

Investigating feature interactions by exploiting aspect oriented programming

Mattia Monga
Dip. Elettronica e Informazione
Politecnico di Milano
Piazza L. da Vinci, 32
20133 Milano, Italy
monga@elet.polimi.it

Fatima Beltagui Lynne Blair
Computing Department
Lancaster University
Bailrigg
LA1 4YR Lancaster, UK
{beltagui, lb}@comp.lancs.ac.uk

Abstract

Users think of systems in terms of the functionalities they provide. Any coherent and identifiable bundle of system functionality perceived by users is then a *feature* of the system. Features cut across the entire system. Thus, aspect oriented languages can be exploited in order to avoid the tangling between the code implementing a feature and the other parts of the system. The problem of feature interactions becomes the problem of discovering aspect interactions and can be studied by analysing the source code units. In this paper we show how programme slicing can be used to build models of aspect oriented units. Every time one wants to reason about an aspect, a developer can use the associated slice instead of the whole programme to build useful models. We believe that this motivates the use of aspect-oriented constructs as an effective tool to manage complexity.

1 Features of Systems

When software engineers design systems, they try to cope with their intrinsic complexity by decomposing them into interacting, but clearly separated, modules that encapsulate the code concerning a particular issue. This practice has several benefits on the development process: it enables division of labour, it promotes comprehensibility and it fosters flexibility because any module can be manipulated largely independently from the others [13].

Instead, users think of systems in terms of the functionalities they provide. Any coherent and identifiable bundle of system functionality perceived by users can be viewed as a *feature* of the system. A feature is no more than a clustering of individual requirements within the specification of the behavioural characteristics of a system [14]. Users request new functionality and report defects in terms of features.

However, it is worth noting that the notion of feature, while natural in the “problem domain”, it is not always present in the “solution domain”. In fact, researchers in *feature engineering* propose to promote features as “first class objects” within the software process, in order to bridge the gap between the user needs and design or implementation abstractions.

Nevertheless, in the solution domain it is often difficult to reason about features because they affect the system as a whole. Thus, in general they are not parcelled in a well defined software module. Instead, the parts responsible for implementing a given features are scattered

among several software artifacts. Also, conversely, every single artifact contains statements related to more than one feature, all tangled in the same code.

Recently, *aspect-oriented* languages were proposed to make cross-cutting concerns clearly identifiable with special linguistic constructs.

Since features cut across the entire system, they are perfect candidates to be implemented by an aspect. This should result in greater cognitive help for implementors because they make the mapping between the problem domain and the solution domain more straightforward. Issues in the problem domain can be easily translated into issues in the solution domain, without further abstractions.

The aim of this paper is twofold. Firstly, we intend to investigate the use of aspect-oriented techniques to implement features and use the telecommunications system models of [8] for our case study. Secondly, we intend to investigate the concept of programme slicing to efficiently analyse the feature-based (or aspect-based) programme for interactions.

The paper is organised as follows: in section 2 we give a short description of our case study and introduce the basics of aspect oriented programming and how we used it to implement the case study in section 3. In section 4 we summarise the technique of programme slicing and, in section 5, show how this technique can be applied to aspect-oriented programming before discussing ways in which slices could be used to help find interactions (section 6). In section 7 we discuss some related work and we finally draw some conclusions in section 8.

2 A Case Study

In order to explore how aspect oriented programming can help in structuring systems where multiple features are planned, we implemented a simple model of a telecommunications system, whose specification comes from [8]. The model is composed of several telephone devices connected to a message switching system. Each device is controlled by a control software providing its functionalities: the core functionality just enables two telephones to be connected through the telecommunication network and is called “Basic Call Software”. A number of “features” can be added to “Basic Call Software” to enable advanced user services like “Call Forwarding on Busy” and “Ring Back when Free”.

