Testing for Unexpected Interactions in AOP

André Restivo
Artificial Intelligence and Computer Science Laboratory
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
arestivo@fe.up.pt

Ademar Aguiar
INESC Porto
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
aaguiar@fe.up.pt

Abstract—Aspect Oriented Programming (AOP) is a powerful programming technique with the objective of improving modularity by encapsulating crosscutting concerns. The nature of AOP makes it prone to unexpected and harmful interactions between the different components of a system.

The claim behind this PhD is that unit tests can be used to detect these interactions.

In this paper we explain how these can be accomplished. A brief state of the art, work plan and a support tool (drUID) are also presented.

I. INTRODUCTION

Aspect Oriented Programming (AOP) is a programming technique that allows crosscutting concerns to be separated into their own units of modularity.

This improved modularization is usually achieved by allowing developers to identify points in the control flow of the program, called joinpoints, where pieces of code, called advices, can be applied. Joinpoints can also be grouped together into something that is usually called a pointcut. An aspect is then composed of pointcuts and advices in such a way that concerns that usually crosscut several modules of the application can be kept in a single place.

AOP has already proven to be an important step towards better code modularization. However, by allowing developers to change the control flow of a program, it increases the chances of unexpected interactions between different modules occurring.

For example, imagine an application where messages are exchanged between a server and several clients. This could be easily done by using a classical Object Oriented approach by creating a server class and a client class. These classes would know how to communicate between them by using some other TCP class.

Suppose that we want to use AOP in order to cypher the messages between the server and the client. This could be easily done by targeting the joinpoints where each class sends their messages and cypher the messages before they are sent. We would then only need to do the same thing in the receiver side and target the joinpoints where the messages are received and decrypted. This could be done in a single aspect thus keeping the cypher concern in a single unit of modularity. If everything goes as planned, the main concern of the application, sending and receiving messages, would interact with the cypher concern in a correct an expected way.

However, if another aspect that removed curse words from the text being sent was added to the system, it could easily behave in an unexpected way. This would happen if the cypher concern was applied before the remove curses concern as it would impede it from having access to the plain text version of the message being sent.

In this example, we have two aspects that when applied individually to the base code work correctly but interact with each other in and unexpected and incorrect way.

It is important to notice, that in this case, simply changing the weaving order of the two aspects would solve the problem but this is not the general case. Sometimes interactions are far more complicated to solve.

We claim that using a carefully planned testing methodology would allow to detect at least some of these unexpected interactions.

II. RELATED WORK

Several researchers have been working to solve this same problem. In this section we present some of the other studies in this field that we consider to be the most relevant. We separated the techniques into automatic analysis techniques and user controlled analysis techniques. A more complete review of the state of the art can be found in [8].

A. Automatic Interactions Detection

Balzarotti [2] claims that the interaction detection problem can be solved by using a technique proposed in the early 80s, called program slicing. Although totally automatic, this technique does not account for intended interactions.

Havinga [3] proposed a method based on modeling programs as graphs and aspect introductions as graph transformation rules. Using these two models it is then possible to detect interactions caused by aspect introductions. Both graphs, representing programs, and transformation rules, representing introductions, can be automatically generated from source code. Although interesting, this approach suffers the same problem of other automatic approaches to this problem as intentional interferences cannot be differentiated from unintentional ones.

B. User Controlled Interaction Detection

Lagaisse [6] proposed an extension to the Design by Contract (DbC) paradigm by allowing aspects to define what they expect of the system and how they will change it. This will
allow the detection of interferences by other aspects that were woven before, as well as the detection of interferences by aspects that are bounded to be woven later in the process.

Katz [4] proposed the use of regression testing and regression verification as tools that could help identifying harmful aspects. The idea behind this technique is to use regression testing as normally and then weave each aspect into the system and rerun all regression tests to see if they still pass. If an error is found, either the error is corrected or the failing tests have to be replaced by new ones specific for that particular aspect.

It has been noticed by Kienzle [5] that aspects can be defined as entities that require services from a system, provide new services to that same system and removes others. If there is some way of explicitly describing what services are required by each aspect it would be possible to detect interferences (for example, an aspect that removes a service needed by another aspect) and to choose better weaving orders.

III. Research Objectives and Approach

Unexpected interactions are one of the main issues preventing AOP from becoming a mainstream development approach. Obliviousness, the capability of developing aspects without having to reason about other concerns of the problem at the same time, has been tagged as the culprit of this problem.

Several ways of tackling this issue have been proposed in literature but none has proven sufficiently effective or even widely accepted by the community.

Having in mind the difficulties of achieving modularity with AOP, the proposed work has the following objectives:

1) To describe, formally, the different types of interactions in order to better understand the scope of the problem.

2) To develop a test driven approach to allow the specification of incompatibilities and dependencies between aspects.

3) To develop a supporting tool, integrated into an existing development platform, to aid in the development of aspects by ensuring compatibility between them.

