Reasoning on AspectJ Programmes

Lynne Blair
Computing Department
Faculty of Applied Sciences
Lancaster University
LA1 4YR Lancaster, UK
lb@comp.lancs.ac.uk

Mattia Monga
Dip. Informatica e Comunicazione
Università degli Studi di Milano
Via Comelico 39/41
20135 Milano, Italy
mattia.monga@unimi.it

ABSTRACT
In this paper we suggest that in order to analyse the properties of an AspectJ **aspect** one can consider the aspect itself and the part of the system it affects. In fact, we argue that in AspectJ every pointcut declaration defines a "slicing criterion" that can be used to compute the associated slice. One can use the sliced programme to build useful models of the system and the aspects and exploit them to prove properties. For example, non-interference at code level can be guaranteed if the slices associated to different aspects are disjoint.

1. MOTIVATION
When software engineers design systems, they try to cope with the intrinsic complexity of their systems by decomposing them in 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 pretty independently from the others [10]. However, it has been noticed that sometime concerns are difficult to encapsulate because they cross-cut several parts of the system and are tangled with the code that addresses different concerns. For example, this is typically the case of concurrency and security issues.

Recently, *aspect-oriented* languages were proposed to make cross-cutting concerns clearly identifiable with special linguistic constructs. 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 [16]. Designed and implemented at Xerox PARC, it is aimed at managing tangled concerns in Java programs. 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, **aspect**s may define sets of join points (**pointcut**), introduce an attribute or a method in existing **class**es (*introductions*), and declare pieces of code (*advices*) that can be woven **before** or **after** a join point.

Aspect oriented programming is very convenient to express cross-cutting concerns. A typical aspect oriented statement can be something like "before any division, check if the divisor is not zero"; in a very economical way it is possible to affect all the divisions in the code, *even without knowing where these divisions will occur.* However, this strength is also a source of complexity, because it can be difficult to figure out the behaviour of the whole system: every time a division is performed, one has to remember that also the aspect oriented code is executed. In fact, it is often very difficult to guess where are the join points affected by aspect oriented code.

Moreover, one often does not want to reason on the "woven" programme, but on the aspect oriented code in isolation: after all this was the motivation for structuring it as a separate module. For example, if a synchronisation policy is codified with an aspect, programmers would be able to prove that the policy is deadlock free, without studying the details of the code to which is applied [2].

In this paper we suggest that in order to analyse the properties of an aspect one can consider the aspect itself and the part of the system it affects. This part is just a *slice* of the entire system and, while the minimum slice is not computable, some good approximations can be extracted exploiting well known programme slicing algorithms [11]. We argue that in AspectJ every pointcut declaration defines a "slicing criterion" that can be used to compute the associ-

GI-AOSDG 2003 Essen, Germany

ated slice. One can use the sliced programme to build useful models of the system and the aspects and exploit them to prove properties. The paper is organised as follows: in Section 2 we briefly introduce programme slicing, in Section 3 we explain how programme slicing can be applied to aspect-oriented programmes, in Section 4 we discuss the current limitations of our approach, and, after a short discussion on related works in Section 5, in Section 6 we draw some conclusions envisioning our future work.

2. PROGRAM SLICING

Program slicing [14] 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 [3]. A *slice* of a programme is a set of statements which 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.

In order to formalise the concepts of programme slice and slicing criterion, we follow [6] 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* $G_\pi =< N, E, s, e >$ 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 $C = \{n_1, \ldots, n_k\}$ 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 [15, 5] (*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 backward. 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 variable: for example, in [1] 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 [8]. In the following Section, we will show how slicing algorithms can be applied to AspectJ programmes.

3. 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 $\alpha =< M, P, A >$, 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 $\chi =< M, \tilde{A} >$ 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 2) 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 2. 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 1 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,

[1] This is critical in presence of aliasing, see [9]

```
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 TraceMyClassesAspect {
  before_myMethod() {
    Trace.traceEntry("Entering");
  }
  after_myMethod() {
    Trace.traceExit("Calling");
  }
```

Figure 1: An **aspect** `TraceMyClasses` and its conjugated **class**, `TraceMyClassesAspect`

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 of the aspect.

4. PROBLEMS

In the previous section we showed how slicing algorithms can be applied to aspect-oriented programmes. However, real aspect-oriented languages 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 breech of modularisation rules and should be avoided. There are no conceptual problems in dealing with them, but the analysis becomes inherently more difficult. The use of wildcards in pointcut declaration has a similar impact: it does not preclude the analysis, however

1. some closed world assumption is needed;

2. the analysis itself cannot be modularised, implicitly increasing its complexity.

Thus, the use of inter-type declarations and wildcards forces the analisys to take into account all the code of system.

Another issue is the use of dynamic properties in pointcut definitions. For example, Figure 2 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 $(\sigma \odot \textbf{true}) \cup (\sigma \odot \textbf{false})$;

2. the criterion defined by σ. in this case the code bound to the pointcut need to be enclosed in a conditional instruction **if**(δ)

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 even affected by the aspect code.