The specification of “Basic Call Software” is depicted in Figure 1 (taken from [8]). For this system, the specifications of features are given as differences to “Basic Call Software”. For example, Figure 2 shows how the state machines associated to the subscriber and to all the other users has to be changed when the “Call Forwarding on Busy” is activated.

Our implementation is intentionally directly mapped on the specification. The “Basic Call Software” is no more than a finite state automaton with explicit codification of states and transitions (see Figure 3).

Ideally, one would like to have every other feature superimposed on “Basic Call Software” by its own separated module, in order to have a single point of inspection for further perfective or adaptive maintenance. In general, a programmer who wants to add a new feature to “Basic Call Software” faces three options:

1. change directly the code of `BCFSM`;
2. use traditional object oriented constructs to evolve the `BCFSM` class, in particular by benefiting from *inheritance* and *delegation*;


```

class BCfsm {
    // states
    public final static State bc1 = new State("idle");
    public final static State bc2 = new State("o_offhook");
    // others follow
    // transitions
    public final Transition tr1to2 = new Transition(
        bc1, Action.action( new Action( OFFHOOK, A, A, blank ) ), bc2 );
    public final Transition tr1to9 = new Transition(
        bc1, Action.action( new Action( I_ALERT, B, A, blank ) ), bc9 );
    // others follow
}

```

Figure 3: An implementation of “Basic Call Software”

3. write a new unit with the new code needed and exploit an aspect oriented mechanism to weave it with the existing classes.

The first option should be avoided because is error prone and it produces tangled code that is hard to understand and maintain. In fact, programmer intentions are totally lost and cannot be retrieved by analysing the structure of the code.

The second option is normally used by experienced programmers. A smart composition of inheritance and delegation may produce a maintainable and decoupled solution [5]. Unfortunately, it has also a number of undesirable side-effects that limit its effectiveness. The main source of difficulty is that by using complex solutions the link between the solution domain and the problem domain is no more direct: the code contains several entities that are there just to support the solution. Traceability between the concrete implementation of the solution and the concepts that programmers followed to find it becomes harder. Moreover the comprehension of the solution may be further hindered by the so called *object schizophrenia* [1]: an object that conceptually is a single entity is split in one or more objects for implementation reasons. Furthermore, it is worth noting that in general the application of these techniques requires the refactoring of the whole application in order to be effective in decoupling the feature code.

In the following section we describe our solution based on an aspect oriented language. The attractiveness of this approach lies in the ability to isolate every feature in a different artifact that is woven with the rest of the system automatically. Thus, as we are going to show in section 6, features can be understood and analysed abstracting from the complexities of the rest of the system.

3 Aspect Oriented Programming

Aspect-oriented languages provide support for writing encapsulated units that:

1. syntactically isolate the code implementing the cross-cutting concern;
2. identify *join points* in the rest of the code;

Join points are nodes in the control flow graph of the programme and a *weaver* is responsible for mixing in the cross-cutting code when a join point occurs during programme execution.

Thus, by using these languages, it is possible to write aspect oriented modules that can be merged (*woven*) with the rest of the system, affecting all the modules featuring the relevant join-points.

The best known system implementing an aspect-oriented approach is probably AspectJ [17]. Designed and implemented at Xerox PARC, it is aimed at managing tangled concerns in Java programmes. In AspectJ it is possible to define a first-class entity called an **aspect**. This construct is reminiscent of the Java **class**: it is a code unit with a name and its own data members and methods. In addition, **aspects** may define sets of join points (**pointcuts**), introduce an attribute or a method in existing types (*introductions*), and declare pieces of code (*advices*) that can be woven **before**, or **after** a join point.

Our implementation uses aspects to change the basic behaviour of “Basic Call Software”. Since “Basic Call Software” is no more than a finite states machine, an aspect implementing a feature may simply change the underlying data structure according to the specification. Figure 4 shows some excerpts from the implementation of the “Call Forwarding when Busy” feature:

- the points in the control flow of the programme where the basic state machine is initialised are identified by the **pointcut** declaration: they are the calls to the method `initFSM()`;
- immediately **after** the execution of the method `initFSM()`, code is executed that changes the state machine according the specification of the “Call Forwarding when Busy” feature.

