Interaction Analysis for Fault-Tolerance in Aspect-Oriented Programming*

Nathan Weston, Francois Taiani, Awais Rashid

Computing Department, InfoLab21, Lancaster University, UK.
{westonn,f.taiani,marash}@comp.lancs.ac.uk

Abstract. The key contribution of Aspect-Oriented Programming (AOP) is the encapsulation of crosscutting concerns in aspects, which facilities modular reasoning. However, common methods of introducing aspects into the system, incorporating features such as implicit control-flow, mean that the ability to discover interactions between aspects can be compromised. This has profound implications for developers working on fault-tolerant systems. We present an analysis for aspects which can reveal these interactions, thus providing insight into positioning of error detection mechanisms and outlining candidate containment units. We also present AIDA, an implementation of this analysis for the AspectJ language.

1 Introduction and Problem Statement

The key contribution of Aspect-Oriented Programming (AOP) is the encapsulation of crosscutting concerns in *aspects*, which are then introduced into the system using *advice* - code which applies at a particular *joinpoint* in the system, be that in the base program or within aspect advice. Advice is then *woven* into the system, either statically at compile-time or dynamically later on. This crucial feature of AOP supports modularity and evolvability of otherwise scattered and tangled code, as well as offering the possibility of aspect reuse.

However, it also raises a potential difficulty for developers working with aspects in a fault-tolerant context. For example, the usual method of implementing aspects - such as the popular AspectJ[1] compiler - allows a developer to code a piece of advice which applies implicitly at multiple points in the code. This can cause problems in determining how faults might propagate through the system, as it is not immediately clear from the code how advice code interacts with the base system, and especially how aspects interact with one another.

To see this, let us consider the example of a version control system which includes the following code:

* This work is supported by European Commission Framework 6 Grant: AOSD-Europe: European Network of Excellence on Aspect-Oriented Software Development (IST-2-004349).

```
void commitChanges(String username, String server) {
    ...
    sendFile(username, file , server );
    ...
}
```

The system has two aspects - one which logs all calls to the `sendFile()` method, and one which encrypts usernames at calls to the `commitChanges()` method:

```
void aspect LogFileSends {
    after(): call(void sendFile(String, String, String)) && args(uname, file, serv) {
        printToFile("Sent to server, sender is "+uname);
    }
}

void aspect Encrypt {
    around(): call(void commitChanges(String, String)) ∧ args(uname, serv) {
        encrypt(uname);
        proceed(uname, serv);
    }
}
```

Although these two advices apply at different joinpoints in the system, they have an indirect interaction with one another - the `Encrypt` aspect modifies the `username` variable, which the `LogFileSends` aspect reads. Therefore there is a potential coupling between the two advices. Knowing this could impact the strategy for making this system fault-tolerant - for example, if the `LogFileSends` aspect is considered particularly crucial to the system, this might require the `Encrypt` aspect to be hardened with additional fault-detection mechanisms.

As well as this *indirect* interaction, we must also consider the possibility of *transitive* interactions between aspects. By contrast to indirect interactions, which are based on shared accesses of a single variable (for example, the `uname` variable in the above example), transitive interactions occur when a chain of variable accesses link advices. That is, consider the system in Fig. 1. The system has an aspect A which modifies a variable x. The variable is then passed to a method f, which uses x to define a variable y. Subsequently, the value of y is used in aspect B to determine its behaviour. Hence there is a transitive interaction between the two aspects, even though they do not access a shared variable.

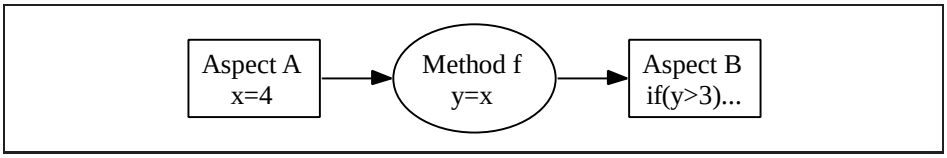

Fig. 1: Transitive interaction

In general, the ability to identify and trace these interactions can help developers understand how error might propagate, and thus decide where to position error detection mechanisms (such as assertions or acceptance tests) in order to

maximise error detection coverage while minimising overhead. It can also provide insight into error handling by identifying parts of the system which might simultaneously become corrupted, thus outlining candidate containment units.

