
FRAMEWORK-BASED MODEL CONSTRUCTION WITH AOP ASSISTANCE

by

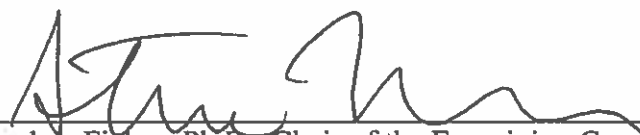
ZEBIN CHEN

A DISSERTATION

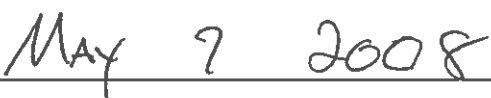
Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2008

“Framework-based Model Construction with AOP Assistance,” a dissertation prepared by Zebin Chen in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:



Stephen Fickas, Ph.D., Chair of the Examining Committee



Date

Committee in Charge: Stephen Fickas, Ph.D., Chair
 Michal Young, Ph.D.
 Yannis Smaragdakis, Ph.D.
 John Orbell, Ph.D.

Accepted by:



Dean of the Graduate School

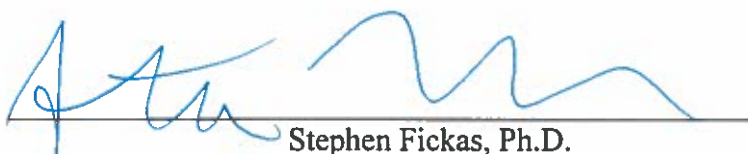
© 2008 Zebin Chen

An Abstract of the Dissertation of
Zebin Chen for the degree of Doctor of Philosophy
in the Department of Computer and Information Science

to be taken June 2008

Title: FRAMEWORK-BASED MODEL CONSTRUCTION WITH AOP ASSISTANCE

Approved:

A handwritten signature in blue ink, appearing to read 'Stephen Fickas', is written over a horizontal line. The signature is stylized and cursive.

Stephen Fickas, Ph.D.

I am part of a group that is building software for smart homes using a Java framework called *Open Service Gateway Initiative (OSGi)*. Our concern is with proving certain properties correct about the framework and any application built on top of it. I have found it useful to take a model-checking approach to the proof problem. In recognition of the commonality of OSGi applications, I believe it is possible to build a modeling framework parallel to the OSGi specification to ease the model construction for OSGi applications.

I have tested the ideas by (a) constructing a modeling framework, and (b) model-checking several benchmark OSGi applications pulled off the web using that framework. The good news is that I was able to discover property violations in these applications through model checking, with relatively small efforts to formalize the applications. The bad news is that when checking certain applications, it requires customizations scattered in various places of multiple Java files, even including the modeling framework itself.

To accommodate such crosscutting concerns, I propose to use *Aspect-Oriented Programming (AOP)* to add another unit to modularize the needed changes. With AOP

techniques, I pre-process a generic modeling framework with pointcuts, inter-type definitions and other AOP constructs. The pre-processing step adds the only needed details to the bare essentials. While these details may be scattered in different parts of the application, with AOP techniques I am able to specify such variations in an explicit and modular way. Varying application details then becomes a simpler task to specify and select the related aspects.

Two problems arise due to the adoption of the AOP techniques: one concerns native code and the other concerns performance penalties. I have created an abstraction library to resolve the native code in the AspectJ runtime library as needed, and have come up with a test suite in JUnit for regression test. The performance penalty is due to the internal variables used by aspect transformation and subsequent interleaving in a multi-threaded system. I thus make a distinction between model checking AspectJ applications in general and using AOP techniques to vary an existing formal model. For the latter problem, I present several approaches to reduce the search space, and am able to reduce the state space of an AspectJ program to a comparable size of its counterpart in pure Java implementation.

I have thus used the above technologies for rapid construction of formal models for OSGi applications. I am able to uncover several bugs in some benchmark OSGi applications with comparable search space. On the other hand, the extra interleaving of AspectJ programs suggests a potential data race. I have uncovered bugs in general Aspect-Oriented programming mechanisms that have been widely used and cited. To patch such bugs, I propose a general solution to avoid data races during aspect instantiation for a class of applications based on the current AspectJ compilers.

This thesis includes both my previously published and my co-authored materials.

CURRICULUM VITAE

NAME OF AUTHOR: Zebin Chen

PLACE OF BIRTH: Xiamen, Fujian, P.R.China

DATE OF BIRTH: April 6, 1975

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, Oregon

South China University of Technology, Guangzhou, Guangdong, P.R.China

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science,
University of Oregon, 2008

Master of Science, Computer and Information Science,
University of Oregon, 2003

Master, Communication and Information System specialty
in Electrical Engineering, South China University of Technology, 1999

Bachelor, Electrical Engineering, South China University of Technology, 1996

AREAS OF SPECIAL INTEREST:

Software Engineering and Requirement Engineering

Formal Modeling and Model Checking

Smart Home System and Ubiquitous Computing

PUBLICATIONS:

Chen, Z., Fickas, S.: Do no harm: model checking eHome applications. In: 1st Int. Workshop on Softw. Eng. of Pervasive Computing Applications, Systems and Environments (SEPCASE '07), Minneapolis, Minnesota, 6 pp (2007), ISBN: 0-7695-2970-4

Fickas, S., Pataky, C., Chen, Z.: DuckCall: tracking the first hundred yards problem. In: Proc. 8th Int. ACM SIGACCESS Conf. on Computers and Accessibility, Portland, Oregon, pp. 283-284 (2006)

Chen, Z., Fickas, S.: The plain old television in a smart apartment. In: Proc. the 1st Int. Conf. on Collaborative Computing: Networking, Applications and Worksharing (CollaborateComm '05), San Jose, California, 8 pp (2005), ISBN: 1-4244-0030-9

ACKNOWLEDGMENTS

I wish to express my greatest thanks to my advisor, Professor Stephen Fickas, for the opportunity to work in his wearable computing lab and for his continuous guidance and support. Without his help, I would not have been able to conduct the research in this manuscript.

In addition, special thanks are due to Professor Michal Young, who often spends extra time to discuss with me, and Professor Yannis Smaragdakis, who sparks some major ideas in this manuscript.

I am grateful to my parents, Qingming Chen and Xiuhua Guo, and my sister, Lurong Chen. Without their spiritual support, it was nearly impossible for me to finish the dissertation.

