` calls. The basic system provides core functionalities to simulate customers, devices and connections. To these basic functionalities, other services can be added, such as the *interrupting callee* and the *call forwarding* features described below.

- The *interrupting callee* feature is offered to handle busy lines by simply interrupting the called party. It intervenes at the beginning of a connection,

by checking if the destination is busy, in which case the current connection is interrupted, and the destination status is set to idle.

- The *call forwarding* feature allows calls addressed to a party (destination) which is unavailable (busy) to be redirected to another telephone number (destination). It intervenes at the beginning of a connection, by checking if the destination is busy and if the number is in the forwarding list. If so, the connection is re-routed to another destination specified in the forwarding list.

In an aspect implementation of this application, the *interrupting callee* and the *call forwarding* features are both captured by aspects. At first sight, nothing let's think that these two aspect interfere with each other since the respective concerns they implement seem unrelated. In fact, adding the *interrupting callee* and the *call forwarding* aspects to the base phone service causes unexpected interactions between the aspects and with the base phone service itself. Using this example, we will demonstrate how our specification and verification approach can detect potential conflicts generated by the these aspects. To be specific, the interaction problem can be presented as follows:

Let P_1, P_2, P_3 be properties respectively satisfied by the *interrupting callee* aspect, the *call forwarding* aspect and the base system: (P_1)"Completing a connection to a busy destination, causes the interruption of the current connection"; (P_2)"If the phone number of a given destination is in the forwarding list, then calls to this destination must be forwarded consequently, if it is busy" and (P_3)"Emergency calls are never interrupted". If no undesirable interaction occurs, P_1, P_2 and P_3 should still hold in the final woven system; otherwise we can conclude there is a conflict.

3 ASPECT-UML

UML [8] is a general purpose modeling notation for specifying and visualizing software systems. It has emerged as the standard modeling language endorsed by the 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 AO systems at an early stage of the development life-cycle, we proposed in [3] a UML profile called "Aspect-UML". This profile is a 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 [4].

Figure 1 shows how the interrupting and the forwarding requirements are integrated into the UML class diagram, using Aspect-UML notation. These crosscutting

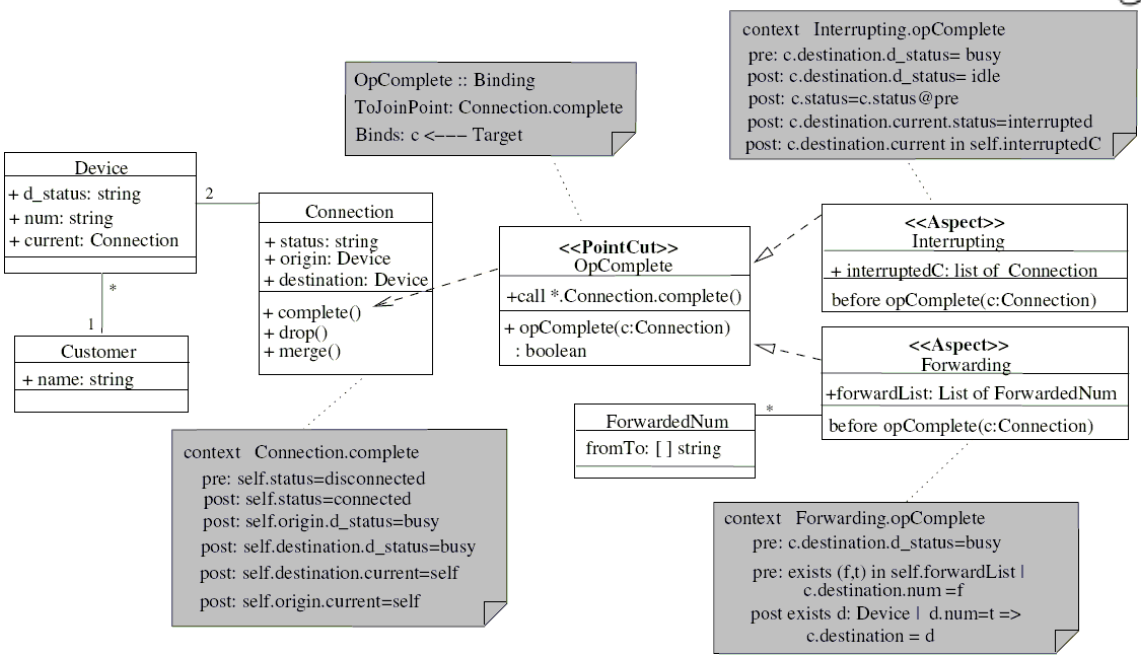


Figure 1: Aspect-UML class diagram for the Telecom application

requirements are respectively captured by aspects `Interrupting` and `Forwarding` which are depicted as UML classifiers decorated with stereotype ≪ `Aspect` ≫. These aspects crosscut the basic application through the `OpComplete` pointcut which is modeled as a special interface accordingly stereotyped. A ≪ `crosscuts` ≫ dependency relationship is then used to related the 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. Both the `Interrupting` and the `Forwarding` aspects implements the `OpComplete` pointcut interface and thus each provide a corresponding advice to implement it.

Each advice extending a pointcut interface will be triggered at all join points designated by the pointcut. In this case, `OpComplete` is composed of a single join point, that is the `complete` method of the `Connection` class. A realization relationship relates each aspect to the pointcuts it implements. As for advices, they are annotated with either one of the stereotypes ≪ `before` ≫, ≪ `after` ≫ or ≪ `around` ≫[2], depending on whether they must be executed before, after or in place of the join points referenced by the pointcut.

Aspect-UML models are to be enhanced with annotations and constraints which formally specify the semantics of model fragments such as join points, advices and pointcuts. The semantics of these elements is required to later achieve the verification of aspect interactions. Aspect-UML constraints are specified directly on the class diagram using UML notes (shown as rectangles with down-right corner bent

[2]To simplify the presentation of the translation in Alloy, the `around` advice will not be considered in the sequel.

over on Figure 1). Alternatively, these constraints can be listed in some independent text file. These are briefly covered below, for more detail refer to [4].

- **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 `complete` operation in class `Connection` on Figure 1). These specifications are given the following frame semantics: *any thing which does not appear in the pre or post conditions of an operation is assumed to remain unchanged while it is executing.*

- **Specification of pointcuts.** Aspect-orientation allow advices to take on the contextual data captured by join points. Pointcuts are used, among other, to expose the context of join points and pass it over to advices. 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. A pointcut specification defines how the execution context is passed from join points to related advices. Aspect-UML proposes a simple notation for pointcut specifications (e.g., see the annotation attached to the `OpComplete` pointcut on Figure 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 specifying it on the class diagram. A priority annotation `Interrupting` $<$ `Forwarding` could have been added to disambiguate the execution order of these two aspects. We will later see the impact of this omission on aspect interactions.

4 OVERVIEW OF ALLOY

Alloy [5, 9] is a structural modeling language, based on first-order logic and designed for the specification of object models through graphical and textual structures. It is based on ideas inspired from Z [10] and the many attempts to formalize object modeling[3].

To introduce the reader to the Alloy specification language, let's consider a simple example taken from [11]. This example serves as a means for illustrating the standard features of the language and their associated syntax. Suppose we want to specify a simple address book for an email client that maintains a mapping from names to addresses. To start, we define the sets of atoms that we will be using for denoting names and addresses. In Alloy, such sets of atoms are defined using signatures as shown below:

[3] Alloy is not object-oriented but simulates some O-O concepts using abstract data types.

```
sig Name, Addr {}
```

A basic type is implicitly associated to each declared signature and can thus be used in other declarations. With type for name and addresses elements, we can now specify what constitutes an address book. Since an address book constitutes a mapping from names to addresses, we introduce a `Book` signature containing a field `addr` linking names to addresses.

```
sig Book {addr: Name-> lone Addr}
```

In fact, `addr` is a ternary relation associating books, names, and addresses, such that $(B, N, A) \in$ `addr` means that, according to `addr`, book B links name N to address A. The expression `b.addr` denotes the mapping from names to addresses for book `b`. The keyword `lone` in the declaration indicates the relation multiplicity: in this case, each name is mapped to at most one address.