The AspectJ compiler weaves together the Basic Call code with the aspect oriented code implementing the features. After weaving, the final product is a pure Java programme that can be executed on any standard Java virtual machine. It is worth noting, however, that the generated code contains not only the entities related to the “Basic Call” and to the features, but also a number of other entities needed to guarantee the AspectJ semantics. Thus, although in principle it is possible to apply to the woven programme all the analysis techniques known for Java programmes (for example, [2]), this is not desirable, because the structure of the woven programme contains complexities that one does not want to deal with.

The resulting aspect oriented programme is significantly less complex than an analogous object oriented one. In fact, each feature is isolated in a new programming unit and one can understand it at a glance.

However, the interactions among different aspects is not easy to grasp, and they could interfere one with another. If, in order to understand aspect interactions, one had to analyse the whole woven programme and how aspects are intertwined within it, the separation of the aspect code would be just apparent. In the following sections we describe an approach to aspect interaction analysis based on the idea that programme slicing techniques can be used to reduce the part of the code that one needs to analyse to understand what is the effect of each aspect oriented unit.

4 Programme Slicing

Programme slicing [15] is a technique aimed at extracting programme elements related to a particular computation. It has been studied mainly in the context of procedural programming languages [4]. A *slice* of a programme is a set of statements that affect a given point in the programme (*slicing criterion*). Producing the minimal slice is known to be uncomputable, however it is possible to compute non-minimal slices with fairly efficient algorithms.

```

aspect CallFwBusy {
pointcut PCinitFSM( BCfsm fsm): // assign a name to a pointcut
call ( void initFSM());          // the set of all calls to initFSM
&& target( fsm)                  // exposing the Basic Call FSM
                                   // (target of the method call)

    // a new State for the Basic Call FSM public final static
    State cfb2 = new State("forward_call");

void after ( BCfsm fsm):PCinitFSM( fsm) { // after pointcut PCinitFSM
    fsm.addAction( fsm.I_NOTIFY);          // add a new action ...

    fsm.addState( cfb2 );                  // ... and a new state

    fsm.addTransition( tr5tocfb2);        // add 4 new transitions to
    fsm.addTransition( trcfb2to6);        // the FSM data structure
    fsm.addTransition( trcfb2to8);
    fsm.addTransition( trcfb2to1);
}
}

```

Figure 4: An **aspect** implementing the feature “Call Forwarding when Busy”

In order to formalise the concepts of a programme slice and slicing criterion, we follow [7] considering the simplest programming language with just assignments, conditionals and jumps. Given a programme or a code fragment π written in this language, it is always possible to build a *control flow graph* $G_\pi = \langle N, E, s, e \rangle$ where

- N is the set of nodes representing the statements of π : a node for each assignment and each jump;
- E is the set of directed edges representing the control flow among the statements of π : if it is possible that a statement s_2 follows immediately another statement s_1 , then there is an edge from n_1 , representing s_1 , to n_2 , representing s_2 ;
- a unique start node s with no incoming arcs
- a unique end node e with no outgoing arcs

All nodes in N are reachable from s and e is reachable from all nodes in N .

A *programme slice* π_C is an executable portion of a programme π that may affect the values of variables that are referenced at some programme points. The set of interesting points is called a *slicing criterion* (C). If G_π is the control flow graph associated to π , a slicing criterion is a non-empty set of nodes $C = \{n_1, \dots, n_k\}$ where each $n_i \in N$ and $k \geq 1$. Informally, any execution of π_C is indistinguishable from an execution of π by looking at the values of variables at the points in the slicing criterion.