I wish to thank my many friends in the Department of Computer and Information Science and the University of Oregon, who shared much fun time with me and accompanied me in different periods of the studies.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
II. LITERATURE REVIEW	7
2.1. Application Context	7
2.1.1. The OSGi Specification	7
2.1.2. The Correctness Properties	9
2.1.3. The Knopflerfish Framework	10
2.2. Model Checking	10
2.2.1. Overview	10
2.2.2. Java PathFinder.....	15
2.3. AspectJ	18
III. FRAMEWORK-BASED MODEL CONSTRUCTION	20
3.1. The Programming Model of the OSGi Framework	20
3.2. The Construction of the Modeling Framework	23
3.3. Using the Modeling Framework	31
3.4. Proposed Solutions	37
IV. TAILORING A MODELING FRAMEWORK WITH AOP ASSISTANCE ...	41
4.1. Dilemma of Performance and Modularity	41
4.2. Tailoring the Modeling Framework to the Right Abstraction Level.....	45
V. MODEL CHECKING ASPECTJ PROGRAMS	49
5.1. Native Code	49
5.2. Performance Overhead	53
5.3. Reducing the Performance Overhead	58

Chapter	Page
5.3.1. Atomicity of Aspect-Specific Methods	59
5.3.2. Other Methodologies of Reduction	70
5.3.3. Aspect-Specific Methods in <i>ajc</i>	74
VI. VERIFYING ASPECT-ORIENTED DESIGN PATTERNS	76
6.1. Aspect-Oriented Design Patterns	76
6.2. A Classic Example: Using AOP for Concurrency Control	78
6.3. A Generic Model to Avoid the Aspect Instantiation Problem	88
6.4. Model Checking Aspect-Oriented Design Patterns	91
6.4.1. Observer Pattern	91
6.4.2. Flyweight Pattern	95
6.4.3. Singleton Pattern	98
VII. THE MODELING FRAMEWORK REVISITED	102
7.1. Stability of Models for OSGi Applications	102
7.2. Aspectizing the OSGi Framework at the Model Level	103
7.2.1. FrameworkListener	104
7.2.2. ServiceListener	108
7.2.3. BundleContext Validator	112
7.3. Summary of Improvements	114
VIII. CONCLUSION AND FUTURE WORK	117
8.1. Summary	117
8.2. Lessons to Learn	118
8.3. Future Work	122
BIBLIOGRAPHY	123

LIST OF FIGURES

Figure	Page
1. Layered Structure of the OSGi Framework	8
2. The Operational Model of JPF	17
3. The UML Diagram of the Core OSGi Infrastructure	22
4. The Typical Procedure to Register and Use a Service	25
5. The Pseudo-code to Install a Bundle	27
6. State Transition in a Thread for Each Bundle	28
7. Sample Code of BundleImpl.java at the Model Level	30
8. A Naïve Example of Using a Service	32
9. Error Trace for the Application in Figure 8	32
10. Subscribing Service Events to Avoid the Stale References Problem	34
11. Error Trace for the Application in Figure 10.....	35
12. An Example to Use the ServiceTracker	36
13. Error Trace for the Application in Figure 12.....	37
14. Collaboration between OSGi Applications and the Framework	39
15. A Dilemma of Efficiency and Modularity	44
16. Modularizing the Crosscutting Concerns in Figure 15 with Aspects	47
17. Sample Code in AspectJ Runtime Library that Raises an Exception	50
18. A Reimplementation that Uses only the System Classpath	51
19. The Runtime Library Code that Raises a Runtime Exception	52
20. Peer Method for System.getProperty(String, String)	53
21. A Java Program and the Equivalent AspectJ Program	55
22. The Woven Output (in Java Format) for the <i>issingleton</i> Version	57
23. Aspect Generation Code for <i>issingleton</i> Aspect	62
24. On-the-fly Partial Order Reduction Algorithm in JPF	63
25. The Actual Implementation of Atomic Execution of a Method	64
26. Counting Access to Each Individual Resource	66
27. The Woven Output (in Java) for the <i>perthis</i> Version	68
28. An Implementation of NegativeLineVMLListener	72
29. Adding Atomicity Boundary to Aspect-specific Code	73

Figure	Page
30. An Example to Assure the “Multiple Read, Single Write” Policy	80
31. An Implementation of Lock for Shared Read and Exclusive Write.....	81
32. A Generic Aspect to Enforce the “Multiple Read, Single Write” Policy ...	83
33. Instantiating MultipleReadersSingleWriter Aspect	83
34. Printout of the ReadWriteLock before/after a Lock/Unlock Operation	84
35. The Simplified Error Trace for the AspectJ Program in Figure 33	85
36. Aspect Instantiation (<i>perthis</i>)	87
37. Aspect Instantiation (<i>issingleton</i>)	89
38. A Reimplementation of Figure 32 (without Concurrency Violations)	90
39. A Sample Application Using Aspect-Oriented Observer Pattern	94
40. Error Trace in the Original Aspect-Oriented Observer Pattern	95
41. A Sample Application that Uses Aspect-Oriented Flyweight Pattern	97
42. Error Trace in the Original Aspect-Oriented Flyweight Pattern	98
43. A Sample Application that Uses Aspect-Oriented Singleton Pattern	100
44. Error Trace in the Original Aspect-Oriented Singleton Pattern	101
45. Models of OSGi Applications with the Façade Pattern	103
46. The Aspect to Enforce Observer Pattern for Framework Events 1-39	106
47. The Aspect to Enforce Observer Pattern for Framework Events 40-74	107
48. The Aspect to Enforce Observer Pattern for Service Events 1-36	109
49. The Aspect to Enforce Observer Pattern for Service Events 37-69	110
50. The Aspect to Check the Validity of a Bundle Context	113

LIST OF TABLES

Table	Page
1. A Comparison of Search Space: Java and AspectJ Versions	56
2. Names of the Five Aspect-Specific Methods (<i>perthis</i>)	69
3. Names of the Five Aspect-Specific Methods (<i>pertarget</i>)	70
4. Modifications Needed to Add/Remove a Feature	114
5. Comparison of State Space of Different Configurations.....	115

CHAPTER I

INTRODUCTION

This manuscript is motivated by our experience in building formal models of domain-specific applications. My thesis is that a formal-model framework can be defined that allows efficient models to be generated through a tailoring process via Aspect-Oriented programming (AOP) techniques. To establish my thesis, I will convince the readers that I have established AOP as a proper technique to specialize a modeling framework, in terms of modularity as well as efficiency. I have also identified the Façade Pattern to encapsulate application models from varying details, enabled model checking AspectJ programs in general and proposed useful optimization techniques for the verifications. These arguments are backed by a case study to construct and use a generic, aspectized Open Service Gateway initiative (OSGi) framework, and examples to check Aspect-Oriented design patterns. A formal-model developer may benefit from this manuscript by learning about the advantages and the required changes to use AOP techniques to construct and customize a modeling framework, and an AspectJ programmer may gain insights about the potential traps when using current AOP techniques and the approach to go around them. The rest of this section gives an overview of our work in more detail.

Our research group has actively participated in the research and design of Smart Home applications for cognitively impaired people (Chen 2005; Fickas 2006). We have built a domestic reminder system to allow care providers to enter scheduled events (e.g. daily meds, clinic appointments). The system will play reminders at the appropriate time. We have also built a travel bag to check in their trip items, and provide on-route navigation and emergency help. Due to the very nature of the user group, we are committed to reliability and longevity of our systems, e.g. the device must run continuously without the need to reboot, and acts as it is supposed to in emergency situations. There is a strong need for us to rigidly assure certain correctness properties of our software.