This kind of interaction analysis also has applications should AO techniques be used to implement fault-tolerance itself. As has been noted, AOP can be an extremely helpful tool to aid developers in building fault-tolerant systems. For example, transaction mechanisms can be implemented as aspects in order to perform rollback in the event of failure[5]; similarly, the mechanisms for switching between versions in N-version programming can be encapsulated in an aspect. Contract enforcement[3] can also be modularised in this way. In this formulation, then, the interactions between fault-tolerance aspects and others can shed light on potential problems which could cause the system to be intolerant to faults - for example, if it can be seen that the presence of another aspect causes a contract enforcement aspect to be bypassed.

In this article we investigate how Data-Flow Analysis can be adapted to be applied to aspect-oriented programs in order to help developers with this problem. We also present AIDA, an implementation of our analysis for use with the AspectJ language. Section 2 gives some background in Data-Flow Analysis, and we discuss possible modifications with respect to AO programs in Section 3. Section 4 presents AIDA, and Section 5 concludes and looks to potential future work.

2 Data-Flow Analysis

Data-Flow Analysis (DFA)[6] is an ideal tool in determining this kind of indirect interference between aspects. DFA gives us the ability to see which data aspect advice modifies, and (crucially) trace the effects of that modification throughout the program, including its effect on other aspects. In this section we present the basic tenets of DFA which are necessary in order to understand our approach.

The classical *Definition-Use* analysis forms the basis of our approach. The idea behind this analysis is to find *Definition-Use chains* or *du-chains*, associations between an assignment to a variable and all its uses in a program. The def-use analysis is based on a classical data-flow analysis called Reaching Definitions Analysis[6]. Given a program point, the analysis returns the definitions which may have been made and not re-defined when the execution of the program reaches that point. Comparing these definitions with uses of variables at the program point enables us to determine du-chains, which are candidates for error propagation paths. Our approach to performing this inter-procedural analysis is an extension to the functional approach proposed by Shahir and Pnueli[7], which has an acceptable tradeoff between efficiency and accuracy.

The approach operates on an Inter-procedural Control Flow Graph (ICFG), which contains a control-flow graph (representing code statements as nodes and potential flow as edges) for each of the methods and aspect advices in the program. From this, an intra-procedural analysis computes a *transfer function* for each program point within a method, which represents the effect of the Reach-

ing Definitions Analysis up to that point. As this happens within the method, it models which definitions reach the program point based on abstract initial values at the start of the method. For example, in Fig. 2, the transfer function at program point n_2 in method A is $\rho_2 = f_1$.

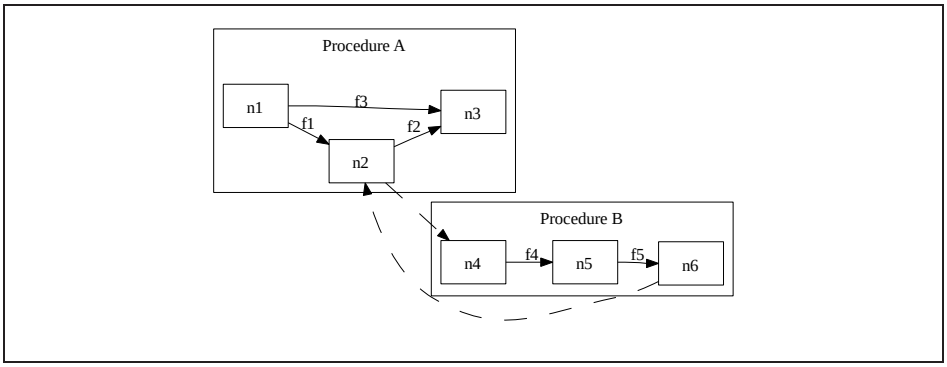

Fig. 2: Computing transfer functions

In order to model method calls, a special instance of a transfer function is created which defines the summary of calling a method - effectively the conjunction of the transfer functions at the exit points of each method. In Fig. 2, the summary transfer function ϕ_B modelling the effect of calling method B is $\phi_B = \rho_6 = f_5 \circ f_4$. This information is propagated bottom-up using a fixpoint calculation to determine the transfer functions for each method, taking calls into account. For example, the final transfer function at node n_3 is:

$$\begin{aligned}\rho_3 &= f_3 \wedge (f_2 \circ \phi_B \circ f_1) = f_3 \wedge (f_2 \circ \rho_6 \circ f_1) \\ &= f_3 \wedge (f_2 \circ f_5 \circ f_4 \circ f_1)\end{aligned}$$

The next step is to propagate real data-flow information in a top-down fashion, starting from `main()` methods with an empty set of reaching definitions. At this stage, information is only propagated to entry points of procedures and to call sites, which is possible because transfer functions based on abstract initial values have already been computed at these points. Again a fixpoint calculation is used to resolve any circular dependencies. In the above example, the real solution S_4 at node n_4 is $S_4 = \rho_4(\rho_2(\eta))$, where η is the data-flow information present at the beginning of procedure A.