All these objectives aim to prosecute a more fundamental goal that is to improve the usability of AOP languages making obliviousness, when desired, a reality.

IV. Current Work and Preliminary Results

In this section we will present the current state of our work. We will start by explaining the concept of dependency graphs in this context. Then we will show how these graphs can be used to compile programs incrementally in order to allow the detection of interactions. We will conclude by presenting a support tool that implements the technique presented.

A. Dependency Graphs

Each module that composes an application has a set of other modules that it depends upon. These dependencies can be of different forms. For example, a module that calls a function from another module has a dependency relationship with that module as it will not work if the module, where the called method is implemented, is not present in the system.

We can obtain the dependency graph of an application manually or by using some kind of code inspection technique. Figure 1 contains the dependency graph for the example introduced earlier.

This graph can be used to choose an order in which the system can be composed module by module. This incremental composition allows us to test the system step by step while searching for interactions between modules.

To obtain this ordering, we first use a strongly connected components (SCC) algorithm to find cycles in the graph. These cycles will be treated as single modules. And then we do a topological sort (TS) to compute one possible composition order.

In the next section we will show how this graph can be used to detect interactions between modules.

B. Testing for Interactions

With the dependency graph sorted, we start by choosing the module that does not depend on any other modules, compile it and run unit tests against it. In the example that we have been using this would be the TCP module.

If the module is correct, the unit tests will all pass. We then proceed by choosing one of the modules that depend only on modules already tested. In this case there is a circular dependency between the Server and Client modules. This dependency would be detected by the SCC algorithm and both modules would be treated as one.

This means that the next step is to compile the TCP, Server and Client modules together into a single application. We then proceed by running tests against the modules composed in previous steps. If one of these tests fail, it means that there is an interaction between the modules compiled previously and the ones just added to the system. If all tests pass then we run the tests against the new modules to assure they are working properly.

This process continues by choosing either the Remove Curses module or the Cypher as both depend only on modules already tested. Assuming we choose to compile the Remove Curses module first and after applying the same process all
tests are successful we are left with only the Cypher module. By applying the same process to this last module, we would find that the tests for the Remove Curses module would fail. This means that an interaction between these 2 modules exist as the module worked before the Cypher had been added.

We can summarize our claim by using a more generic example. If we have a program $P$ composed by modules $M1$ through $Mn$, and we take a subset $P'$ of $P$ where all dependencies are satisfied and tests for all the modules contained in $P'$ pass, and if we take a module $Mi$ from $P - P'$ and add it to $P'$ creating a new subset $P''$, we can get three different scenarios:

- All tests on $P''$ pass. This means there are no interactions between the modules in $P'$ and module $Mi$.
- All tests on $P'$ pass but some tests for module $Mi$ fail. This means that there is an interaction between module $Mi$ and one of the modules contained in $P'$.
- Some tests on $P'$ fail. These means that there is an interaction between the module of $P'$, whose test failed, and module $Mi$.

This of course assumes that the tests are well designed and have sufficient code coverage to detect the fault introduced by the interaction.

C. Unexpected Interactions

Not all interactions detected by the process just proposed are unexpected or even harmful. Invasive aspects can change the behavior or flow of a program in an intended way. After all, they have a specific purpose and most of the times that purpose is exactly to change how the code works.

After an expected interaction has been detected, the developer could simply change the unit tests of the modules whose behavior was changed by aspects in other modules. This would not be a good solution as it would break modularity and prevent the reuse of modules.

In our approach, we expect developers to specify which behaviors they expect to break with their aspects. This can be done before the detection of an interaction, preventing false positives, or after, preventing the interaction from being detected the next time around. Besides stating which behaviors are broken, developers can add tests for the new behavior of the altered modules.

In order to improve the granularity of the process, dependencies can be stated at the feature level. Each module exports several features (i.e. the behaviors of the module as seen by the user or by other modules through an API) and each one of them can depend on one or more features from other modules. This still allows us to calculate the dependency graph and, at the same time, improves the feedback obtained by the process. Other advantage of this approach is that it makes it easier to insert information about the dependencies inside the source code of the application making the whole process easier to use.

D. Tool Support

drUID [7] is an Eclipse plug-in that helps AspectJ developers to detect interactions between aspects using the technique that has been presented in this paper. In order to accomplish this, the plug-in allows developers to define several characteristics about system artifacts using Java annotations:

- Which features are provided by each artifact.
- Which JUnit tests are used to test each feature.
- Which features depend on other features.

Several aids have been implemented to guide the developer in this process in the form of Eclipse quick fixes and quick assists. Each time a file is saved in Eclipse, the annotations are inspected and any errors are reported. Besides that, a dependency graph is created and shown in a graphical form that allows the developer to navigate through the code following the dependencies between artifacts.