We believe it is beneficial to base our systems on existing standard platforms; in particular, we are interested in the *Open Service Gateway Initiative (OSGi)* (OSGi 2005). The OSGi framework offers a standard way to manage the software lifecycle and more quickly develop applications across platforms. However, we find that the adoption of a standard platform like OSGi doesn't exempt us from concurrency issues. In particular, some known problems, e.g. *the stale references problem* (OSGi 2005), can be caused by the concurrent access to a *service* rendered by the OSGi framework, and are difficult to catch with traditional testing.

We have relied on model checking to uncover software bugs in real systems (Feather 2001; Chen 2007) and would like to use it to check our Smart Home applications. However, we have found that the construction of formal models is an expensive procedure that requires expertise unavailable to typical programmers.

Furthermore, most projects are under time-constraints; managers cannot wait for lengthy model-building efforts before commencing implementation and deployment.

The difficulty in constructing formal models motivates us to seek a way to leverage model-building experience from experts. In particular, since OSGi applications share much commonality in design and implementation, we are interested in whether such similarities will carry over to their counterparts at the model level. It seems natural to build a formal model parallel to the OSGi framework, reuse much of the modeling framework and specialize the modeling framework to construct a formal model for an OSGi application. An interested reader can refer to (Chen 2007, 2008) for our existing work.

This approach is complicated by the very nature of model checking. The specter that haunts formal modeling is *state space explosion*, where the interleaving of process steps leads to an exponential increase of system states that quickly exceeds the capacity of a typical computer. This restriction requires a formal model to be reduced to its *bare essentials* before the verification step. However, since OSGi applications may use different features of OSGi, it is nearly impossible to come up with a modeling framework that has *just* enough details for all OSGi applications. It is thus critical to have the capability to vary features of a modeling framework in a modular way.

The object-oriented nature of Java has offered some help for extending the modeling framework to construct a formal model for an OSGi application. One can specialize the modeling framework with *hooks* and *slots*, in a way similar to using a framework at the application level. However, some of the reduction changes are difficult

to modularize by hooks and slots. For example, in the OSGi domain, when we are interested only in the stale references problem, we don't need permission checks in the framework and may remove all related fields and statements to save the state space. On the contrary, when we want to assure that no malicious application spoils the OSGi framework, we have to add back the missing fields and statements to enable permission checks. Changing the feature of permission checks is a sticky task that breaks the OO modularity: Modifications involve fields and statements scattered in multiple places in different methods, classes and Java files. It is even worse when we vary the combination of features for a particular application. In our experience, such crosscutting concerns prevent us from effective reuse of formal models and incur much overhead in comprehension and maintenance.

In this dissertation, I consider *Aspect-Oriented Programming (AOP)* techniques to handle crosscutting concerns like those described above. The AOP approach adds another unit to modularize the needed changes to a modeling framework. With AOP techniques, we pre-process a generic modeling framework with variations defined in *pointcuts*, *inter-type definition* and other AOP constructs. The pre-processing step adds only the needed details (e.g. fields and statements) to the bare essential. While the needed details may be scattered in different parts of the applications, with AOP techniques, we are able to specify such variations in an explicit and modular way. Varying application details then becomes a simpler task of selecting and weaving the related aspects.

Two problems arise due to the adoption of AOP techniques, namely, native code and performance penalties. *Java PathFinder* (Visser 2003; Mehlitz 2005), the model

checker we use to check Java bytecode, is an explicit state checker that tracks the execution of the JVM, but the execution of native code carries information outside the scope of the JVM. For this matter, we have created a regression test suite in JUnit (JUnit 2007) to discover native code in the bytecode woven by the AspectJ compiler, and have created an abstraction library to resolve the native code in the AspectJ runtime library as needed. The second problem is more severe to the adoption of the AOP approach: in our benchmark testing, we encountered cases that increase the state space several times that of the pure Java implementation for a single usage of aspects. The performance penalty is largely due to the internal variables used by aspect instantiation and subsequent interleaving in a multi-threaded system. We thus make a distinction between *model checking AspectJ applications in general* and *using AOP techniques to vary an existing formal model*. For the latter problem, we would rather ignore the extra interleaving due to the AspectJ weaving: we therefore present several approaches to reduce the search space, namely, creating a specialized *Aspect-Benchmark Compiler* (Avgustinov 2005; ABC 2007) and implementing various search heuristics plugged into the *Java Pathfinder (JPF)*. By such optimizations, we are able to reduce the state space of an AspectJ program to a size comparable with its pure Java counterpart.

We have used the above technologies for rapid construction of formal models for OSGi applications. We were able to uncover several bugs in a set of benchmark OSGi applications, with minimal specialization required.

On the other hand, the extra interleaving of AspectJ programs indicates a potential data race. Using our tool set, we uncovered bugs for general AspectJ programs, including

common programming techniques and Aspect-Oriented design patterns. Some of these bugs are due to the unawareness of the lack of synchronization in the bytecode woven by AspectJ compilers, even though their pure Java counterparts are free from concurrency errors. It is worth noticing that I am able to reproduce these errors on AspectJ programs woven by *ajc* as well as *abc*. These findings are surprising, as some of these programming techniques are widely taught as classic AspectJ examples. In this dissertation, I will explain the detailed cause and show my solution to these problems based on current AspectJ technologies.

This dissertation includes my published, co-authored materials in Chapter III, which describe a unique case study to show the benefits of framework-based model construction and lay down the background and motivation for Chapter IV.

CHAPTER II

LITERATURE REVIEW

For the purpose of explanation, I briefly describe the application context, i.e. the OSGi framework and some sample applications based on it. I also give a brief introduction to the model checker Java Pathfinder, and an overview of AspectJ.

2.1. Application Context

2.1.1. The OSGi Specification

The proliferation of digital home applications brings new challenges for software engineering. Without a common standard, it is difficult to accommodate applications for different devices and platforms. Therefore, many common functions, like device access, application management, etc., have to be created anew for each project. The *Open Service Gateway Initiative (OSGi)* aims at providing a platform independent infrastructure that eases the integration of pre-built, pre-tested components (OSGi 2005).

Figure 1 shows the layered structure of OSGi (modified from (OSGi 2005)). At the bottom level, OSGi runs on top of the Java 2 environment. The Modules layer defines the class loading policies, which, in addition to standard Java classpath, allows a module to provide private classes to another module in a static, declarative way. A pre-built module

is called a *bundle* in OSGi jargon. The Life Cycle layer relies on the Modules layer for class loading, but adds the rules and a set of APIs to install, start, stop, update and uninstall a bundle at runtime. It allows component management that are normally not part of an application. The Services Registry layer uses a service-oriented architecture to enable decoupled, dynamic collaboration among bundles. A service is a special Java object, which is shared by publishing all interfaces it has implemented. A bundle can be notified of the coming and going of services by subscribing to service events delivered by the framework. OSGi is a Service-Oriented architecture: a software vendor provides functionality in terms of services, which can be sold as a pre-built component without the worry of installation and maintenance; the service provider and the service consumer are decoupled so that they may be provided by different vendors. Unlike other layers, the Security layer is entangled with all other layers, assuring that the OSGi framework won't be spoiled by a reckless component from a third-party vendor (OSGi 2005).