Moreover, Alloy allows the definition of sub-signatures as extensions of already defined basic signatures. Sub-signatures define subtypes (which are also basic types), that can in turn be extended. As specified in the definition below, `BookWithSpamAlert` is a subset of `Book`. In fact, `BookWithSpamAlert` is a special kind of address book, in which a subset of addresses are recognized as spams.

```
sig BookWithSpamAlert extends Book{spams: set Addr}
```

Relations defined by signatures can be constrained using facts. In Alloy a fact is a constraint that is assumed to always hold. For example, the following fact can be added to force any book that contains two identical addresses to map them over the same names.

```
fact {all  b: Book | all r,r': Addr | r=r' => b.addr.r=b.addr.r'}
```

We have shown so far how the structure of data domains can be specified in Alloy. As for operations, they are usually modeled using predicates and functions, which can themselves be seen as relations over defined signatures. As an example, let's consider defining two new operations: one for adding a new address to an address book; and another one to look up a name in an address book.

```
pred add(b,b':Book, n: Name, a: Addr){b'.addr = b.addr + n -> a}
fun lookup(b: Book, n: Name): set Addr {(b.addr).n}
```

The first operation is a predicate with four parameters respectively denoting the state of the book before adding the address, the state of this book just after, the name and the address to be added to the book. This predicate is true if it satisfies the constraint composing its body. In this example, the body of the predicate is a constraint that says the address mapping in the new book must be equal to the address mapping in the old book, with the addition of a link relating the new name to the new address. As for the lookup operation, it is written as a function rather than a predicate: its body is an expression, and says that the result of a lookup is whatever set of addresses the name n maps to under the *addr* mapping of b.

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. An assertion is a constraint that is intended to be valid: it must be true for all model instances within some specified scope. For example, the following assertion says that adding an entry for a name n should not affect the lookup result of other names $n\prime \neq n$:

```
assert addLocal {b,b':Book, n,n': Name, a: Addr |
    add(b,b',n,a) && n!=n' => lookup(b,n') = lookup(b',n')}
```

To check if a model satisfies an assertion, 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.

Simple and flexible, Alloy quickly appeared to be a suitable tool to achieve some of our immediate goals concerning the formal verification of aspect interactions in Aspect-UML models.

5 TRANSLATION OF ASPECT-UML MODELS INTO ALLOY

Since aspect interactions are to be analyzed using Alloy, each Aspect-UML model must be translated into an Alloy specification. Aspect-UML semantics has formerly been defined in terms of Petri Nets[4, 12]. The Petri net marking graph of an Aspect-UML model is a simple state machine that can be simulated and analyzed within Alloy. This section explains how the translation to Alloy can be done.

A classic solution. It is worth noting that Alloy has no fixed idiom to describe state machines, although many examples proved the relevance of some modeling approaches for this task. One usual technique consists in describing states as signatures and transitions as predicates (i.e. constraints) over states.

- In Aspect-UML, a system state is given by the set of states of individual objects. Classes (set of objects with attributes) can be modeled by Alloy

signatures (i.e. set of atoms with fields). The state of an object is therefore obtained by consulting field values in the corresponding signature.

- As for transitions, they are described by a predicate (i.e. constraints) relating source states to target states. Two states are related if and only the predicate is true.

This approach would pose no problem if Alloy's signature fields were mutable and could be assigned different values in function of time, like it is the case for attributes in imperative languages. But Alloy's signature fields can only denote a single value within a given model (like variables in mathematics). To simulate some sort of multiple assignment, we simply modify the type of fields, thus forcing them to denote traces of relational values over time. To do this, we append a $Time$ signature to each signature field type. A field f with type $A \rightarrow B$ therefore becomes a dynamic relation of type $A \rightarrow B \rightarrow Time$ matching "A"-values to a "B"-values over time. As explained later, we introduce $Time$ as a signature defining a set of ordered atoms.

An elegant solution. The above modeling strategy might not be the most flexible, neither the most elegant with regard to genericity. Trying to build a generic translation template for advice composition, we encountered one of Alloy's limitation concerning high-order relations. Alloy only allows the definition of flat relations (relations that don't contain relations). This was sort of an inconvenient for us. To model the weaving of two or more Aspect-UML advices at a single join point, we thought of generalizing the composition of two advices by defining a `compose_Adv\3` predicate. This predicate would have been parameterized by two advices (modeled as relations satisfying some given predicate) to be composed, and would have constrained its third parameter to be the sequential composition of the first two. Yet, to avoid defining a predicate such as `compose_Adv\3` for each relational type associated to an advice, we wished `compose_Adv\3` could be defined to accept parameters of any relational type. Of course, there is no such universal relational type within Alloy. On the other hand, if we agree to define a predicate `compose_Adv\3` for each relational type, the analyzer might encountered tractability problems. Considering such a predicate is parameterized three n-ary relations typed over s signatures bounded by m atoms, verifying the satisfiability of this predicate may require the SAT solver to explore up to $((m * s)^n)^3$ states.

Considering these drawbacks, we propose to adopt a modeling approach which will lead us to some elegant and nicely generalized Alloy solution. The Alloy translation model will treat Aspect-UML methods/advices as first class citizens and allow their easy composition by the definition of generalized constraints. In other words, this means that methods/advices will be reified into Alloy constrained signatures, rather than being translated into predicates. The solution we propose is inspired by a case study found in [11] which itself has its source in McCarthy's situation

calculus [13]. The idea consists in translating each method/advice into a signature extending some abstract signature (called `Operation`) denoting the set of all method/advice operations. Each concrete signature extending `Operation` is to provide individual fields representing the method/advice's specific arguments and target object. Pre and post conditions are to be described by facts constraining respective concrete signatures. Advice compositions at join points will be represented by signature `Composition` declared as an extension of `Operation`. General composition rules are to be expressed as facts over the `Composition` signature. These rules shall therefore apply to all advice compositions, no matter what is the type of the advices being composed. Since all advice compositions are atoms grouped under a common `Composition` type, we evacuate the need for a universal relational type within Alloy.

The sequel of this section presents the details of Aspect-UML's translation into Alloy. Examples coming from our telephony case study are used to illustrate how the translation works. In short, the translation proceeds in three main steps. First, structural elements (classes, aspects, data types) of an Aspect-UML model are translated into Alloy. The second step is concerned with the translation of Aspect-UML behavioral specifications (pre and post conditions associated to methods/advices, context passing rules defined by pointcuts). The third step takes up the translation of the weaving process. It is in way concerned with connecting behaviors to structures. It consists of two sub-stages: a) the composition of individual advices executing at the same join point, 2) the effective weaving of advice compositions at respective join points.

Translation of Aspect-UML structural elements

An Aspect-UML model typically gathers a set of structural elements composed of classes, aspects, enumerations and primitive data types. These elements are defining sets either of objects or of data values. Each one can be naturally translated into an Alloy signature. Aspect-UML's primitive type `integer` is however an exception. It is directly mapped onto Alloy's predefined `int` data type. Alloy provides a certain number (although limited) of arithmetic and comparison operators which can be used for the translation of Aspect-UML expressions containing integer values. The translation of the other Aspect-UML structural elements is explained below.

