Understanding Aspects via Implicit Invocation

Jia Xu Hridesh Rajan Kevin Sullivan
Department of Computer Science, University of Virginia
{jx9n, hr2j, sullivan}@cs.virginia.edu

Abstract

Aspect-oriented (AO) design and programming methods promise to improve the modularity properties of software-intensive systems. However, AO is also seen as violating fundamental design principles; and we lack a theory to guide its appropriate use. Our work rests on the idea that successful AO techniques have deep roots in implicit invocation (II) mechanisms. Elaborating this connection provides for an expedited development of both a theoretical understanding and an effective practice of AO design techniques. In this paper we show, in particular, that this bridge can be exploited to enable model checking of AO systems using existing techniques for II systems.

1. Introduction

Aspect-oriented programming (AOP) [14] has the potential to improve software design, but it also challenges prevailing design ideas. Aspects can modify the behaviors of other modules, which conflicts with ideas of abstraction and the integrity of encapsulated data. Aspects also come to depend on implementation details of other modules, which conflicts with traditional ideas of abstraction and information hiding.

One response to these tensions is to restrict the power of aspect mechanisms. Proposals of this form encounter stiff resistance in the AOP community as "throwing out the baby with the bath water." If we are to retain the power of AO, however, it is clear that we need a new theory of design to guide us in the appropriate use and development of its mechanisms.

The idea—not new—on which this paper is based is that AO rests on implicit invocation [12][17][19]. The difference is that events are represented explicitly in II systems, but are provided implicitly by the semantics of AO language designs. In other words, II is *explicit* implicit invocation; AO is *implicit* implicit invocation. A thesis of our research program is that we can exploit this deep connection between AO and II as a bridge to transport knowledge and theory developed in the II realm to the AO realm.

In past work [22], we interpreted AO join points as events, and showed that limitations of current AO event models prevent their use to support design techniques that ease the design and evolution of integrated systems [23]. Next, we showed that extending language models in light of this analysis solves the problem [18]. This paper shows that a formal reduction from AO to II enables model checking techniques for II [4][11] to be used to check properties of AO systems automatically. Studying the relationship between II and AO can help resolve thorny theoretical and practical issues for AO.

In the rest of the paper, we start by presenting a simple formal reduction from AO space to II space. We then support the claim that the reduction is useful by showing that it enables existing model checking techniques for II to be used to check AO programs.

2. Reduction from AOP to II

In this section, we make the connection from AOP to II concrete as a simple formal reduction.

2.1 Aspect Oriented Programming Space

Given a program, we denote its set of objects by O:

$O = \{o \mid o \text{ is an object in the program}\}$

A *join point* is a point in program execution exposed by the language as an advisable event. The join points exposed by AspectJ-like languages include method call and execution, field get and set operations, exception, and object initialization, etc. We denote the join points of the program by J:

$J = \{j \mid j \text{ is a join point in the program}\}$

P denotes the set of all *pointcuts* in a program, where a *pointcut* is a predicate on the join points:

$P = \{p \mid p \text{ is a pointcut in the program}\}$

The selection by a pointcut P of a join point set J is denoted as a binary relation between P and J named PJ:

$PJ \subseteq P \times J$

An *advice* body is a special method meant to run when the program execution encounters selected join points. We denote the set of all advice constructs by A:

$A = \{a \mid a \text{ is an advice construct in the program}\}$

The execution of advice can be ordered in one of three ways with respect to a join point: *before, after,* or *around*. For simplicity, we do not model *around* in this paper. The set T denotes the available orders:

$T = \{after, before\}$

A key construct in AO is the association of an advice body with a pointcut. Such an expression means that the advice should run at each selected join point, with the specified ordering. The ternary relation *ATP* represents these relations in the given program:

$ATP \subseteq A \times T \times P$

Finally, and most importantly, the pointcut is resolved to a set of join points (events) and the advice is registered with each such event. The ternary relation *ATJ* represents this composition of ATP and PJ.