It is possible to build good approximations of the minimal slice, by using only compile-time information about a programme [16, 6] (*static programme slicing*). A static slicing algorithm computes programme slices by building the transitive closure of programme dependencies starting from the points in the slicing criterion and proceeding backwards. In particular

it usually takes in account *control* and *data* dependencies. However, other kinds of ad-hoc dependencies can be defined when one is interested in more complex properties of the state of the programme than the plain value of a variable: for example, in [2] some special dependencies are exploited to reason about concurrency.

A statement s_1 is *control dependent* on statement s_2 if s_2 is a conditional (therefore two possible paths start from s_2 in the control flow graph) and, assuming that the programme will terminate, only one of the paths starting from s_2 contains s_1 . A statement s_1 is *data dependent* on statement s_2 if the computation at s_1 requires a value computed in s_2 . Thus, the computation of data dependency requires the ability to compute the set of variables defined and used in every statement ¹.

Static slicing algorithms were extended to cope with statically typed object oriented systems. The major problem is the resolution of polymorphic method calls. In order to deal with them, “polymorphic” edges that represent the dynamic choice among all the possible destinations were introduced [9]. In the following section, we will show how slicing algorithms can be applied to AspectJ programmes.

5 Slicing Aspect-Oriented Programmes

In AspectJ an **aspect** is a container for data fields, methods, pointcut declarations, and advice definitions. Therefore, an aspect can be represented by a tuple $\alpha = \langle M, P, A \rangle$, where M is the set of all members, P is the set of all pointcut declarations, and A is the set of all advice definitions. Since slicing algorithms are suitable for object-oriented code, in order to use them without modifications, we introduce the concept of a class *conjugated* to an aspect. Let $\chi = \langle M, \tilde{A} \rangle$ a conjugated class with the same members M and a set of methods \tilde{A} in which there is a method m_a for each advice $a \in A$, such that m_a has the same signature and the same body of a .

Let π be an aspect-oriented programme, composed by some classes κ_i and some aspects α_j . Since conjugated classes are traditional object-oriented classes, it is always possible (see section 4) to build the control flow graph G_ψ of the conjugated programme ψ composed by the same κ_i and by the χ_j respectively conjugated to each α_j .

Moreover, each pointcut p defines a slicing criterion C_p , because it identifies some points in the control flow graph of the aspect-oriented programme ψ , and this matches the definition of slicing criterion we gave in section 4. Therefore each aspect α defines a slicing criterion $C_\alpha = \bigcup_{i \in P} C_i$. It is worth noting that the slice produced by this criterion may contain statements taken from \tilde{A} .

For example, in Figure 5 shows an **aspect** `TraceMyClasses` and its conjugated **class**. The **pointcut** `MyClass` defines the criterion C_{p_1} : the set of all statements within the definitions of the types `Circle` or `Square`. The **pointcut** `MyMethod` defines the criterion $C_{p_1} \cap C_{p_2}$: the set of all statements that are an execution of any method and satisfy C_{p_1} . The **pointcut** `MyCall` defines the criterion $C_{p_1} \cap C_{p_3}$: the set of all statements that are an execution of any method and satisfy C_{p_1} . **Summing up**, the **aspect** `TraceMyClasses` defines a slicing criterion $C = C_{p_1} \cup (C_{p_1} \cap C_{p_2}) \cup (C_{p_1} \cap C_{p_3}) = C_{p_1}$. In fact, since the **pointcut** `MyClass` is never used alone we could use the stricter criterion $(C_{p_1} \cap C_{p_2}) \cup (C_{p_1} \cap C_{p_3}) \subseteq C_{p_1}$.