Translation of enumerations and primitive data types. An Aspect-UML enumeration data type E composed of a set of literals $l_1, l_2, ..., l_n$ is translated into an Alloy abstract signature E extended by the respective singleton signatures l_i created for each literal. In our case study, an enumeration type called `Status` is provided to qualify the actual state of a connection. Its Alloy translation is the following signature:

```
abstract sig Status {}
```

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

Aspect-UML's primitive types such as `Booleans`, `Real`, `Date`, etc. must be translated as if they were enumeration types. Function operating over these types must be explicitly translated into Alloy functions. Indeed, Alloy does not provide any other primitive data type than `int`. It does not even have a boolean type (see [11] for explanations). Alloy nevertheless offers a module containing a signature for boolean atoms, as well as a collection of functions simulating boolean operators.

Translation of classes and aspects. From a data perspective, Aspect-UML classes can be seen as sets of objects sharing a common collection of attribute declarations. The same thing can be said about aspects with the little difference that they are singletons. A class C (resp. an aspect A) can reasonably be translated into an Alloy signature C (resp. A) containing a field f_i with type $C \rightarrow F_i$ (resp. $A \rightarrow F_i$) for each attribute $f_i : F_i, 1 \leq i \leq n$ declared in C (resp. A).

Methods/advices are not translated into fields of a class/aspect signature. As briefly explained above, we rather reify them as atoms belonging to an abstract `Operation` signature. Yet, this is not altogether satisfactory. As we know, Alloy is a declarative language used to specify what a system do, rather than how it does it (like imperative languages would do). Intermediate states are not naturally taken into account by Alloy which only provides non mutable fields. We need to overcome this limitation to specify how advices are composed and what are their intermediate effects on object states. To achieve this, each field declaration (denoting a class attribute likely to evolve) is extended with a `Time` signature, as it is shown in the following example.

```
open util/ordering[Time]  as timeorder
sig Time {}

sig Connection {
      status: Status one -> Time,
      origin: Device,
      destination: Device-> Time}
```

Attributes appearing in pre and post conditions of methods/advices are typically identified as the ones likely to evolve. These attributes, translated into signature fields, are ended by the `Time` signature. This is what we observe in the Alloy translation of the `Connection` class above. As specified by pre and post-conditions, connections can change status (e.g., change from disconnected to connected) as well as be transferred to a new destination. Both `status` and `destination` field types are therefore augmented with `Time`. Before the addition of `Time`, `c.status` denoted a single `Status` atom. It now becomes a relation while `c.status.t` denotes the status of connection c at time t. The `Time` dimension allows signature fields to keep

trace of object attribute values at different times of an execution. Type `Time` is simply an ordered set of atoms. The ordering is managed by the parameterized module `ordering` provided by Alloy.

Translation of Aspect-UML behavioral specifications

The behavior of a particular Aspect-UML model is specified by annotations describing the respective pre and post conditions that methods/advices must satisfy. Some behavioral information is also given by the context passing constraints annotating pointcuts.

Translation of advices and methods used as join points. Not all methods/advices declared in an Aspect-UML model need to be translated into Alloy. Methods which do not represent join points can be discarded, as well as advices which do not implement a pointcut. We also assume that method calls appearing in pre and post conditions are discarded and replaced by equivalent expressions containing attribute consultation and predefined functions only. Methods and advices related to a pointcut are the only operations which need to be translated into Alloy since they are the sole ones concerned by our aspect interaction analysis.

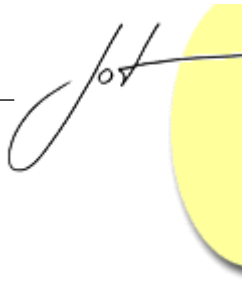
According to our modeling strategy, each method and advice is translated into a signature extending an abstract signature called `Operation` defined as follows:

```
abstract sig Operation { begin, end: Time}
```

The `Operation` signature has two fields respectively used to record the beginning and ending time of an operation. The information is later used to sequentialize operations and to identify the effect of an operation on object states during some time interval. Given a method $myOp(f_1 : t_1, ..., f_n : t_n)$, a concrete signature $myOp$ is created which extends `Operation`. The $myOp$ signature must also declare 1) a `target` field to record the object being invoked and 2) an additional field f_i of type t_i for each of its parameters. An atom from signature $myOp$ represents a specific invocation of method $myOp$.

Being reified into signatures (rather than translated into predicates), methods/advices can be managed uniformly within Alloy. They can be used in field, predicate and function declarations. Some operations can be defined as extensions of others and therefore reuse, for example, the arguments factored out by a parent operation.

Since playing distinct roles in the weaving process, operations representing advices need to be distinguished from methods used as join points. This distinction is realized by the declaration of subtypes `Advice` and `JP`.



```
abstract sig JP, Advice  extends Operation{error: Boolean}
fact indivisible {all op: JP+Advice | op.end = op.begin.next}
```

Although denoting disjoint sets, `JP` and `Advice` both gather operation instances involved in a weaving process which may or not result in a valid composition. To track potential composition errors, `JP` and `Advice` introduces a field `error` that is set to true when such an error is detected. The Alloy model must also assume Aspect-UML advices and join points to be atomic operations having a unitary duration. A fact named `indivisible` is added to specify these behavioral constraints.

Facts applying to abstract signatures `JP`, `Advices` and `Operation` introduce the generic constraints that any join point or advice must respect. The specific behavior of an operation still needs to be specified. Aspect-UML suggests to do this by means of pre and post conditions. Aspect-UML pre and post conditions rely on a subset of OCL's formulas which can be translated into Alloy constraints quite straightforwardly[4]. Our translation is similar to the ones proposed in [14, 7, 6]. One must only be careful when translating access to an attribute. Accessing an attribute a of an object o at a time t requires consulting signature field $o.a$ and navigating through this relation to find $o.a.t$ i.e. the value of the attribute at time t. Taken from our case study, here is an example of a join point specification translated into Alloy.

```
sig Complete  extends JP {self: Connection }
      { (self.status.begin = disconnected)      //precondition
        => (self.status.end = connected) &&     //postconditions
           (self.origin.d_status.end = busy) &&
           (self.destination.end.d_status.end = busy) &&
           (self.origin.current.end = self) &&
           (self.destination.end.current.end = self) &&
           (error = False)
         else (error = True) }  //error
```

The join point described by signature `Complete` implicitly satisfies the generic constraints imposed to all `JP` atoms, while together satisfying the specific pre and post conditions introduced in the `Complete` signature fact. This join point has a single parameter `self` representing the connection targeted by the method call. To achieve a complete operation (line 7), a connection must a priori be disconnected (line 1) otherwise an error is signaled (line 8). At the end of its execution, the complete operation must leave the connection in a `connected` state (line 2), have marked the destination or origin devices as `busy` (lines 3-4), have linked origin and destination devices to the actual connection denoted by `self`(lines 5-6). The time values respectively associated to the before and after states of the connection are compelled to matched the `begin` and `end` field values of the `Complete` operation.

[4]We do not claim nor expect translating the whole OCL into Alloy.