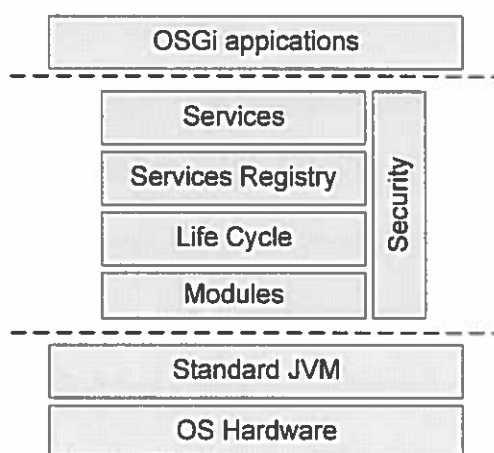


Figure 1. Layered structure of the OSGi framework (modified from (OSGi 2005))

2.1.2. The Correctness Properties

I set out to check correctness properties of the framework as well as its applications. For example, an implementation of the OSGi specification ought to provide the newest driver for a device, check the proper permission for a third-party bundle, catch malicious behaviors, and deliver service events in synchronous (sequential) mode. An OSGi application ought to respect certain conventions in implementing a `BundleActivator`, e.g. it should not block the whole framework. For the purpose of explanation, I introduce a common pitfall in OSGi applications, *the stale reference problem*.

The stale reference problem is due to the indirect reference of a service object for decoupling components. It happens when a service used by a consumer bundle has actually been unregistered by the producer bundle. A stale reference to a service object leads to undefined behaviors, e.g. it may raise runtime exceptions at various execution points, or quietly proceed without actually carrying out the desired functionality. This problem is acknowledged in the OSGi specification, and an auxiliary class, *ServiceTracker*, has been created to track valid services. However, the specification acknowledges that one may still suffer from the same problem even when using a `ServiceTracker` (i.e. due to concurrent execution, theoretically, there are chances that a stale service is retrieved from a `ServiceTracker` object before the service is removed from `ServiceTracker` in response to a “service remove” event), and doesn’t propose any approach completely immune to the stale reference problem. This has decreased the confidence in building highly reliable systems on top of the OSGi framework.

I hypothesize that the modeling framework I built will ease model construction to uncover violations to these correctness properties, so that we are able to minimize the risk to the special user group we target.

2.1.3. The Knopflerfish Framework

There exist several reference implementations for the OSGi specification. *Knopflerfish* is an open source implementation and has been certified to be OSGi compliant (Knopflerfish 2007). It comprises a base framework to fulfill the OSGi specification, mandatory bundles for common core functionalities like input/output and optional bundles for extended functionalities like logging. Other leading, open source implementations include Felix from Apache (Felix 2007) and Equinox from Eclipse (Equinox 2007). An interested reader can refer to (OSGi 2005; Knopflerfish 2007; Felix 2007; Equinox 2007) for a full explanation of OSGi and the Knopflerfish framework.

2.2. Model Checking

2.2.1. Overview

With the increased enhancement of language features for concurrent programming (e.g. the concurrency support in Java) and the wide availability of reactive systems (e.g. wireless sensor networks), concurrent programming takes on a larger and larger portion in software development. However, concurrent programming is notoriously difficult to write correctly, even after careful requirement engineering, rigid code inspection and rigorous software testing. There are numerous accidents caused by such complexity, to

name a few, the lethal overdose accidents of Therac-25 between 1985 and 1987 (Magee 1999), the malfunctioned communication component of NASA's Pathfinder in 1997 (Holzmann 2003), the automated baggage system problem in Denver International Airport between 1993 and 1994 (Wayt 94). Concurrency issues cost opportunity, money, happiness and even lives.

The difficulty of concurrent programming arises due to its very nature: many errors only occur when processes in the concurrent software proceed in a particular order, while the total permutations in different orders can be a huge number far beyond human inspection and/or software testing. Furthermore, some conditions that cause errors are transient and hard to detect and replay.

To assure the quality of concurrent software, formal methods have been proposed. Formal methods rigorously reason about the correctness of the computer programs, often covering each of possible executions. In particular, one of their branches, model checking, has gained wide acceptance for the merits of mechanical verification, easy replay of violation traces and natural specification of a large portion of important properties.

Typically, model checking takes a dual-specification approach: (1) construct a transitional model as the artifact specified in the software, and (2) succinctly specify the desired property as a temporal logic formula. Subsequently, a model checker mechanically checks the transitional model against the temporal formula, and reports a trace to the violation if there is any. In decades, researchers have devised tens of model checkers, to name a few, SPIN (Holzmann 1997; Holzmann 2003), LTSA (Magee 1999), Bogor (Robby 2003), dSPIN (Demartini 1999; Iosif 2001) and Java PathFinder (Visser

2003). There have been numerous reports on the discovery of concurrency flaws by model checkers (Feather 2001; Griesmayer 2005; Markosian 2007; Deng 2003; Holzmann 2003; Magee 1999; Heitmeyer 1998a; Hatcliff 2003; Jackson 2002a).

There are two fundamental problems restricting the practical use of model checkers: *the state space explosion* and *the limited-participants* problems. The first problem arises since the combination of different orders in a complex system can easily be an astronomical number well beyond the storage of modern computers. The second problem arises since a model checker emulates the execution of each process, while some complex systems have no concrete limitation on the number of simultaneous processes allowed (or such limitation is too much for model checkers). Therefore, without other proof measures, e.g. natural deduction, there is no way to assert correctness when no violation is found.

Relevant to the two fundamental problems, usability is a big challenge. In decades, the primary focus of model checking is verification efficiency, due to the state space explosion problem. Usability is less studied and actually becomes a major obstacle for the industrial application of model checking (Barjaktarovic 1998). The following factors contribute to the difficulty.

- Formal specifications are very different from programs. Most specification languages have a natural mapping to their underlying mathematic notions, which typically has very different concepts from modern programming languages like C/C++ and Java. For example, one needs to deal with primitives like non-determinism, liveness, Never in Promela (Holzmann 2003), which are rarely seen for a Java programmer.

- Model checkers are very different from each other. There is no “universal” model checker as there is no general proof procedure, or the problem is undecidable (even first-order logic is undecidable). Moreover, the more expressive a specification language is, the less efficient the verification algorithm tends to be. Therefore, a model checker needs to be a compromise between generality and efficiency, and there is no “universal” specification technique that fits all domains (Lamsweerde 2000).

- There is no or little support for comparative analysis (Lamsweerde 2000). There are no precise criteria and measures for assessing formal specifications written in the same language, not to mention specifications written in different languages. A good specification or a good choice of model checkers hide somewhere deep in the heads of experts.

- Abstraction is critical but very hard. Abstraction discards unimportant details and reduces the reachable states in the verification procedure. Without appropriate abstractions, even a simple program potentially runs out of memory reasonable for a PC. However, there is no universal abstraction technique; all practical abstraction techniques, complete and/or sound, are rather modest compared to the transformations well-trained people can bring up. Logical abstraction remains the most powerful reduction technique, but it requires non-trivial expertise to apply reduction techniques correctly and efficiently.