$ATJ = ATP \circ PJ \subseteq A \times T \times J$

2.2 Implicit Invocation Space

The functionality of an II system can be viewed as an event-handler binding relationship [12][23], which can be modeled as follows:

$C = \{c \mid c \text{ is a component in the system}\}$

$E = \{e \mid e \text{ is an event in the system}\}$

$H = \{h \mid h \text{ is an event handler in the system}\}$

$HE \subseteq H \times E$

HE models the event-handler binding as a binary relation between the set of events *E* and the set of event handlers *H*.

2.3 Reduction Rules

To reduce constructs in the AOP space to constructs in the II space, we established a set of reduction rules from advising in aspect space to event-handler binding in II space. First, a function *w* maps objects in AO space to components in II space.

$w : O \to C$

The role of an advice, with respect to the advised join point is the same as the event handler with respect to the captured event. Here *x* represents a one-to-one relation from the set of advice bodies (*A*) to the set of handler (*H*).

$x : A \to H$

Next, the partial function *y* maps the cross product of join points (*J*) and orderings in AO space to events in II space. *Before* and *after* a join point are two distinct events. The mapping *y* is partial because not all join points have meaningful *before* or *after* events. In AspectJ, for example, before object-initialization or after the execution of an exception handler are not defined.

$y : T \times J \to E$

The critical reduction rule is *z*, a composition of the reduction rules *x* and *y*. It produces an event-handler binding corresponding to an advice invocation. To reduce the state space of the II model, we prune the events that do not have a mapping to any handler.

$z : ATJ \to HE$

$\forall < a,t,j > \in ATJ, z(a,t,j) = < x(a), y(t,j) > \in HE$

In addition, the reduction process adds a dispatcher and a dispatch policy module to the reduced II system. The dispatcher is responsible for event storage, binding, delivery, and interacting with the policy module. The policy module implements the event delivery policy, which could be first-come-first-serve or some other policy. When the dispatcher receives an event, it inquires of the policy module to decide the action to take. We assume first-come-first-serve.

3. Model Checking AO Programs

We now show that the bridge we've constructed between AOP and II can improve our ability to use AO techniques. In particular, this bridge allows us to model check AO programs (AOP's) by reducing them to II systems and applying previous techniques [4][8][11].

Model checkers generally verify the correctness of assertions or temporal logic expressions. To model check AOP's, we first translate the AOP's to equivalent II programs. We also reduce properties of AOP's to assertions over components in II space:

$u : P \to P'$

The reduction rule *u* maps *p*, a property assertion in aspect property space *P*, to *p'*, a property assertion in implicit invocation property space *P'*.

The verification condition is that the property *p* is true if and only if *p'* is true.

$\forall p \in P, V(p) \Leftrightarrow V(u(p))$

This is true since our work focuses on the impact of an aspect module on other modules. Assertions about the impact of aspects can be mapped to assertions about the semantics of events over components. The existing model checking approaches are then applied to the II program and the reduced property. On a yes output, no result reduction is required. On a no output, a counter-example is produced in the II space that could, in principle, be mapped back to the AOP space.

In practice, we translate an aspect-oriented program reduced to an II system into a PROMELA [21] program, which is the input language for SPIN [21] model checker. For the base code, which is equivalent to an OO program, we adapt the translation approach used by Java Pathfinder (JPF) [13]. For the aspect code, we perform our formal reduction from AOP to II. Rest of the section explains the process.

3.1 Classes and Aspects

In PROMELA, one cannot directly access the data variables of a process from outside the process. Hence, we choose to separate the set of data variables and the set of methods. Like JPF, we encapsulate the data of a class using a record. In addition, we pre-define a data area for every class, i.e., an array of records. For each new creation of an object, a new record is allocated. An index variable points to the next free record. We treat each instance of an aspect as an object of a class.