Adding time to signature fields compels us to reason with dynamic domains. Not only do we need to specify what each operation changes in a system (using pre and post conditions), but we must also explicitly specify all the signature fields that it does not modify. This might be tedious if the number of unchanged fields is important for each operation. This matter is known as the *frame problem*. Fortunately, it can be solved without needing to explicitly state all the conditions that must not change with regard to each operation. A more succinct approach consists in adding *explanation closure axioms* [15] to assert which operations are likely to have modified some given field. In the case of Aspect-UML models, it comes to adding facts stating things such as "if a field f has changed, then some advice or join point e happened". Following this approach, a new fact called `unchanged` is added to our translation model, thus solving the frame problem succinctly and modularly.

```
fact unchanged { all t: Time - last | let t'= t.next |
      some e: JP + Advice {
      ((current.t=current.t') || (e in Complete && e.error = False)) &&
      ((status.t=status.t' && d_status.t=d_status.t') ||
        (e in Complete + OpCompleteInterrupting && e.error=False)) &&
      ((interruptedC.t = interruptedC.t') ||
        (e in OpCompleteInterrupting && e.error = False)) &&
      ((destination.t = destination.t') ||
        (e in OpCompleteForwarding && e.error = False))}}
```

Translation of context passing constraints. In Aspect-UML, a pointcut is a sort of abstraction used to expose the context of a certain number of join points under a common interface. An aspect is said to implement a pointcut if it provides an advice of the right type to be executed either before or after one of the pointcut's join points. An advice crosscutting a join point must be able to access the context in which the join point is executed. Pointcuts are used to pass the context of join points (i.e. its actual parameters) to advices in some generic way. In Aspect-UML models, each pointcut is annotated with a set of mapping rules specifying how to relate join points arguments to advice parameters. The Alloy translation of pointcuts therefore consists in creating a predicate $passCtxt(p : JP)$ enforcing the mapping of p's actual parameters over the arguments of each crosscutting advice a.

For example, in the Telecom application, a `passCtxt(p:JP)` predicate is created which, among others, forces context passing from join point `Complete` to crosscutting advices `OpCompleteInterrupting` and `OpCompleteForwarding`.

```
pred passCtxt( p:PJ ) {
   (p in Complete => (Complete.self = OpCompleteInterrupting.c)&&
                     (Complete.self = OpCompleteForwarding.c))
   (p in Drop => ... ) ... }
```

Translation of advice composition and weaving

To model and analyze aspect interactions, the Alloy translation must show how aspects are composed together and how they are woven to the base model. As we did previously for join points and advices, we use abstract signature constrained by facts to specify composition and weaving operations. Alloy's non-recursive functions and flat relations would not have been an appropriate choices to deal with the recursive composition of advices. This approach allows us to define a modular translation which avoid suffering from some of Alloy's limitations.

Composition of advices. According to the aspect paradigm, when a join point is reached during an execution, one or many advices may have been declared to execute at this point. Conflicts can arise if two or more advices require to execute exactly at the same time (i.e. either all before, or all after the join point). Aspect-UML allows developers to define a precedence relation between aspects. This relation can be used to solve out conflicts and therefore justify the sequential composition of two conflicting aspects. If no ordering is imposed, conflicting aspects can execute in any order. They are composed into a non deterministic sequence of advices. Each collection of *before*/*after* conflicting advices will therefore give rise to a composition of advices to be executed respectively before or after corresponding join points. The composition of advices is naturally defined as a recursive operation. Alloy does not allow recursion inside predicates and functions, but it does accept recursive relations. Taking advantage of this, we reify the composition operation into an Alloy abstract signature `Composition` extending the `Operation` signature previously defined. This `Composition` signature introduces two new recursive relations: `comp1` maps the composition on either an advice or composition, while `comp2` relates it to another composition. In short, a composition is either an advice or a pair of compositions.

```
abstract sig Composition  extends Operation{}
    {comp1: Composition + Advice, comp2: lone Composition}
```

As mentioned above, we need to distinguish sequential compositions from non-deterministic sequential ones. This is achieved by defining two sub-signatures `Seq-Composition`, `NDComposition` extending `Composition`. The particular behavior of each kind of compositions is specified by a signature fact that constrains the beginning and ending time of the composition itself and of its nested components. Figure 2 schematizes the internal coordination constraints that each kind of compositions must satisfy.

In Alloy, the two kinds of compositions are straightforwardly modeled by the signatures shown below. These signatures include facts entailing internal coordination constraints. Let's remark the use of predicate `lte` ("less than or equal") to ensure the end of an operation precedes the beginning of the next one. Forcing equality would have allowed no possibility for later interleaving.

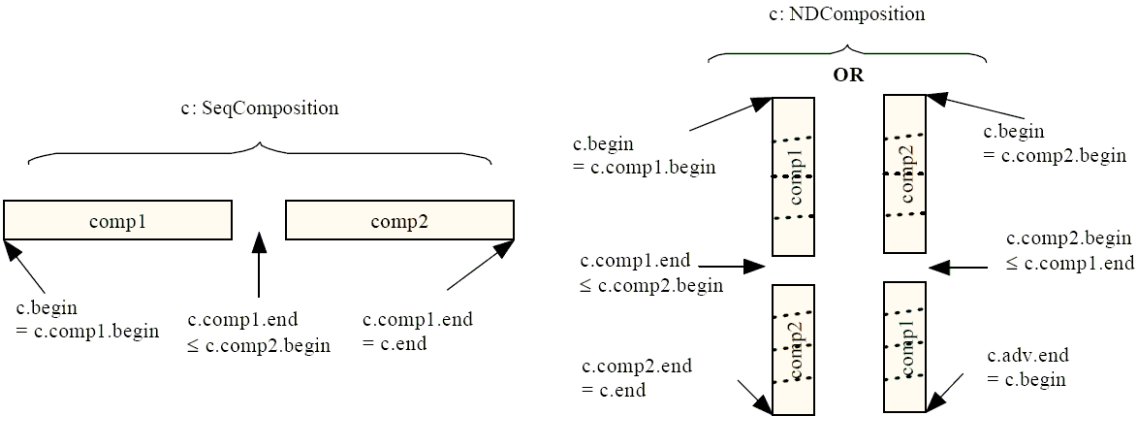


Figure 2: Coordination constraints for `SeqComposition` and `NDComposition`

```
abstract sig SeqComposition extends Composition{}
fact seqComposition {all s: SeqComposition |
      (s.begin= s.comp1.begin) &&
      (no s.comp2 => s.end = s.comp1.end
                  else (s.end = s.comp2.end &&
                      timeorder/lte(s.comp1.end, s.comp2.begin) ) }

abstract sig NDComposition  extends Composition{}
fact nonDeterministicComposition {all nd:NDComposition |
      no nd.comp2 => (nd.begin = nd.comp1.begin && nd.end= nd.comp1.end)
                  else ((nd.begin = nd.comp1.begin &&
                         nd.end = nd.comp2.end &&
                         timeorder/lte(nd.comp1.end, nd.comp2.begin) )
                        ||
                        (nd.begin= nd.comp2.begin &&
                         nd.end = nd.comp1.end &&
                         timeorder/lte(nd.comp2.end = nd.comp1.begin)) ) }
```

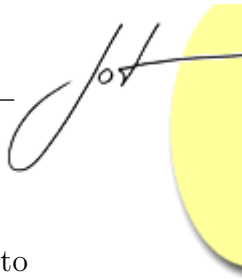
Given those signatures, conflicting sets of advices can now easily be translated into Alloy. Let $Confl^{bef}_{\prec}$ be a set of conflicting *before* advices partially ordered by a precedence relation $\prec$. Let $a_1^{\prec}, a_2^{\prec}, ..., a_n^{\prec}$ be the list of ordered advices[5] from $Confl^{bef}_{\prec}$ such that for each $i = 1..n$, $a_i^{\prec}$ is translated into an Alloy signature ad_i^s extending `Advice`. Similarly, let $a_1^{\not\prec}, a_2^{\not\prec}, ..., a_n^{\not\prec}$ be the list of unordered[6] advices from $Confl^{bef}_{\prec}$ such that for each $j = 1..m$, $a_j^{\not\prec}$ is translated into an Alloy signature ad_j^{nd} extending `Advice`.