- There is little reuse on specifications. It is generally agreed that problems in the domain are more likely to be similar than solutions (Lethbridge 2001). Since specification is somewhere between problems and solutions, therefore, specification reuse should be more promising than code reuse (Lamsweerde 2000). However, little has

been done on reuse in model checking. Dwyer et al. proposes temporal formula reuse based on patterns (Dwyer 1999) and Smith et al. generalizes such reuse by parameterized templates (Smith 2002), and there are also case studies (Xie 2003) on reuse of verified components and patterns. However, such work focuses on the reuse of temporal formulas (which is much more restricted) or software components. Reuse of specifications is much more complex but not extensively studied.

In view of the difficulty of model construction, researchers have proposed various techniques and tools for model construction from its implementation counterpart. These include using program analysis techniques (Nielson 1999) to automatically remove fields and statements not related to the correctness criteria (Dwyer 2006; Dwyer 2001), limit the range of variables and the number of participants, and represent values in a more abstract way (Corbett 2000). These techniques are applied before rendering the model to a model checker, and are generally orthogonal to the space reduction techniques adopted in model checkers (e.g. partial order reduction) (Dwyer 2006). However, none of the automation techniques are strong enough to reduce a complex system to a checkable size in the general case (Dwyer 2006, 2001). Often, one has to manually enforce atomicity towards arbitrary code blocks, remove components and/or functionalities, and restrict the scope of variables and the number of participants in the system. To improve the overall coverage, several abstractions may be carried in a trial-and-error style in various runs, i.e. one may apply various combinations of different (possibly lossy) abstractions, and repeat the verification until an error is found or the coverage has reached some (possibly empirical) criteria.

On the other hand, despite the rich literature reporting experience using model checkers to discover bugs in practical design and implementations, model construction is an expertise-intensive procedure and there is little literature to address the methodology of model construction. Taghdiri et. al. reported a case study to build a generic modeling framework for a class of multicast communication protocols, which can be specialized to verify a specific multicast protocol (Taghdiri 2003). Garlan et. al. reported a case study to build a generic modeling framework that can be reused to check event processing based on the Publish-Subscribe design pattern (Garlan 2003). Chen et. al. reported a case study to build a modeling framework of OSGi applications, which eases the model construction by design reuse and code reuse (Chen 2007). However, none of these works addresses the methodology to customize the abstraction level of the modeling framework, which is the focus of this manuscript.

2.2.2. Java PathFinder

Part of my goal in this dissertation is to build a formal model of an OSGi framework, and then check it for concurrency errors. Since the OSGi framework is written in Java, we would rather use a model checker that directly takes Java as input; we can then use much of the implementation directly as the model! Java PathFinder (JPF) (Visser 2003; Mehlitz 2005) is such a model checker. It is an explicit model checker that directly checks Java bytecode and is extensible with different data and search heuristics.

Figure 2 shows the architecture of JPF (Visser 2003). The core JPF has a specialized virtual machine (VM), which takes on the role of a model checking engine for state management (e.g. state cache, query, and restoration). It allows customization of

search strategies, e.g. adding checkers of different properties, gathering execution statistics, by an Observer Pattern that lets concrete listeners subscribe to Search/JVM events exposed by JPF. It also allows plugging a specific Choice Generator to choose and to order choices at a branching point, which influences thread scheduling and non-deterministic data acquisition. Furthermore, JPF allows interception of Java method calls via the Model Java Interface (MJJI), which can be used to resolve native methods and reduce state space. The above extension schemes in JPF allow expert users to experiment with novel search heuristics and state representations, which may lead to significant performance improvements in a specific domain.

There are some restrictions on JPF. First, it is not able to check platform-specific native methods (e.g., System-level service calls, networking APIs, user interaction), since this execution information is not available to the JVM. Second, due to state space explosion, it typically deals with small Java programs no more than 10k lines-of-code (Visser 2003). Therefore, before using JPF to check a real system, one would have to (a) resolve Java native methods with MJJI or replace the native methods with pure Java implementation, (b) apply various reduction techniques to construct an abstract model for the actual system so that the verification may complete within a reasonable time and within a reasonably available main memory. The reduction often includes bounding the number of participants in a model, and model checking only guarantees the correctness of the model with these limited participants. Additional methodologies, like deduction, are required to extend the result to unlimited participants. However, empirical studies show

that most errors can be discovered with a properly selected small number of participants, i.e. the small scope hypothesis (Dennis 2006; Andoni 2003).

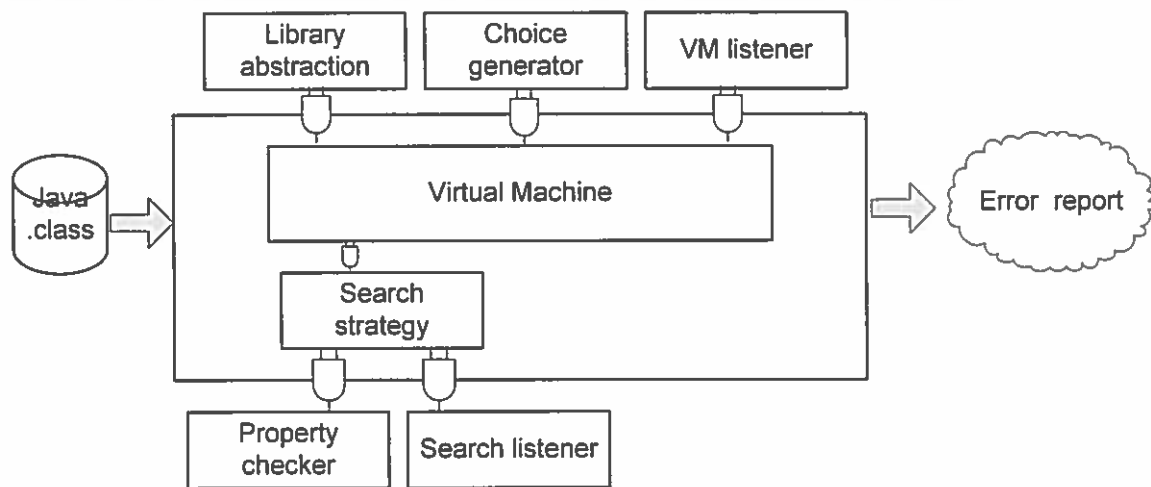


Figure 2. The operational model of JPF (modified from (Visser 2003))

The MJI is the main facility of JPF to help resolve native methods. To resolve a native method via the MJI, a Java class with native methods has two classes for the replacement: a model class and a native peer class. The model class is exposed to the core JPF and thus is state traceable; the native peer class is executed by the host JVM and is not state traceable. JPF associates the model class and the native peer class using a name mangling scheme similar to the Java Native Interface (JNI), i.e. using the model class package name and class name to deduce and match the native peer class name. Although the host JVM doesn't directly change the system states in the core JPF, methods in the native peer are passed a reference to the execution environment of the core JPF and may

access and alter the JPF object model. The naming convention and the environment reference passing in JPF are analogous to the Java Native Interface (JNI).