3.2 Methods and Advice

PROMELA does not natively support function definitions and calls. To implement function calls we declare a separate *active* process (initiated from the beginning) for each method, which acts as a server: responding to requests and receiving parameters from the user processes via a call channel and returning results to the user process via a return channel.

As for parameter passing, arrays cannot be parameters of a proctype. Although passing arrays via message channels is feasible, it is still unappealing to refer to an object by itself, since PROMELA does not natively support call-by-reference or call-by-value on complex data structures. Therefore, we prefer to use an index value to refer to an object.

Advice bodies are treated as methods, except that the parameters of the former are often implicit.

3.3 Events and Binding

An implicit invocation system is driven by events. We represent events as messages. Whenever the base program raise an event, it sends a message containing the event information: the join point, the source of the message, namely, the object that raises the event and the reflective information available at the join point as event parameters to be used in event handling.

We model the message dispatcher as a separate active process in PROMELA. It monitors the event channel and dispatches messages to specified event handlers (i.e., advice). The message dispatcher is also responsible for the context matching of dynamic join points like *cflow* and *cflowbelow*.

3.4 Checking properties

To verify properties, we use a separate monitoring process. This process checks every assertion at stable points like *timeout*, a system-defined condition in PROMELA, which has the value *true* in all global system states where no statement is executable in any active process, and false in all other states.

3.5 Other issues

Scalability is an important issue. On one side, our translation from aspect code to PROMELA is a pre-compilation process and can perhaps be automated to an extent. On the other side, the scalability of the model checking approach depends on the scalability of SPIN and PROMELA. In particular, there are limits on the maximum number of objects and the maximum number of active processes. Some SPIN extensions [24] address these issues.

Specifying properties of aspect-oriented programs is a challenge. A property about a program has two parts: first, a specification of the property, and, second, when it must hold. Currently PROMELA does not provide a way to express points where assertions hold. In the future, we will explore better ways to specify properties.

4. Related Work

Our work provides the necessary bridge to apply Dingel et al's. [8], Garlan et al's. [11], and Bradbury et al's [4] work on II to AOP. Filman and Havelund [10] and Walker and Murphy [25] elicit roots of AOP in II. Our contribution is in exploring ways to formalize and then exploit the connection in order to develop theory, methods, and tools to improve our abilities to use AOP.

There are other approaches to directly or indirectly reason about AOP. Blair and Monga [3], for example, view each pointcut as a *slicing criterion*. They then propose to check slices using Bandera [6], but the expressiveness of aspects is difficult to capture by any slicing technique, and the work remains at the conceptual stage. Clifton et al. [5] on the other hand suggest classifying aspects and making the aspect invocation explicit compromising the obliviousness [8]. Our approach on the other hand, does not impose any restrictions on the language model of AOP; nor are we concerned with component properties, here, but only with system properties. Devereux [7] expresses program properties as assertions in alternating time logic; however, the lack of tool support for automated reasoning in alternating-time logic makes the reduction less attractive. Sihman and Katz [20] explore model checking of aspects. They are concerned with verifying that aspect code does not compromise base code satisfaction of its specification, and with verifying aspects independently of specific base code. Model checking of aspect code per se is not a new idea. Our contribution is to show that developed techniques for II can be adapted directly for analysis of AO programs.

Aldrich proposes type theory to ease reasoning about programs in restricted AO languages [1]. He presents an aspect language called TinyAspect based on module sealing and explicit declaration of exported join points. The idea is to enforce abstraction by prohibiting aspects from exploiting implementation details such as calls from within components to their own methods (as in our running example). The approach rests on a problematical hiding of certain join points, leaving visible only those that are likely to be adequate as proxies for semantic events of interest.

5. Conclusion and Future Work

We have shown that the bridge between AO and II promises to help us better understand and use AO. In particular, it allows us to import concepts, methods, tools, and techniques from the II domain for use in AO. In this paper, we showed that this connection enables the direct application of model checking methods for II to AO systems.