This criterion C can be used to slice the programme to which the aspect is applied: only that slice can be affected by the aspect code. Thus, we can use the slice to build proper models

¹This is critical in presence of aliasing, see [11]

```

aspect TraceMyClasses {
  pointcut myClass():
    within(Circle)
    || within(Square);
  pointcut myCall():
    myClass()
    && call (* *(..));
  pointcut myMethod():
    myClass()
    && execution (* *(..));
  /**
   * Prints trace messages before methods.
   */
  before (): myMethod() {
    Trace.traceEntry("Entering")
  }
  after (): myCall() {
    Trace.traceExit("Calling");
  }
}

class TraceMyClasseTilde {
  before_myMethod() {
    Trace.traceEntry("Entering");
  }
  after_myMethod() {
    Trace.traceExit("Calling");
  }
}

```

Figure 5: An **aspect** TraceMyClasses and its conjugated **class**, TraceMyClassesTilde

of the aspect.

5.1 Problems

In the previous section we showed how slicing algorithms can be applied to aspect-oriented programmes. However, real aspect-oriented languages such as AspectJ provide powerful constructs that, while giving great power to programmers, pose a number of problems.

A first issue is the use of “inter-type declarations”. As we said above, an **aspect** can introduce a member in another type. Moreover, it is also possible to manipulate the type hierarchy, saying, for example, that an type A **extends** a type B. This might be useful when one wants to adapt an existing class to a given interface. The use of these constructs, though handy in most cases, has the evil effect to force any analysis to be “holistic”, because most of the system modules must be taken in account. If one wants, for instance, to decide if between two classes A and B an inheritance relationship holds, s/he has to analyse all the aspects, because any of them could declare such a relationship. In the authors’ opinion, the inter-type declarations are a breach of modularisation rules and should be avoided. There are no conceptual problems in dealing with them, but the analysis becomes inherently more difficult.

Another issue is the use of dynamic (run-time) properties in pointcut definitions. For example, Figure 6 shows a pointcut definition that depends on the value of the function `If.getInputFromUser()`.

In general, the use of dynamic properties means that the slicing criterion cannot be determined statically. There are two possible solutions. Let the criterion be $\delta \odot \sigma$ where δ is the part of the predicate whose value is known only at run-time, σ the part known statically, and \odot denotes any composition of the two parts. One can consider

1. the criterion defined by $(\sigma \odot \text{true}) \cup (\sigma \odot \text{false})$;

```

public class If{
    static boolean getInputFromUser(){
        return
            showDialog()
            == YES_OPTION;
    }
    public method(){
        System.exit(0);
    }
}

aspect Trace{
    before (): call (void System.exit(int))
        && if(If.getInputFromUser()){
        // do something
    }
}

```

Figure 6: A pointcut definition that uses dynamic properties

- the criterion defined by σ . in this case the code bound to the pointcut need to be enclosed in a conditional instruction $\text{if}(\delta)$

The first solution is a conservative approximation. The second solution is more efficient, but it modifies the code one is going to analyse, at the risk of including new, unexpected, joinpoints. In fact, the computation of the value of the predicate is in general a complex function that can be affected by the aspect code.

6 Finding Interactions

Effectively, the problem of feature interactions becomes the problem of discovering aspect interactions that we propose can be studied by analysing the source code units. Intuitively, if two features are implemented by two different aspects α, β a sufficient condition to ensure that no feature interaction arises is that $S_{C_\alpha} \cap S_{C_\beta} = \emptyset$, where $S_{C_\alpha}, S_{C_\beta}$ are the slices associated to the two aspects. This is of course a conservative approach, because it is possible that although the intersection of the two slices is not empty, the intersection is such that interferences are in fact collaborations. Moreover, since the slices in general are not minimal (i.e., the computed slice contains some code that has no actual influence on the interesting points specified in the slicing criterion), it is also possible that the intersection is not empty, but the code in it has no influence on the rest.

Let's consider the feature "Call Forwarding when Busy" codified by the **aspect** showed in Figure 2. The actual code is more complex, but the statements in the example are sufficient to get the idea of the approach.