In addition to resolving native methods, the MJI is also used to intercept JPF system level functionality and for system space reduction. The interception is required for standard library classes in the package `java.lang`, as they have to affect JPF internal class and thread model. For example, the MJI is used to intercept thread scheduling, which is critical to explore all interleaving in model checking. The MJI can also reduce the state space, since it can hide all changes in the intermediate steps and only cause traceable state transitions when objects in the environment reference are explicitly modified.

2.3. AspectJ

In Object-Oriented languages, it is natural to organize programs in units of classes. However, there are certain design concerns that involve code in multiple execution points and are difficult to be modularized using Object-Oriented techniques. For example, feature variations typically involve fields and statements in different methods and classes, breaking encapsulation units of slots and hooks. It is thus desired to have a mechanism to vary features in a modular way, e.g. enable or disable a feature by changing a single variation point.

Aspect-oriented techniques are invented to modularize such variations, and AspectJ is an aspect-oriented extension to the Java programming language. In the jargon of AspectJ, design concerns that span multiple classes are called *crosscutting concerns*, and

the execution of a method call is a *join point*. An aspect is a unit of management in AspectJ, analogous to a class in Object-Oriented languages, but it has the capacity to modularize crosscutting concerns that is not available in common OO languages.

An aspect encapsulates several new constructs to modularize crosscutting concerns: *pointcuts*, *advices* and *inter-type declarations*. A *pointcut* specifies the pattern to pick out join points and may be composed of other pointcuts. It typically involves matching of static method signatures and runtime context. An *advice* is the code that gets executed when the associated pointcut is matched. For a particular pointcut, an advice may be invoked at different stages of method execution, e.g., *before*, *after* and *around*. An *inter-type* declaration may statically change class hierarchy and/or add class members.

An AspectJ compiler compiles and weaves an AspectJ program into normal Java classes. The two mainstream AspectJ compilers (*ajc* from eclipse, and *AspectBench Compiler*, or *abc*, from Oxford University and McGill University) both additionally require a small (~180K) runtime library. A nice feature of AspectJ is that it is designed as a compatible extension to Java, i.e. all legal Java programs are legal AspectJ programs, and all byte code generated by legal AspectJ programs run on the standard Java virtual machine. This is crucial for our hypothesis to introduce AspectJ to formal modeling, since JPF reads only standard Java byte code. An interested reader can refer to (Colyer 2004; AJC 2007; ABC 2007) for more about AspectJ.

CHAPTER III

FRAMEWORK-BASED MODEL CONSTRUCTION

In this chapter, we describe a case study that shows the benefits of framework-based model construction, i.e. design reuse and code reuse. Some materials in this Chapter are extracted from my previous co-authored work (Chen 2007). We focus on the modeling framework construction and its strength in this chapter; in the next chapter, we show the problems that bar the wide usage of framework-based model construction and present our solution to the problems.

3.1. The Programming Model of the OSGi Framework

For the Knopflerfish framework, we create a formal model parallel to the application framework at an abstract level, using the following manual approach. We follow the call-graph reachability to remove unrelated code (e.g. fields, statements and classes). Based on domain knowledge, we make high-level decisions to remove non-essential components, e.g. system bundles. We also revise the presentation of the framework status in a more abstract way, and resolve native code when needed. We briefly describe the procedure to create such a specification (Chen 2007).

To lay out the discussion, we use the UML diagram in Figure 3 to explain the core architecture of an OSGi framework. The two slots, `start()` and `stop()`, in the

interface `BundleActivator`, are the variation points of a bundle application: they shall be filled by an OSGi application to implement the desired functionalities. These two functions are called back while starting and stopping a bundle, respectively. Meanwhile, they are passed an instance of `BundleContext`, which carries the context information in the OSGi framework. From the perspective of the framework, the class `Framework` has access to all internal information. For example, it has a reference to `Bundles`, which maintains a list of all installed bundles; it has a reference to `Listeners`, which maintains a list of listeners interested in various events, using an Observer design pattern; it also has an instance of `Services`, which maintains a list of service registration stubs. The service lists often include collections of `[service_name, service_registration]` tuples and support various query methods to find the proper services. Such indirect access to services decouples the service *consumer* from the service object (and the service *provider*), improving reusability and allowing them to be separately developed. Each installed bundle may be serialized to the main memory; a `BundleImpl` is then instantiated to describe the bundle, and the `BundleActivator` defined in the manifest file is instantiated and started through reflection. An OSGi application may register a service: during such registration, an instance of `ServiceRegistration` is created to describe a service when it is registered, including information like the name of the service and the registering bundle.