In future work, we anticipate developing a critique of the current AO event model and its implications for the design of reliably evolvable systems.

6. Acknowledgements

This work was supported in part by the National Science Foundation under grant ITR-0086003.

7. References

[1] Aldrich, J., “Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming”, FOAL 2004, Lancaster, UK, March 2004.

[2] AspectJ: http://www.eclipse.org/aspectj.

[3] Blair, L., Monga, M. “Reasoning on AspectJ Programmes”, GI-AOSDG 2003 Essen, Germany.

[4] Bradbury, J. S., Dingel, J.,"Evaluating and Improving the Automatic Analysis of Implicit Invocation Systems", ESEC/FSE 2003, Helsinki, Finland, Sept. 2003.

[5] Clifton, C., and Leavens, G. T., “Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning.”, Technical Report TR#02-04, Department of Computer Science, Iowa State University, Mar 2002.

[6] Corbett, J.C. et al., “Bandera: Extracting Finite State Models from Java Source Code”, Proc ICSE 2000.

[7] Devereux, B., “Compositional Reasoning About Aspects Using Alternating-time Logic", FOAL 2003.

[8] Dingel, J., Garlan, D., Jha, S., Notkin, D., "Reasoning about implicit invocation", FSE-6, November 1998.

[9] Filman R., and Friedman, D., “Aspect-oriented programming is quantification and obliviousness”, In Proc. Workshop on Advanced Separation of Concerns, OOPSLA 2000.

[10] Filman, R.E., Havelund, K.. "Realizing Aspects by Transforming for Events." ASE Sept. 2002.

[11] Garlan, D., Khersonsky, S., and Kim, J. S., "Model Checking Publish-Subscribe Systems", SPIN 03, Portland, Oregon, May 2003.

[12] Garlan, D., and Notkin, D., “Formalizing Design Spaces: Implicit Invocation Mechanisms”. *VDM '91: Formal Software Development Methods*, Oct. 1991.

[13] Havelund, K., and Pressburger, T., “Model checking JAVA programs using JAVA pathfinder”, International Journal on Software Tools for Technology Transfer, 2(4):366--381, 2000.

[14] Kiczales, G. et al., “Aspect-oriented programming,” ECOOP 1997, June 1997.

[15] McMillan, K, “ Cadence SMV”, http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/

[16] C#: http://msdn.microsoft.com/net/ecma.

[17] Notkin, D. et al., “Adding Implicit Invocation to Languages: Three Aproaches,” Proc. JSSST Symp. Object Technologies for Advanced Software, Springer-Verlag LNCS 742, November 1993.

[18] Rajan, H. and Sullivan, K., “Eos: Instance-Level Aspects for Integrated System Design”, ESEC/FSE 03, Helsinki, Finland, Sept 2003.

[19] Reiss, S.P., **“**Connecting Tools Using Message Passing in the Field Environment,” *IEEE Software*, Jul. 1990.

[20] Sihman, M. and Katz, S.. Model Checking Applications of Aspects and Superimpositions, FOAL 2003.

[21] SPIN, http://spinroot.com/spin/whatispin.html.

[22] Sullivan, K., L. Gu, and Y. Cai, “Non-modularity in aspect-oriented languages: integration as a cross-cutting concern for AspectJ,” Proceedings of AOSD 2002.

[23] Sullivan, K. and Notkin, D., “Reconciling environment integration and software evolution,” ACM Transactions on Software Engineering and Methodology 1, 3, July 1992, pp. 229–268.

[24] Visser, W. et al., , “Adding Active Objects to SPIN”, Lecture Notes in Computer Science, Jul & Sept. 1999.

[25] Walker, R. J. and Murphy, G. C., “Joinpoints as ordered events: towards applying implicit context to aspect-orientation”, Workshop on Advanced Separation of Concerns at the 23nd ICSE, 2001.