Once this is done, it is trivial to calculate the results of the reaching definitions analysis on-demand, as both the concrete data-flow information at the beginning of each procedure and transfer functions for each program point within that procedure are available. Therefore, the analysis result at node n_5 is the result of the function $\rho_5(S_4)$, both elements of which have previously computed.

3 Transfer functions for advice

One consideration in applying this technique to AO programs is that of computing transfer functions for advice. In languages such as AspectJ, advice which applies *around* a joinpoint can be difficult to reason about independently of the base system, mainly due to the inability to discover what a `proceed()` statement could refer two. We present two main options:

Advices as methods Perhaps the simplest option is to perform the analysis after the aspect advice has been woven, and treat aspect advices identically to method calls. An *around* advice's `proceed()` statement would therefore be transformed to another call back to the advised procedure, and computing the transfer functions would proceed as normal.

Binding functions The disadvantage of the first option is that, as we perform the analysis after the advice has been woven, it is difficult to consider the effect of the advice independently of its binding. Therefore, there could be no partial analysis results associated with library aspects which include *around* advice, as it would be impossible to know the effect of a `proceed()` statement before weaving. One solution is to transform the *around* advice into $before$ and $after$ advice and treat it as two separate advices - however, real-world advices may well have multiple `proceed()` statements or have control-flow paths which bypass the `proceed()` instruction altogether (see Fig. 3).

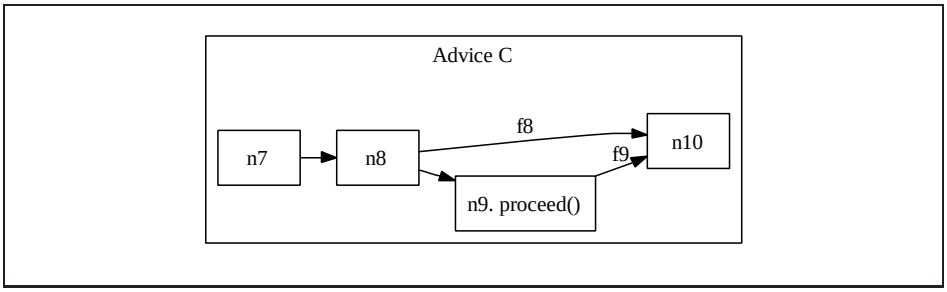

Fig. 3: Computing transfer functions for advice

Instead, then, we can compute a partial transfer function based on unknown bindings using a binding function ψ, which represents the effect of the `proceed()` statement. So for the advice in Fig. 3, the summary transfer function for the whole advice would be $\phi_C = \rho_{10} = f_9 \wedge (f_8 \circ \psi_8)$.

Using this formulation, then, we compute the summary transfer functions for methods without considering advices first. We then introduce binding information based on the possible joinpoints of each advice, and at each binding point, propagate the values of the binding function - namely, the call to the method

which is being advised - to the advice such that a summary transfer function can be calculated. The fixpoint calculation is then re-run to propagate the effect of the advice to the rest of the system.

The main advantage of this approach is that partial analysis results can be pre-computed for aspect advice, which means that when an advice is used in a different context - as a library advice, for example, or at a different joinpoint - we no longer have to start from scratch in our analysis. As well as this, we may be able to infer generic properties of the advice - in the form of a categorisation of its behaviour, for example - which could then inform our later analysis.