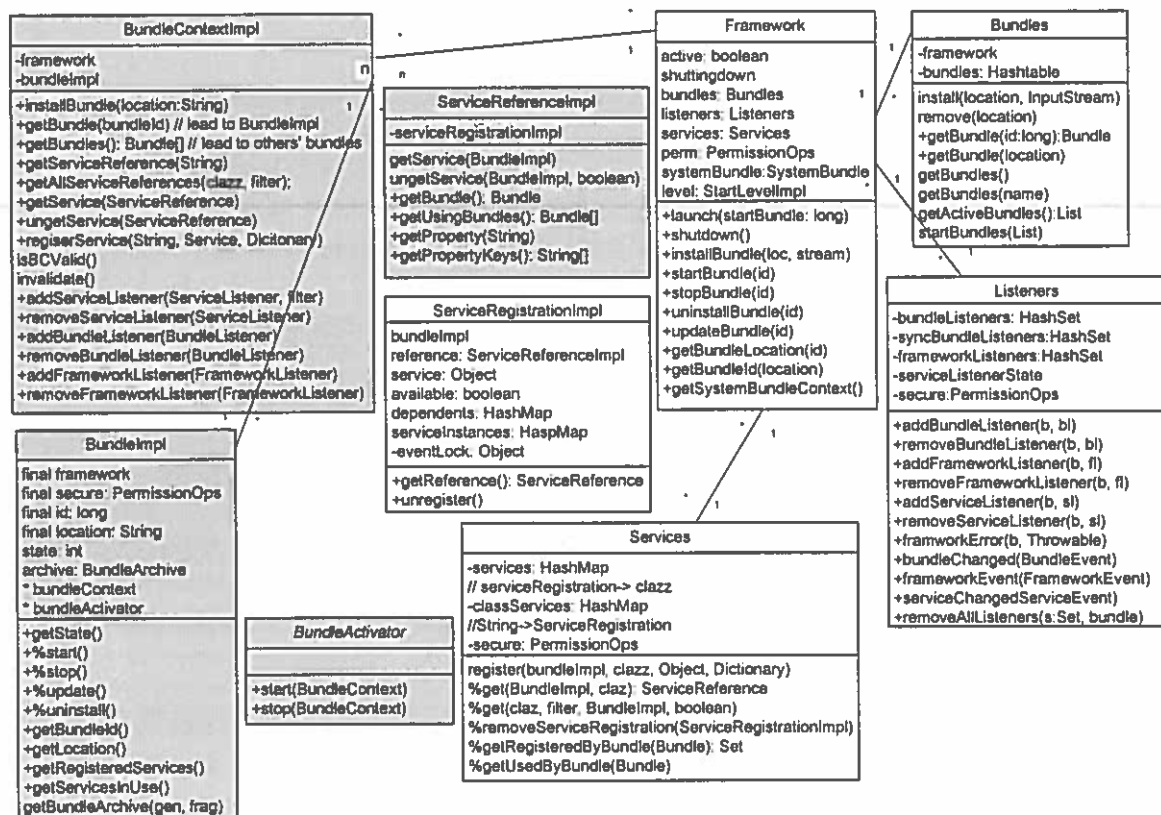


Figure 3. The UML diagram of the core OSGi infrastructure

It is worthy to point out that the Façade design pattern is used to encapsulate the internals of the OSGi framework: the four classes in **grey** color are those instances accessible to a regular bundle; other classes in white color are not visible to a regular bundle. For example, to be immune from a mal-functioning bundle, the OSGi framework refrains from exposing `ServiceRegistrationImpl` to a *consumer* bundle; instead, the class `ServiceReferenceImpl` is created as a façade for `ServiceRegistrationImpl`. The consumer bundle can only access restricted information and functionalities about a service through the published interfaces. For

example, the registering bundle can use `ServiceRegistration` to unregister a service, but a `ServiceReference` retrieved by a consumer bundle through query may not be used to unregister a service. Similarly, `BundleContextImpl` acts as a façade for a regular bundle to carry out a subset of functionalities (e.g. to register a service, to subscribe to a particular event) in `Framework`, `Bundles`, `Services` and `Listeners`. The Façade design pattern enables encapsulation, which benefits us not only at the application level (e.g. to reduce framework vulnerability and to ease OSGi applications development) but also at the modeling level, as we shall see later.

3.2. The Construction of the Modeling Framework

We describe our efforts in constructing a modeling framework in this section. Our target is to build a generic, abstract framework, which can be used by plugging in OSGi applications and checking them against a wide spectrum of correctness criteria. These correctness criteria shall catch the core functionalities of an OSGi framework, e.g. bundle management, service management and listener management. Since our modeling framework is based on the Knopflerfish implementation, this procedure includes removing fields and statements from the application framework, resolving native code by the MJI scheme and/or rewriting in pure Java, rewriting the model in an abstract way, and creating a closed execution environment for the OSGi applications.

Figure 4 shows the sequence diagram of a user case to install and start a GPS bundle, register and provide a GPS service, and use the GPS service from a consumer bundle. It catches the fundamental functionalities in an OSGi framework and the main

operations that we want to support in the modeling framework. A GPS bundle (i.e. a jar file) that provides the *Global Positioning Service (GPS)* service is first installed in the OSGi framework. During the installation, an instance of `BundleImpl` is created to describe the bundle information; after the installation, a bundle event (`bundle_installed`) is broadcast to all bundle event subscribers. After a bundle is installed, the bundle can be started. Starting a bundle includes the following actions: create an instance of `BundleContext` to gather the needed environment information for the OSGi application; broadcast a bundle event (`bundle_starting`) to all bundle event subscribers; use class reflection to create an instance of the specific implementation of `BundleActivator` (i.e. `GPSServiceActivator` implements `BundleActivator`); invoke the `start(BundleContext)` method of the instance of `BundleActivator`, which registers a GPS service in the OSGi framework and causes a service event (`service_registered`) broadcast to all service event subscribers. Upon the reception of the `service_registered` event, a GPS service consumer can query the GPS service and uses the GPS service. Meanwhile, a bundle event (`bundle_started`) is broadcast to all bundle event subscribers. These steps are common in starting and using a service.

A registered service may be explicitly unregistered by the registering bundle, or implicitly unregistered when the registering bundle is about to be stopped. Figure 4 also shows the latter case. When the GPS bundle is about to stop, a bundle event (`bundle_stopping`) is broadcast to all bundle event subscribers, then `GPSServiceActivator.stop(BundleContext)` is invoked, followed by the

actions to unregister all active services that have been registered by the GPS bundle. At this moment, a `service_unregistered` event is broadcast to all service event subscribers. Finally, a bundle event (`bundle_stopped`) is broadcast to all bundle event subscribers.

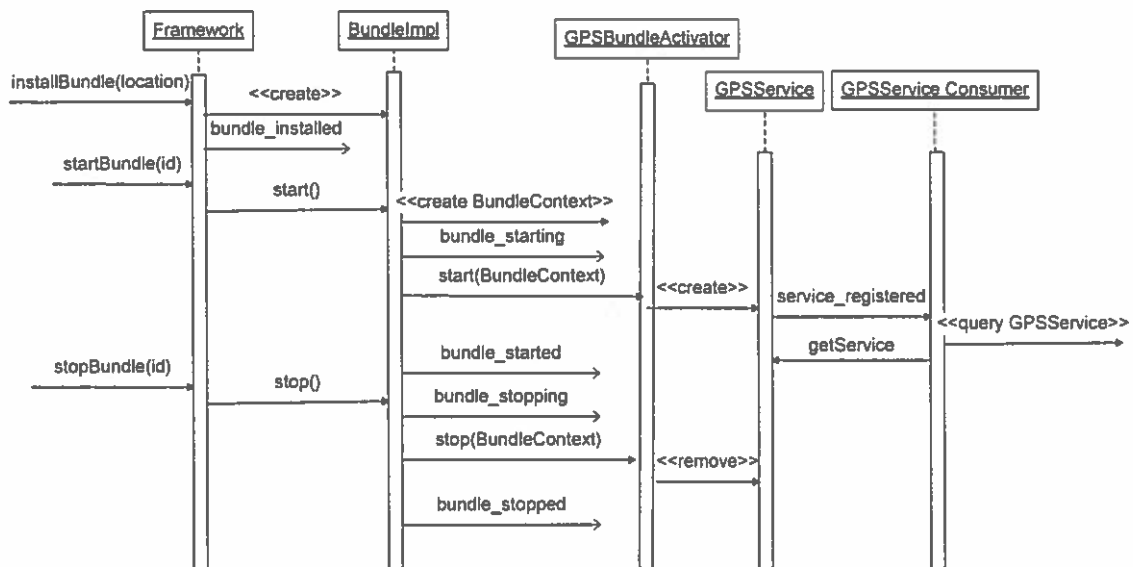


Figure 4. The typical procedure to register and use a service

After modeling the fundamentals of the OSGi frameworks, we show the procedure to mimic the installation of a bundle at the model level. The pseudo code (after slicing unrelated code) is shown in Figure 5. It is worth pointing out that the pseudo code in Figure 5 has causality to the bundle installation thus is not removed by reachability analysis. However, some of them may be removed by arbitration of the model developer. For example, the above procedure involves native code to serialize a bundle jar from

some URL, parse the jar file and instantiate proper entities (e.g. instances of `BundleImpl`, `BundleActivator` and `BundleArchive`). We may resolve all native code via MJI and keep the framework APIs unchanged, but it is fairly involved in the actual development and memory-inefficient in the runtime (e.g. all contents in a jar file are saved in each state). Instead, we ignore the manifest file and require additionally passing two parameters for the instantiation: a static unique counter to mimic the `Bundle` location (the location is required for bundle storage in the framework), and the class name for the `BundleActivator` for class reflection. Other instances can be created from such information. We use a unique integer counter to abstractly represent system time; we also ignore security features, since we focus on violations caused by the concurrent access and such violations won't be disabled due to the removal of permission checks. These modifications complete the changes to model the bundle installation.