The translation of the composition of advices in $Confl^{bef}_{\prec}$ consists of:

1. creating a collection of signatures $seq_1, ...seq_n$ extending `SeqComposition`;

[5] i.e. $a_i^{\prec} \prec a_{i+1}^{\prec}$ for each $i = 1, ..n-1$,

[6] i.e. $\forall j \in \{1...m\}. \forall a \in Confl^{bef}_{\prec}. ((a_j^{\not\prec} \not\prec a)\ \ and\ \ (a \not\prec a_j^{\not\prec}))$



2. creating a collection $nd_1, ...nd_m$ extending `NDComposition`;

3. adding facts to constrain $seq_i, 1 \leq i \leq n$ and $nd_j, 1 \leq j \leq m$ signatures to behave as if composed in a structure having the following shape:

$$nd_1(ad_1^{nd}, nd_2(ad_2^{nd}, ..., nd_n(ad_m^{nd}, seq_1(ad_1^s, seq_2(ad_2^s, ..., seq_n(ad_n^s, \emptyset)...)))...)))$$

For this, each signature seq_i, $i = 1..n-1$ is added the constraint $seq_i.comp1 = ad_i^s$ && $seq_i.comp2 = seq_{(i+1)}$. Since it ends the sequence, signature seq_n is constrained by $seq_n.comp1 = ad_n^s$ && (no $seq_n.comp2$).

Similarly, each signature $nd_j, j = 1..m-1$ is added the constraint $nd_j.comp1 = ad_j^{nd}$ && $nd_j.comp2 = nd_{(j+1)}$. To link the non-deterministic composition of $Confl_{\prec}^{bef}$ to its sequential composition, we simply need to constrain signature nd_m by $nd_m.comp1 = ad_m^{nd}$ && $nd_m.comp2 = seq_1$.

Accordingly, nd_1 denotes the composition of all advices in $Confl_{\prec}^{bef}$. Facts constraining `NDComposition` and `SeqComposition` ensure nd_1 to denote all the possible interleaving of advices in $Confl_{\prec}^{bef}$ with respect to precedence relation $\prec$.

In the Telecom application the signature shown below describes the non deterministic composition of the two advices `OpCompleteInterrupting` and `OpCompleteForwarding`. (We eliminated the empty set at the end of the composition.) It is worth remarking that the sequential composition of these two advices would differ from the one given below (for the non-deterministic sequence) only by removing the name of the extended class `NDComposition` and changing it to `SeqComposition`.

```
sig Compos  extends NDComposition
fact compos {all s:Compos| s.comp1= OpCompleteInterrupting  &&
       s.comp2=OpCompleteForwarding   }
```

Weaving

Once composed, advices need to be weaved at join points. Following the same reification approach, we defined a signature `Weaving` extending `Operation` to describe the weaving of advices at a given join point. The weaving process requires three parameters which we record in the signature's fields: the join point (`jp: one JP`) and the two advice compositions to be woven respectively before and after this join point (`beforeAdvice, afterAdvice : lone Composition`).

```
abstract sig Weaving  extends Operation{}
      {jp: one JP, beforeAdvice, afterAdvice: lone Composition}
```

```
fact mustWeave {all w:Weaving |
       (w.beforeAdvice!=none || w.afterAdvice!=none) }

fact weavingBefore {all w:Weaving |
       w.afterAdvice =none => (w.begin=w.beforeAdvice.begin &&
       w.end=w.jp.end && w.beforeAdvice.end=w.jp.begin)}

fact weavingAfter {all w:Weaving |
       w.beforeAdvice =none => (w.begin=w.jp.begin &&
       w.end=w.afterAdvice.end && w.jp.end=w.afterAdvice.begin) }

fact WeavingBeforeAfter {all w:Weaving |
      (w.beforeAdvice !=none && w.afterAdvice !=none )=>
      (w.begin=w.beforeAdvice.begin &&  w.end=w.afterAdvice.end &&
       w.beforeAdvice.end=w.jp.begin && w.jp.end=w.afterAdvice.begin)}
```

The `Weaving` signature is constrained by fact `mustWeave` to have at least one advice composition to weave (either before or after the join point). The following facts (`weavingBefore, weavingAfter, weavingBeforeAfter`) describe the three possible executions of a weaving: weaving an advice composition 1) before the join point, 2) after, or 3) both. These facts coordinates the sequential execution of the *before* advice composition (if need arises) followed by the join point operation and the *after* advice composition (if need arises). Again, coordination between operations is possible thanks to the `begin` and `end` fields of supertype `Operation`.

Coming back to our case study, the weaving of the advice composition `Compos` at join point `Complete` is defined by the following signature. A fact is added to specify the actual parameters of the weaving operation and the context passing constraints for the join point.

```
sig WeavingBeforeComplete  extends Weaving {}
fact weavingBeforeComplete { all w:WeavingBeforeComplete |
     (w.jp= Complete) && passCtxt(w.jp)
     && (w.beforeAdvice=Compos) && (w.afterAdvice=none)}
```

Each join point in the Aspect-UML model of the Telecom application is to be tackled the same way as join point `Complete`. The translation of each join point implies 1) composing advices reusing abstract signatures `SeqComposition` and `NDComposition`, and 2) weaving advice compositions at the join point reusing the `Weaving` signature.

6 VERIFICATION

Our verification approach, for the detection of aspect interactions, lies within the scope of Model-Based Verification "MBV". MBV aims to provide systematic means

for finding errors in software requirements, designs or code [16]. It relies on the use of mathematical formalism and models, as well as on a disciplined and logical analysis practice. Following the MBV spirit, our verification approach requires the creation of a formal system behavioral model to be analyzed against formal representations of expected properties. Figure 3 shows the verification process of aspect-oriented systems using Aspect-UML as design notation and Alloy as formal analysis tool. The Aspect-UML model to analyze can as well be the result of a forward or of a backward engineering process.

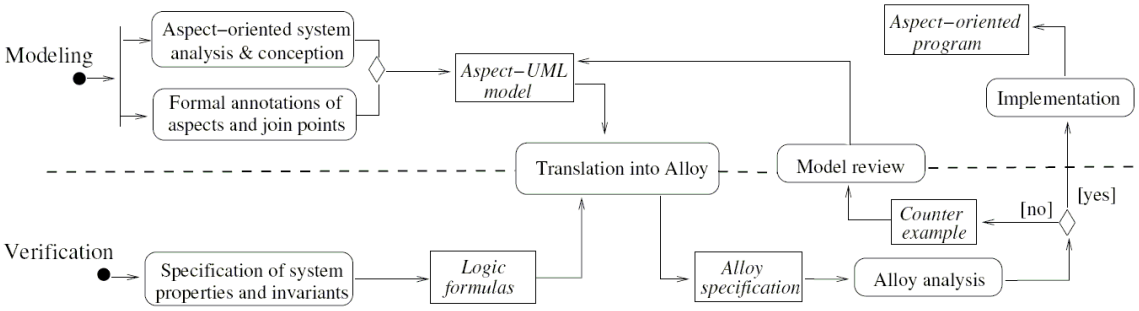


Figure 3: Aspect interaction detection process in Aspect-UML models

Assuming the base system and the aspects are both individually correct, the formal verification process 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 AO systems, therefore capturing the essence of the weaving mechanism, without having to take into account the whole set of system requirements or design decisions. Solely retaining critical parts of a system (join points, advices and weaving specifications), we can focus on the verification of aspect integration within the base system as well as on interactions between aspects. More precisely, this verification aims to reveal important interference problems such as:

1. Violation of local properties. Operations within the system (such as methods/join points and advices) are *locally* specified by means of pre and post conditions. Aspect weaving can cause the violation of the base program properties at join points or the violation of other advices also executing at these join points; an advice specification can also be violated by the base program itself.