4 AIDA

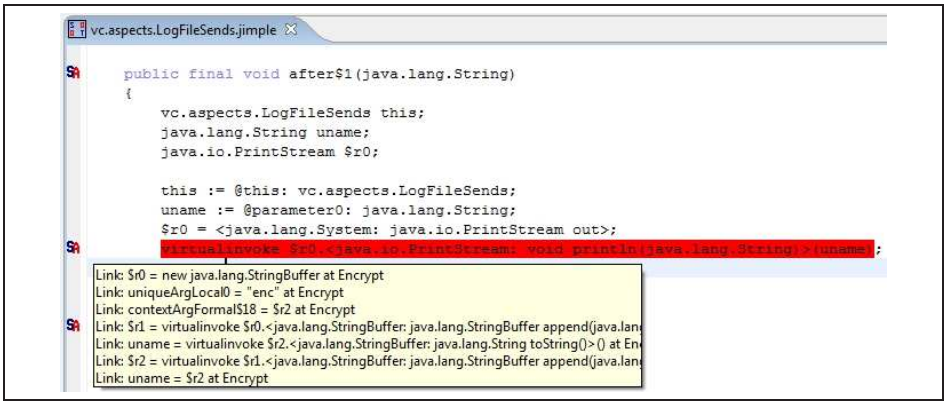

Fig. 4: AIDA in Eclipse

We have implemented the simpler version of the algorithm presented above - namely, that which treats advices as method calls - as an extension to the abc compiler[2] for AspectJ called AIDA - Aspect Interference Detection Analysis. The goal of AIDA is to provide developers with a summary and visualisation of the potential interactions within the system, such that they can develop and evolve a strategy for fault-tolerance.

abc is built on the Soot[8] framework, and so transforms the woven bytecode into an intermediate representation called Jimple, which allows inspection and analysis of the code. On running the analysis, AIDA produces annotated Jimple code which shows the links between advices in terms of which statements are directly affected by other advice being present in the system (see Fig. 4 for how this looks to the user in Eclipse[4]). The analysis also works at any specified depth of transitivity - that is, it can chain any given number of definition-use chains together to find even more subtle interactions. AIDA also produces a visual representation of interactions by means of an interaction graph, shown in Fig. 5. Here we have trimmed the graph to show the results of analysing the example

Version Control system presented in Section 1, and the interaction between the `Encrypt` and `LogFileSends` aspects is clearly shown by tracing the red arrows.

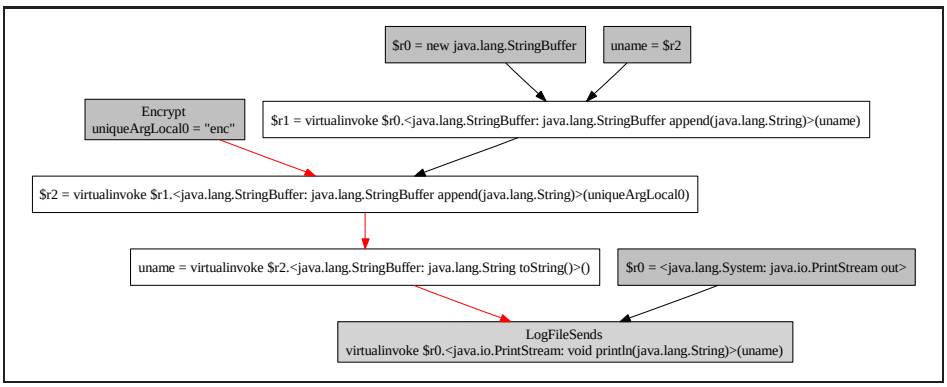

Fig. 5: AIDA-generated interaction graph

5 Conclusion and future work

In this article we have motivated the adaptation of summary-based DFA for discovery of interactions between aspects. We have motivated the importance of this interaction analysis with respect to its application to fault-tolerant systems - it can reveal error-propagation paths and outline containment units; give hints to the developer on where modules need hardening with assertions and the like; and show how aspects for fault-tolerance can be impacted by other aspects in the system. We have presented AIDA, an implementation of this analysis which works on AspectJ programs. We are currently extending our implementation to incorporate the more AO-specific algorithm described above, which treats advices differently to methods and is able to pre-compute some analysis results such that they can be reused. We have also developed a categorisation schema which describes the interactions between aspects in a more meaningful way, and we hope to incorporate all of this into an Eclipse plugin.

References

1. AspectJ. Home page of the AspectJ project. `http://eclipse.org/aspectj`.
2. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98. ACM Press, 2005.
3. Contract4Java. Homepage of the C4J project. `http://www.contract4j.org`.

4. Eclipse. Homepade of the Eclipse project. `http://eclipse.org`.
5. Jorg Kienzle and Rachid Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 37–61, London, UK, 2002. Springer-Verlag.
6. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*, chapter 2, pages 35–135. Springer, 2nd edition, 2005.
7. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
8. Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.