At the beginning a **pointcut** is declared: the programmer who wrote the aspect was interested in pick up all the points in the code where the method **void** `initFSM()` is called on an object of type `BCfsm`. Therefore, these points are the "slicing criterion" defined by `CallFwBusy`.

The slicing criterion can be used to get the slice of the programme affected by the aspect. We can distinguish a *backward* slice and a *forward* slice [7]: the backward slice is computed by starting from the calls to **void** `initFSM()` and looking backward through the programmes's control flow graph to find other programme statements that influence the execution of the method; the computation of the forward slice is done in the opposite direction, looking forward through the programme's control flow graph to find the other statements influenced by those method calls. It is worth noting that the forward slice also contains the code

of the advice **void after**(BCfsm). Moreover, the slice contains all the statements that change the data structure representing the finite states machine of the “Basic Call Software”.

This operation can be applied to another feature, for example “Ring Back when Free”. A characteristic of our implementation of the system is that all the features simply change the finite state machine of “Basic Call Software” when this is initialised. Therefore, the structure of the aspect implementing the feature “Ring Back when Free” is analogous to the one used for “Call Forwarding when Busy” and, consequently, the slicing criterion is the same as before. This implies that the computed slice is also the same. Therefore, the two features have a non empty intersection: we cannot guarantee that it is safe to weave both aspects with the same code.

In fact, the two features under analysis have a nasty interaction, because they change in inconsistent way the same states of the basic finite states machine. Unfortunately however, due to the structure of our implementation *any* two features analysed with our approach would give a possible interaction. The problem is that all the aspects modify the finite state machine object (a singleton in our implementation) and the slicing is therefore not able to discriminate among the details of the modification. This is a general problem affecting all the implementations in which the behaviour is embedded in a data structure: any modification changes the data structure, but slicing is a technique aimed at find the smallest number of statements affecting some computation without taking in account the details about what such statements really do. Thus, we can conclude that the abstraction provided by slicing is too coarse for applications in which the behaviour is embedded in data structures.

However, the implementation style we have selected for our case study, i.e. the representation of program control through a data structure (to directly mirror the implementation), is not typical of applications in the real world. Hence, in general, reasoning on a slice rather on the entire programme is a worthwhile simplification, that enables efficient analysis to be carried out, as demonstrated in the programme slicing literature. For example, it has obvious benefits when it comes to the state-space explosion problem associated with analysis techniques such as model-checking. Further work is still clearly required though, to establish to what extent this approach can be used to help in the identification of feature interactions.

Moreover, it is worth noting that our approach is aimed at discovering unwanted interactions among the implementations of different feature. Sometimes the interaction can arise at a different of abstraction. In [12] an interaction between “forwarding” and “auto-responding” is discussed notwithstanding how the two features are implemented. In fact, the two operations could be interacting, at user level, even if, at code level, no interference can be found.

7 Related Work

Aspect oriented programming is in its infancy, and while many are advocating its use in order to achieve a better separation of concerns, a real assessment of the power of expression of these new constructs is still lacking. In particular, we need a better understanding about how the properties of the separated parts compose to give the properties of the whole.

Most of the aspects one can write are not orthogonal and the programmer is responsible for identifying interactions between conflicting aspects and for implementing conflict resolution. In [3] the authors propose a framework to find and support resolution of interactions occurring when two aspects cross-cut the same part of the execution trace. They build a general abstract model of aspect oriented programming loosely based on AspectJ semantics, and

they prove that some conditions are sufficient to guarantee that no interaction is possible. Our work is based on the same idea, but we work directly on the code.

The weaving of aspects is non commutative: the order of application counts. [10] shows an example in which the synchronisation policy of a bounded buffer is codified by using aspects. If the aspects are applied in different order the policy may be broken. In the paper they model each aspect with a Labelled Transitions System and by exploiting model checking they show which orderings are deadlock free. We aim at a more general approach in which models can be derived systematically.