2. Violation of global properties. This category includes the violation of a system invariants, following the introduction of new aspects.

Alloy analysis of Aspect-UML models

The Alloy analyzer is a tool, based on algorithms of SAT solvers, for analyzing small model instances. Alloy's model analysis relies on a fundamental premise called

the small scope hypothesis [11]. It states that negative answers tend to occur in small model instances already. Absence of errors in small instances can thus boost one's confidence in the effective correctness of the model. Indeed, 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. When using Alloy analyzer, the recommended strategy consists in starting with a small scope analysis and to increase it gradually until a fault is found or until a satisfying approximation is reached.

Alloy supports two kinds of automatic analysis: simulation and checking. The first one is used to demonstrate the consistency of a given predicate: a state of the model is generated that satisfies this predicate. The second one is used to prove the validity of assertions by attempting to generate a counterexample. All analysis are carried within some user-defined bounded scope that limits 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 drawn about its validity.

Once Aspect-UML models are translated into Alloy, the analysis of aspect interactions is carried by the means of assertions. Assertions are constraints intended to be valid i.e. true for all the models instances. All there is to do is to formulate the desired properties to be verified by the woven (final) system as assertions and to run the Alloy analyzer over each of them. In each case, Alloy will try to find a counterexample invalidating the assertion i.e. proving that at least one instance of the model does not satisfy the set of specified constraints. Local property verification of Aspect-UML models is not carried out the same way as the global property verification. According to the kind of properties (local vs global) being verified, the assertion to be checked will be formulated following one of two patterns, as explained below.

Verification of local properties. At a given join point, it is possible that an aspect advice interferes with the base system or the other aspects by violating their pre/post conditions. Considering the verification of aspect interactions at a given join point, we must formulate an assertion which 1) takes into account all the signatures describing the weaving at this join point; 2) takes into account the initial hypothesis advocating the correctness of the base system and of individual aspects; 3) specifies the final conditions that must be satisfied by the woven system; and 4) ensures that no error has been encountered during the composition and the weaving processes. The pattern used to describe such an assertion is the following:

```
assert localVerifAtJP {all w: WeavingAtJP |
    initialCondAtJP[w] => finalCondAtJP[w] && noErrorAtJP[]}

pred initialCondAtJP(w:WeavingAtJP){//specification of initial hypotheses
     about the base system and about the aspects, before the weaving at JP}
```

```
pred finalCondATJP(w:WeavingAtJP){//specification of final conditions at JP,
     after the weaving}
pred noErrorAtJP(){//indication that no error happened during the
     weaving of advices at JP, ie. operations executed correctly}
```

This pattern indicates that for any weaving operation at a given join point JP, if initial conditions hold before the weaving, then final conditions must hold after the weaving, and the method (join point) together with the advices must also execute correctly. If a counterexample is found by the Alloy analyzer while checking this assertion, this means that the Aspect-UML model has some aspect interaction problem at join point JP.

For example, in the Telecom application, to verify that aspects `Interrupting` and `Forwarding` are interacting correctly at join point `Complete` (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 is given the following assertion to check:

```
assert localVerifAtComplete {all w: WeavingAtComplete |
     initialCondAtComplete[w]=> finalCondAtComplete[w]&& noErrorAtComplete[]}

pred initialCondAtComplete(w:WeavingAtComplete){ let t0=w.begin |
     Complete.self.status.t0= disconnected &&
     OpCompleteInterrupting.c.destination.t0.d_status.t0= busy &&
     OpCompleteForwarding.c.destination.t0.d_status.t0= busy &&
     OpCompleteForwarding.c.destination.t0.num in (Forwarding.forwardList.To)}

pred finalCondAtComplete(w:WeavingAtComplete){ let t=w.end  |
     (Complete.self.status.t= connected)&&
     (Complete.self.origin.d_status.t=busy)&&
     (Complete.self.destination.t.d_status.t=busy)&&
     (Complete.self.origin.current.t= self)&&
     (Complete.self.destination.t.current.t= self)}

pred noErrorAtComplete() {Complete.error= False &&
      OpCompleteForwarding.error= False &&
      OpCompleteInterrupting.error=False}

check localVerifAtComplete for 3 but 6  Operation, 4 Time, 1 Complete,
       1 OpCompleteInterrupting, 1 OpCompleteForwarding
```

When verifying join point `Complete`, we assume the initial hypothesis to be correct, that is to say the base system and the aspects are individually correct at the join point `Complete`. These hypothesis are those imposed by the pre-conditions defined in the Telecom Aspect-UML model (figure 1). Under these assumptions, the Alloy analyzer is asked to check that the final conditions at the join point (which are the post conditions of the join point) hold, and that the operations have been executed without errors.

It is worth recalling that Alloy's assertion checking command requires the definition of a scope bounding the number of elements in each signature. The scope of signatures is sometimes easy to determine. For example, in the above assertion, it is clear that at least six elements `Operation` (defining the join point, the advices, the composition and the weaving) are needed, while at least four `Time` atoms are required to coordinate the woven execution of join point `Complete` with advices `OpCompleteInterrupting` and `OpCompleteForwarding`. Of course, only one element `Complete` is needed to specify the join point. Similarly, only one element from each advice signature is needed.

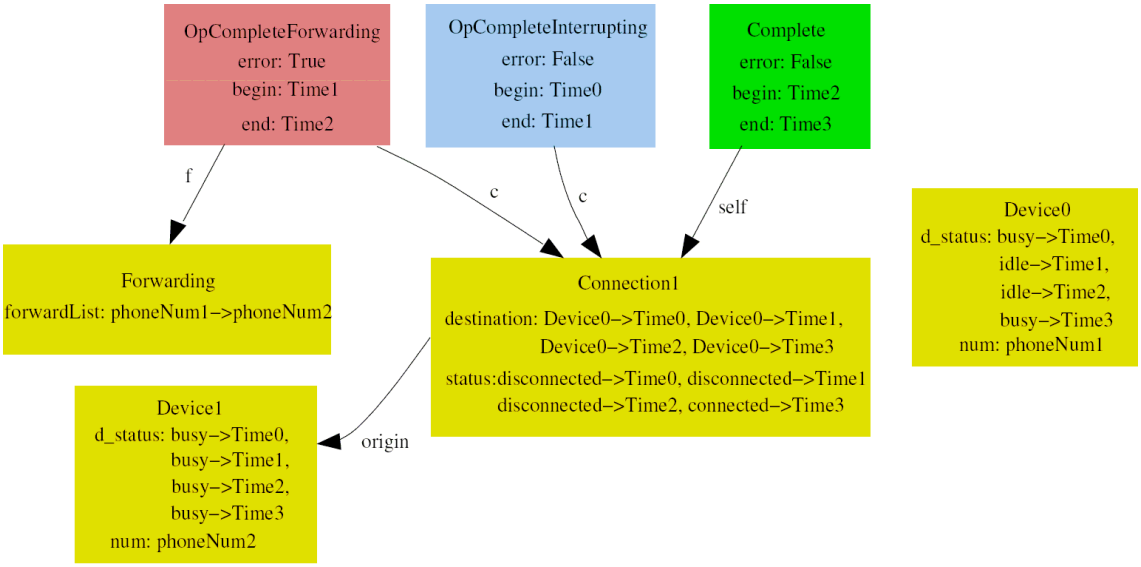
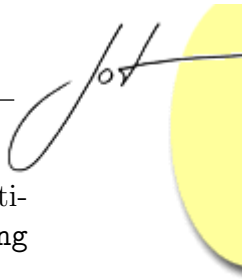


Figure 4:
Counterexample for localVerifAtComplete

When analyzing the above assertion, the Alloy analyzer found a scenario violating the assertion. Alloy returned the counterexample shown by Figure 4 (due to lack of space, only the relevant part of the counterexample is shown here). It clearly shows that an error occurred during the execution of the `OpCompeteForwarding` advice ($error = True$), while the other operations (`OpCompleteInterrupting` and `Complete`) executed correctly. The error could correspond to the following concrete scenario:

- Mrs Apple forwarded her message to work.
- Alice is on the phone while she babysits at Mrs Apple's house.
- Alice answers a second phone call while talking to her friend.
- In the meantime, Peter tries to call Mrs Apple but unexpectedly gets no connection.