With the dependency graph created, the developer can order drUID to execute an interaction test analysis. This analysis executes the following steps (Figure 2):

1) Execute a Strongly Connected Component

2) Execute a Topological Sort

3) For each Component:

   a) Compile Current and Previous Components

   b) Test Previous Components

   c) Test Current Component

   Error in Current Component

   No Interactions

Fig. 2: Interaction Detection Process
When the later happens, an interaction has been detected and the analysis stops. This interaction is reported to the developer with an explanation of which feature has been broken by which component. This will allow the developer to understand what went wrong and act accordingly. Two different conclusions can be extrapolated by the output generated by the tool: (i) the interaction is expected (ii) the interaction reveals a unexpected interaction between aspects.

In the first case, the developer simply has to declare the interaction by using, once again, annotations. By using Eclipse quick fixes added by the plug-in this can be done easily by the developer.

In the later case, we are confronted with an unexpected interaction that has to be dealt with in another way. If possible the developer can change the implementation of one of the aspects in order to remove the interaction. But in some cases, the interaction might have been caused by two incompatible requirements and, in that case, it can only be corrected by changing the software requirements themselves.

E. Evaluation

There are several aspects of the technique that need to be validated:

1) Is the technique applicable to all possible setups?
2) Does the technique detect all kinds of interactions or are there some that simply don’t fit into our model?
3) Is it feasible to use the technique in a real world scenario or is the added workload too much of a burden for developers?

We expect to validate these points using three different approaches:

1) Creating a catalog of different interactions, with code examples, and applying the technique to them. This will allow a certain degree of confidence about the universality of the technique.
2) Creating a mathematical model of the claim and formally proving that it works for all possible cases. Or, if we end up discovering it does not, explaining which interactions it fails to detect.
3) Using the technique in real world, or almost real world, scenarios and annotating the overhead introduced by the extra work needed to implement the technique.

V. WORK PLAN AND IMPLICATIONS

The plan devised for this research work encompasses seven main phases and will make use of diverse research methods and experimental approaches. These phases are:

1) State of the Art Review - This task aims at gathering information about the several topics relevant for the considered problem, namely: Aspect-Oriented Programming, Formal Methods (in particular Design by Contract) and Regression Testing (in particular Unit Testing).
2) Exploratory Projects Development - The development of one or several small projects using AOP in order to better understand the interference problems existing in the field and how they can be tackled.
3) Hypothesis and Problem Definition - The precise definition of the research questions and the formulation of a hypothesis will be the main result of this task. Furthermore, it comprises the definition of the parameters that will be used for validating and assessing the approach.
4) Conceptual Framework Development - Development of the conceptual framework that will allow the detection of interferences and conflicts between aspects.
5) Development of Supporting Tools - Development of a supporting tool, integrated into an existing development platform, to aid developing aspects using formal methods or regression testing to ensure compatibility between them.
6) Result Consolidation - This phase starts after the definition of validation criteria in the scope of the Hypothesis Definition, and gains momentum after a first version of the approach can be validated experimentally.
7) Writing of the Thesis - This task will accompany the considered phases for the research work as the respective milestones contribute for its completion.
8) Participation in Conferences - As the thesis progresses the parcial results will be presented in conferences related to the AOP subject. Papers have already been presented in the SPLAT’07 [8] and LATE’08 [9] workshops. These workshops were part of the AOSD conference (the most important AOP conference). Papers were also presented in the CoMIC’06 [10] and CoMIC’07 [11] Doctoral Symposiums.

VI. CONCLUSIONS

We presented a technique that allows the detection of interactions between aspects using unit tests. We showed how the technique can be used by developers to better understand how their code interacts with other aspects in the system and also what they can do to correct eventual unexpected interactions. We also showed how annotations can be used to signal false positives and improve the overall performance of the technique.

We recognize that this technique does not detect all possible interactions introduced by AOP languages. The quality of its response will always be tied to the quality of code and tests. Even quality code with good tests can sometime have interactions that are not detected by the technique if the changes introduced by the aspects are of such an unexpected nature that they are not accounted for in the unit tests.

Currently we are working on evaluating the technique but there are still several improvements to be done:

1) Automatically detect dependencies using code analysis. Some dependencies should be impossible to find but detecting most of them would provide a faster and more usable process.
2) Improve the Eclipse plug-in.
3) Investigate a possible relation with Feature Driven Development.
ACKNOWLEDGMENT

We will like to thank Ana Moreira, Cristina Videira Lopes, and Miguel Pessoa Monteiro for their help on constantly broadening our perspectives regarding interactions due to aspect composition.

A word of appreciation goes also to the reviewers that took the time to fully understand the work and helped improve the paper with their insightful comments.

We will also like to thank FCT for the support provided through scholarship SFRH/BD/32730/2006.

REFERENCES


[7] A. Restivo and A. Aguiar, DrUID : Unexpected Interaction Detection, Demo at the Aspect Oriented Software Development Conference (AOSD) 2009, Charlottesville, VA, USA.