Slicing of aspect oriented programmes written in AspectJ was proposed in [18], on the assumption that pointcut declarations contained only static characterisations. Our approach takes in account also the use dynamic properties.

8 Conclusions and Future Work

The features in telecommunication systems typically affect a system as a whole. Thus, in general their implementation is not parcelled in a well defined software module, but dispersed among several software units. Therefore we propose to use an aspect oriented language to express them. Using our approach a feature results isolated in a single, more manageable unit.

However, the cognitive benefits would be lost if the comprehension of aspect properties entailed the analysis of the whole programme. Instead, if we are able to define some boundaries around aspect influence, the separation turns out to be not just syntactic sugar but a true aid in dealing with programme complexity.

Thus, we noted that aspect oriented languages define which interesting points of the rest of the system are relevant for the weaving. These points can be used to slice the system with respect a specific aspect. In this paper we showed that every time one wants to reason about an aspect, s/he can use the associated slice instead of the whole programme to build useful models. We believe that this motivates the use of aspect-oriented constructs as an effective tool to manage complexity. Our ultimate goal is to provide programmers with a set of tools that they may use to derive useful models of their code to understand interactions among aspects.

Our original aims for this work were twofold, as stated previously in this paper. We believe that the first, using aspect-oriented programming to model the cross-cutting behaviour of features, has been effectively demonstrated through the use of the “Basic Call Software” and its associated features. Unfortunately, the structure we selected for this case study, intended to ensure a direct mapping from the specification to the implementation, did not lend itself to our second aim, namely investigating the use of programme slicing techniques for feature interaction analysis. However, we have shown how such techniques can be applied to aspect-oriented programmes, and have argued that they are still a valuable solution to managing the complexity associated with analysing such programmes. Further work is required to determine if this value can be transferred to the domain of feature interaction analysis.

Acknowledgments

The authors want to give credit to Katarina Mehner for her fruitful comments on preliminary versions of this paper.

References

- [1] Subject-oriented programming and design patterns. <http://www.research.ibm.com/sopcpats.htm>. IBM Thomas J. Watson Research Center, Yorktown Heights, New York.
- [2] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.
- [3] Rémi Douence, Pascal Fradet, and Mario Südholt. Detection and resolution of aspect interactions. Technical Report 4435, INRIA, April 2002.
- [4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming*, volume 167 of *LNCS*, pages 125–132, Toulouse, France, April 1984. Springer.
- [5] Erich Gamma, Richard Helm, and Ralph Johnson und John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison–Wesley, 1986.
- [6] Mary Jean Harrold and Ning Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 74–83. IEEE Computer Society Press / ACM Press, 1998.
- [7] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, December 2000.
- [8] Mario Kolberg, Evan H. Magill, Dave Marples, and Stephan Reiff. Second feature interaction contest. <http://www.cs.stir.ac.uk/~nko/fiw00/>, 2000.
- [9] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505. IEEE Computer Society Press / ACM Press, 1996.
- [10] Torsten Nelson, Donald Cowan, and Paulo Alencar. Supporting formal verification of crosscutting concerns. *Lecture Notes in Computer Science*, 2192:153–168, 2001.
- [11] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Effects of pointers on data dependences. Technical Report GIT-CC-00-33, College of Computing, Georgia Institute of Technology, December 2000.
- [12] Jianxiong Pang and Lynne Blair. Separating interaction concerns from distributed feature components. submitted for publication, 2002.
- [13] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [14] C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L. Wolf. A conceptual basis for feature engineering. *The Journal of Systems and Software*, 49(1):3–15, December 1999.
- [15] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, March 1981.
- [16] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
- [17] XEROX Palo Alto Research Center. *AspectJ: User’s Guide and Primer*, 1999.
- [18] Jianjun Zhao. Slicing aspect-oriented software. In *Proceedings of the International Workshop on Principles of Software Evolution IWPSE 2002*, Orlando, Florida, May 2002. ACM.