Indeed, initially, at state $Time0$, the destination ($Device0$) is busy, and its phone number is in the forwarding list. However, at state $Time2$, the call is not forwarded (the destination is still $Device0$) as we would have expected given the presence of the `OpCompleteForwarding` advice. The reason is that, executed first, the `OpCompleteInterrupting` advice sets the $Device0$ to idle (at time $Time1$),



thus violating the pre-conditions of `OpCompleteForwarding`. Alloy indeed identified a counterexample illustrating a possible conflict between aspects `Forwarding` and `Interrupting`.

Verification of global properties System invariants can easily be added to the Alloy specification of an Aspect-UML model. Invariants are specified using predicates. In the Telecom application, two significant invariants can be expressed:

- Emergency calls are never interrupted.
- For any given connection, the origin and the destination are different.

These invariants can be specified by the following predicates:

```
pred EmergencyNotInterrupted(t:Time){all c:Connection |
       (c.destination.t.num=emergencyNum || c.origin.num=emergencyNum)
          => (c.status.t != interrupted)}

pred OriginDestDiff(t:Time){all c:Connection |(c.origin != c.destination.t)}
```

Considering the verification of system invariants, we must formulate an assertion which, for each invariant and for each join point, 1) takes into account all the signatures describing the weaving at the join point; 2) assumes the invariant to initially hold (in the first state); 3) assumes the join point and the woven advices to execute without errors; 4) ensures that the invariant holds over all the states (mapped to Time) of the weaving. The pattern used to define such an assertion is the following:

```
assert verifInvariant { all w: WeavingAtJP |
     Invariant(w.begin) && noError() => all t: Time | Invariant[t]}

pred Invariant(t:Time){//specifies the invariant to verify}
pred noError(){//indicates that the operations executed correctly}
```

In the Telecom application, the following assertions were given to the Alloy analyzer to verify the `EmergencyNotInterrupted` and `OriginDestDiff` invariants.

```
assert verifEmergencyNotInterrupted{ all w: WeavingAtComplte |
          EmergencyNotInterrupted(w.begin) && noErrorAtComplete()
                     => all t: Time | EmergencyNotInterrupted[t]}

assert verifOriginDestDiff{ all w: WeavingAtComplte |
          OriginDestDiff(w.begin) && noErrorAtComplete()
                     => all t: Time | OriginDestDiff(t) }
```

When these assertions were checked by the Alloy analyzer, counterexamples were actually found. The first counterexample describes a scenario which we can imagine as being the following:

- Bob's call to Alice suddenly interrupts Mary's current connection to the 911.

This counterexample violates the first assertion. Alloy discovers a scenario in which connection ($Connection1$), to be completed, has a destination ($Device0$) already busy. The current connection of device ($Device0$) is ($Connection0$) and its phone number corresponds to an emergency number. So when the current connection ($Connection0$) is interrupted by the execution of `OpCompleteInterrupting`, the violation of the first invariant occurs.

The second counterexample found by Alloy relates a case which we can imagine as follows:

- Bob forwarded his phone calls to his parents for the week-end.
- He forgot to disable the feature when he tried to call Alice once back home.

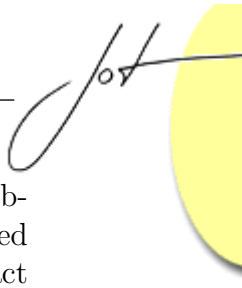
This counterexample violates the second assertion. The invariant is violated following the execution of `OpCompleteForwarding`. Since it has a busy destination ($Device0$), connection ($Connection1$) is forwarded to another destination ($Device1$) which is the connection's origin, yet violating the second invariant.

7 RELATED WORK

Research work related to our project can be resumed under the four following themes.

Aspect-oriented modeling. Considerable research has been done in the area of Aspect Oriented Modeling to clarify and formalize the main concepts of the AO paradigm. Most proposals [17, 18] suggest the use of extension mechanisms of UML for the modeling of AO concepts. Our modeling notation also relies on UML extensions using stereotypes, but yet stands out by its representation of pointcuts. Aspect-UML considers pointcuts as a special kind of interfaces. This conceptual decision proves (1) to provide a better modularization of Aspect-UML models, and (2) to facilitate the identification of aspect interactions in Aspect-UML class diagrams (conflicting aspects implementing a common pointcut interface are easily spotted). Aspect-UML also distinguishes itself from other profiles by providing annotations for the behavioral specification of pointcuts, join points and advices. Although the idea of using semantic annotations in AO models is not totally new [19], it seems that actual proposals for such annotations have not really been further developed, neither have they been used to improve AO model verification.

Aspect interactions analysis based on operational specifications. Once woven, aspect can of course interact with the base program, but may also interfere



with other aspects executing at the same join point. This is a well-known problem which many researchers have tried to solve. To date, most work addressed the problem by tackling the analysis of system source code or some more abstract operational specification of it. Propositions described in [20, 21, 22, 23] all resort to static analysis to identify *observer* aspects within AO programs. Source code is analyzed to detect if an aspect is indeed an *Observer*, that is, checking that it can never modify the state of a system. This kind of syntactic analysis serves proving interference-freedom, but most often remains non conclusive when aspects are not observers. Authors in [24] also present a framework to identify interacting advices from operational specification (rather than source code). Join points are identified by evaluating similarities in the crosscut specifications of aspects. As for previous approaches, conflict detection remains a syntactic matter consisting of identifying shared join points and access to shared variables. Yet, some research is done to improve current static analysis of source code and thus ameliorate both the efficiency and the accuracy of aspect interaction detection. This is the case of authors in [25] applying program slicing techniques to identify the precise part of a program affected by a given aspect. All in all, these approaches remain limited with regards to the partial set of conflicting interactions they can detect and the many false positive they can't reject. As long as the semantics of advices and join points is not considered, semantic conflicts resulting from a violation of individual aspects' properties will not be detected by such approaches. Aspect-UML is actually an attempt to take up this challenge by taking into account the descriptive specification (pre and post conditions) of advices and join points.

Aspect interactions analysis based on descriptive specifications. Descriptive specification try to describe the properties of a system rather than how it actually behaves. It can therefore be used to specify which properties advices and join points are expected to verify for all executions. The semantics of aspects and of base program operations can therefore be considered. Properties are specified by means of mathematical formulas which naturally endows them with a precise syntax and semantics. Formal approaches such as model-checking and theorem proving can be used to verify such specifications. Although less abundantly addressed, the formal verification of aspect-oriented models is yet the subject of quite a few work. Among other, authors in [26] present an approach to verify the properties of systems composed of multiple crosscutting concerns. Concerns are modeled as sets of concurrent processes specified by labeled transition systems. Concerns can be composed by means of merge and override operators. Properties of the composed system are verified using the LTSA model checker. The work in [27] also applies model-checking techniques. This project aims at encouraging the reuse of *generic* aspects to specify collections of aspects which they call *superimpositions*. They also define *validation* aspects denoting initial assumptions about the base program and the *generic* aspects. A prototype tool is presented which uses Bandera [28] to construct a model-checker input from a Java program. Verification annotations are kept separate from the basic program assumption using two different superimpositions.