5. RELATED WORKS

Slicing of aspect oriented programmes was proposed by Jianjun Zhao in [17]. In his paper, Zhao applies slicing techniques to AspectJ code. He tries to identify which statements might affect a given statement in an aspect-oriented programme. He does not consider inter-type declarations, wildcards, and dynamic properties. Our goals, while we use an analogous technique, are rather different: we exploit the fact that each aspect define a slicing criterion that can be used to build a model of the entire system to reason about aspect weaving.

In our work we were inspired by Bandera [1]. Bandera is aimed at model checking plain Java programmes. In order to simplify the model, Bandera uses the property to be verified (in general a formula in a linear temporal logic) to build a slice of a programme (see Figure 3.a). In fact, the model is

```
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()){
        System.out.println(thisJoinPoint);
    }
}
```

Figure 2: A pointcut definition that uses dynamic properties

a model of the relevant slice. Originally we wanted to build a model by examining an aspect in isolation (the experiment is described in [2]). However, this goal resulted very difficult to achieve, because of the number of hypothesis one needs to impose on the code affected by the aspect. In this paper we suggest that an aspect can be used to reduce (but not eliminate) the part of the programme that one must analyse in order to model check properties of the system under verification (see Figure 3.b).

6. CONCLUSIONS AND FUTURE WORK

Aspect-oriented programming as intended by AspectJ is a way to express incremental modifications to the behaviour of a programme. An advice declaration specifies an action to be taken whenever some condition arises during the execution of the programme [13, 4]. We claim that in AspectJ every pointcut declaration defines a "slicing criterion": a set of interesting points in the control flow graph of the programme. Therefore, one can use well known slicing algorithms to compute the part of a programme that is affected or affects a given aspect. The ultimate goal of aspect-oriented programming is the separation of otherwise crosscutting concerns. However, these benefits are lost if the comprehension of aspect properties entails 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.

We plan to use our approach to solve feature interaction problems. A feature is no more than a clustering of individual requirements within the specification of the behavioural characteristics of a system [12]. Since features cut across the entire system, they are perfect candidates to be implemented by an aspect. Thus, the problem of feature interactions becomes the problem of discovering aspect interactions and can be studied by analysing just the source code. Intuitively, if two features are implemented by two distinct 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.

In general, 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.

7. ACKNOWLEDGMENTS

The authors want to thank Katharina Mehner and the anonymous reviewers for their useful comments on early drafts of this work.

8. REFERENCES

[1] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. 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.

[2] G. Denaro and M. Monga. An experience on verification of aspect properties. In T. Tamai, M. Aoyama, and K. Bennett, editors, *Proceedings of the International Workshop on Principles of Software Evolution IWPSE 2001*, pages 184–188, Vienna, Austria, Sept. 2001. ACM.

[3] 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, Apr. 1984. Springer.

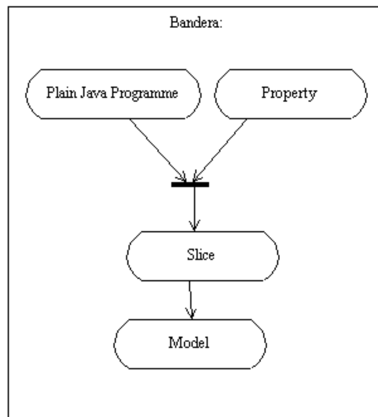
[4] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of OOPSLA 2000 workshop on Advanced Separation of Concerns*, 2000.

[5] M. J. Harrold and N. 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.

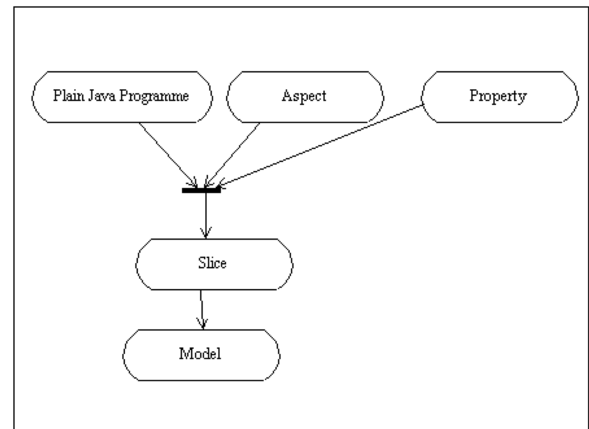
[6] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, Dec. 2000.

[7] R. Lämmel. A semantical approach to method-call interception. In G. Kiczales, editor, *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Enschede, The Netherlands, Apr. 2002. ACM, ACM Press.

[8] L. Larsen and M. J. 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.



(a)



(b)

Figure 3: Flow of data in Bandera and in our proposal

[9] A. Orso, S. Sinha, and M. J. Harrold. Effects of pointers on data dependences. Technical Report GIT-CC-00-33, College of Computing, Georgia Institute of Technology, Dec. 2000.

[10] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[11] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[12] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *The Journal of Systems and Software*, 49(1):3–15, Dec. 1999.

[13] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, number 02-06 in Technical Report, pages 1–8. Department of Computer Science, Iowa State University, Apr. 2002.

[14] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, Mar. 1981.

[15] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[16] XEROX Palo Alto Research Center. *AspectJ: User's Guide and Primer*, 1999.

[17] J. Zhao. Slicing aspect-oriented software. In *Proceedings of the International Workshop on Principles of Software Evolution IWPSE 2002*, Orlando, Florida, May 2002. ACM.