To verify an extended AO system, the verification process can reuse previous superimpositions, but must run the model-checker over the whole system again. It seems that practical restrictions of the model-checker being used are ignored. On the contrary, our verification approach with Aspect-UML actually intends to avoid such scalability problems by providing some support for modular verification.

Authors in [29] use model-checking to modularly verify aspect advices. Invariants, assumed to hold over the original base system, are checked to also hold over aspects added to the system. Verifying a newly added aspect does not require to run the verification over the entire system. The solution proposed however seems a little simplistic since it neglects considering the possible side effects (e.g., disabling a functionality) that overriding aspects may have on the basic system. In [30], authors suggest an assume-guarantee structure to achieve modular and generic verification of AO systems. The proposed prototype verifies that for any base state machine satisfying the assumptions of a given aspect, the woven state machine is guaranteed to satisfy the desired properties. A single generic state machine is constructed from an aspect's assumption (i.e. the pointcut descriptor and the advice state machine) and is verified for the desired property. Then, when a particular base program is to be woven with the aspect, it suffices to establish that its base state machine satisfies the assumption. Concerned about verifying the reusability of aspects over multiple base programs, this work unfortunately does not deal with the problem of conflicting aspects. Yet, it is a major issue that Aspect-UML addresses by formalizing not only the weaving process but also the composition of conflicting advices at join points.

Translation of UML-like models into Alloy. Alloy has been used for quite a few verification projects. Among them, some were concerned with verifying UML models augmented with OCL annotations. In [7], authors propose a translation from UML/OCL to Alloy. The translation deals with class diagram constrained by OCL invariants. Unfortunately, nothing is said about the translation of more subtle dynamic properties, such as pre and post conditions, which require relating states with time. Based on MDA, the transformation tool presented in [6] goes a step further. The tool does indeed translate OCL methods' pre and post conditions into Alloy constraints. But still, it does not explain how to specify their sequential or parallel execution in Alloy. The task of defining MDA transformation for behavioral aspects of systems is put off with future work.

Alloy was also used in [31] to verify if a given invariant is satisfied before and after each model transformation, thus describing aspect weaving as role merging. Again, this approach is limited to invariant verification, and does not address the translation of origin models into Alloy. Compared to existing approaches (and other short examples found in the literature), our translation of Aspect-UML models into Alloy offers an appreciable amount of insights into how to proceed. It provides a systematic approach to object/aspect oriented model translation into Alloy. Moreover, it covers the translation of both structural and behavioral properties of objects as well as the translation of their dynamics. This task was a key step in achieving the formal

verification of Aspect-UML interactions using Alloy.

8 CONCLUSION

This paper showed how Aspect-UML models are translated into Alloy to further be formally verified. The verification aims at revealing possible semantic conflict that may occur between two aspects or between an aspect and the base system. Alloy analyzer is somehow used to solve the set of constraints resulting from the composition and the weaving of advices at join points in Aspect-UML models. Of course, one may fear the use of automatic verification tools given the well-known state explosion problem. We intend to evolve our verification approach with scalability in mind and with solution to cope with state explosion. By limiting the verification to the critical "weaving" parts of the system, our approach already confines the verification to a subset of the system states. Compositional verification and proof reuse are also being considered to reduce larger 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 may still be obtained even for very large systems.

REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, Ch. Maeda, Ch. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97, LNCS*, pages 220–242, 1997.

[2] G. Leavens and C. Clifton. eds.: Foundations of aspect-oriented languagesworkshop. In *In Leavens, G., Clifton, C., eds.: AOSD05. Volume 4*, 2005.

[3] J. Vachon and F. Mostefaoui. Achieving supplementary requirements using aspect-oriented development. In *ICEIS*, pages 584–587, 2004.

[4] 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.

[5] D. Jackson. Alloy: A leightweight object modelling notation. In *MIT Laboratory for Computer Science: Cambridge, MA*, 2000.

[6] B. Bordbar and K. Anastasakis. UML2ALLOY: A tool for light-weight modelling of discrete event systems. In *IADIS Applied Computing*, 2005.

[7] T. Massoni, R. Gheyi, and P. Borba. A UML class diagram analyzer. In *3rd Workshop on Critical Systems Development with UML (CSDUML)*, 2004.

[8] Object Management Group. UML: Superstructure. version 2.0, August 2005.

[9] Alloy Homepage. http://alloy.mit.edu.

[10] J.M. Spivey. *The Z Notation*. Prentice-Hall, 1992.

[11] D. Jackson. *Software Abstractions Logic, Language and Analysis*. MIT Press, 2006.

[12] F Mostefaoui and J. vachon. Formalization of an aspect-oriented modeling approach, (published online). In *Formal Methods 2006*, August 2006.

[13] J. McCarthy. Situations, actions and causal laws. *Semantic Information Processing*, pages 410–417, 1968.

[14] M. Vaziri and D. Jackson. Some shortcomings of OCL, the Object Constraint Language of UML. Response to OMGs Request for Information on UML 2.0", 1999.

[15] A. Borgida, Mylopoulos J, and R. Reiter. On the frame problem in procedure specifications. *Software Engineering*, 21(10):785–798, 1995.

[16] 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.

[17] 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.

[18] D. Stein, S. Hanenberg, and R. Unland. A UML-based aspect-oriented design notation for aspectj. In *AOSD Conference*, pages 106–112. ACM, 2002.

[19] J.M. 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.

[20] C. Clifton. *A design discipline and language features for modular reasoning in aspect-oriented programs*. PhD thesis, Iowa State University, IA, USA, 2005.

[21] D. Sereni and O. de Moor. Static analysis of aspects. In *spect-Oriented Software Development AOSD'03*, pages 30–39, 2003.

[22] M. Storzer and J. Krinke. Interference analysis for aspectj. In *FOAL*, 2003.

[23] M Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *FSE Conference*, 2004.

[24] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse and interaction analysis of stateful aspects. In 3^{rd} *Int. Conf. AOSD*, pages 141–150, 2004.

[25] D. Balzarotti, A. Castaldo, and M. Monga. Slicing aspectj woven code. In *FOAL*, 2005.

[26] T. Nelson, D. Cowan, and P. Alencar. Supporting formal verification of crosscutting concerns. In *REFLECTION, LNCS, vol.2192*, pages 153–169, 2001.

[27] S. Katz and M Sihman. Aspect validation using model checking. In *International Symposium on Verification in honor of Zohar Manna*, 2003.

[28] J.C. Corbett, M.B. Dwyer, J. Hatcliff, and Robby. Bandera: a source-level interface for model checking java programs. In *ICSE*, 2000.

[29] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT'04/FSE-12*, 2004.

[30] M. Goldman and S. Katz. Modular generic verification of LTL properties for aspects. In *FOAL, held with AOSD Conference*, 2006.

[31] S. Nakajima and T. Tamai. Lightweight formal analysis of aspect-oriented modelss. In *Workshop on Aspect-Oriented Modeling at UML'04*, 2004.

Farida Mostefaoui is a PhD candidate at University of Montreal (DIRO), Quebec, Canada. She is a member of the GEODES Laboratory, pursuing a Ph.D. thesis on an aspect oriented approach for the detection and the verification of aspect interactions. She can be reached at mostefaf@iro.umontreal.ca.

Julie Vachon works as an assistant professor at the Departement of Computer Science of University of Montreal in the software engineering lab GEODES. She studied computer science in Montreal, Lyon and Lausanne where she obtained, in 2000, a Ph.D. degree from the Swiss Federal Institute of Technology. Her research interests include aspect-oriented modeling, incremental design of safe and fault-tolerant systems, formal specification and verification. She can be reached at vachon@iro.umontreal.ca.