To check the component management, we also need to mimic the environmental inputs to drive the system. We show an example to simulate user inputs to manage the framework bundles. As mentioned, input/output is typically taken through utility components in the Knopflerfish framework. It includes Desktop bundle, Log bundle, Console bundle, etc. These components are in the control flow of device/bundle management and thus won't be removed by reachability analysis. However, it is fairly involved to resolve native methods with various input/output streams, and extremely intricate to model interactions with the user interface. Fortunately, for our purpose, we are interested in the effects of management (i.e. install, start, stop, etc.), but not how to manage components. Therefore, we can reasonably identify these components and ignore

the low level details like reading strings from a buffered input stream, reading objects from a socket stream, or button events in a graphical user interface.

```
1. BundleImpl install(location) {
2.   return if the bundle is already installed
3.   check permission
4.   download jar file if needed
5.   read jar file into memory // insertBunldeJar()
6.   parse jar file
7.   construct and store bundle archive
8.   set static classpath
9.   check execution environment
10.  create bundleImpl
11.  check bundle admin permission // AdminPerm
12.  set last modified time for the bundle
13.  add bundleImpl to bundles
14.  broadcast bundle_install event
15.  return bundle impl;
16.}
```

Figure 5. The pseudo-code to install a bundle

As mentioned, each bundle runs in a separate environment. To model its management, we create a thread for each bundle, and non-deterministically choose a proper management action according to its current state. The following pseudo code shows the main transition for the thread:

The pseudo code in Figure 6 corresponds to the state diagram of a bundle in the OSGi specification (OSGi 2005). A bundle is initially installed and changed to RESOLVED state (line 3). There a user can update a bundle with a new bundle jar to stay at RESOLVED state (line 5), start a bundle to enter ACTIVE state (which further allows

a user to stop a bundle to return to RESOLVED state, in line 6), or uninstall a bundle to return to UNINSTALLED state (line 8). It differs from the state diagram in that it doesn't explicitly show the internal transitions automatically carried out by the base framework (e.g. from INSTALLED to RESOLVED, from STARTING to ACTIVE). These automatic transitions are hidden in the function calls. By the non-deterministic choices, we have abstractly specified possible user inputs to manage bundles in the Knopflerfish framework.

In the rest of this chapter, we show some fragments of the actual Java code, to give a comparison of the application code and the model code and lay down examples that will be used throughout our discussion.

```
1. public void run() {
2.     while (true):
3.         fw.installBundle(location);
4.         while (true): do one of the following non-det.
5.             fw.updateBundle(location);
6.             fw.startBundle(bundleId); fw.stopBundle(bundleId);
7.             break;
8.         fw.uninstallBundle(location);
9. }
```

Figure 6. State transition in a thread for each bundle

Figure 7 shows a fragment of `BundleImpl.java`. Fields and statements that are removed from the application code are marked in `grey`. The underlying reasoning to remove these fields is: `secure` is removed since we are not checking permissions while executing management actions; `classLoader` and `oldClassLoaders` removed

since JPF supports a single classpath at runtime after all; we ignore version control in the OSGi framework thus `version` and `v2Manifest` are removed; we decide not to check fragmentation functionality thus we remove the fields `fragment` and `attachPolicy`. Correspondingly, in the method `BundleImpl.start()`, we remove statements that check permission for bundle management, that handle fragmented bundles, that delay the starting of a service according to the start level and that create a proprietary classpath for the current bundle. At this moment we carry out such removals manually; an automatic slicing tool like `Bandera` could surely help in the construction of the abstract model, but it still requires human judgment, e.g. specify the slicing criteria and create abstract representation for the bundle serialization.

Similarly, at an abstract level, we mimic the procedure to start, stop, upgrade and uninstall a bundle, and the procedure to start and stop a framework. Other abstraction and resolution include the removal of service cache, simplified property representation with services, emulation for installed `BundleArchive`, etc. A full explanation of the construction is beyond the scope of this dissertation. An interested reader can get the formal model at (Chen 2008).

```

class BundleImpl {
    Framework framework;
    ProtectionDomain pd;
    long id;
    String location;
    int state;
    BundleArchive archive;
    int generation;
    PermissionOps secure;
    ClassLoader classLoader;
    long lastModified;
    boolean v2Manifest;
    String symbolicName;
    boolean singleton;
    Version version;
    BundlePackage bpkgs;
    private HashMap oldClassLoaders;
    FileTree bundleDir;
    bDelayedStart=false;
    ArrayList fragments;
    String attachPolicy;
    Fragment fragment;

    synchronized public void start() {
        check permission for bundle management
        if this bundle is a fragment
            throw exception;
        if there is start level service and the current level is not high
        enough
            delay starting;
        switch (the current state) {
            case INSTALLED: throw error
            case RESOLVED:
                state = STARTING
                broadcast a bundle starting event
                create an associated BundleContext
                check permission for starting a bundle
                instantiate a GlassLoader for the current bundle
                parse bundle descriptor to get the BundleActivator
                instantiate the BundleActivator using the name
                invoke start() method of activator instance
                state = ACTIVE
                broadcast a bundle started event
            case ACTIVE: return;
            case STARTING, STOPPING, UNINSTALLED:
                throw error
        }
        ...
    }
}

```

Figure 7. Sample code of BundleImpl.java at the model level

3.3. Using the Modeling Framework

To use the modeling framework to verify an OSGi application, we also need to specify the correctness criteria. As an example, we focus on verifying the stale reference problem in this section.

There are two types of the stale references problem, i.e. one is that the reference achieved has become *null*, and the other happens when the reference is not *null* but points to an *obsolete* object, which hasn't been garbage collected due to alias references. The former throws a `NullPointerException`, while the latter may or may not throw an exception and leads to an undefined status. In view of this, we use the following pattern to detect the second error: add a boolean mark `valid` to the attributes of a service class; set `valid` to true when registering the service object, and reset it to false when unregistering the service object; assert `valid` to be true whenever invoking a function of the service object. In this way, we will be able to catch both types of the stale references problem via a single criterion of JPF, `NoUncaughtExceptionsProperty`.

Thereafter, we apply the modeling framework to check real examples and show violations we have found. To our best knowledge, some of these errors are not reported elsewhere.

We look at a series of tutorial examples from Oscar (Oscar): a producer bundle registers a dictionary service, and a consumer bundle leverages the dictionary service to look up a word (i.e. `DictionaryService.checkWord()` is called). We are interested to see whether the consumer bundle will use a stale dictionary service.

We first check an implementation that has a known bug. The following pseudo code is the main part of interest, adopted from the actual implementation:

```
1. ... producer bundle registers DictionaryService
2. ServiceReference() refs=context.getServiceReferences(...);
3. if (refs != null) {
4.   DictionaryService dictionary =
      (DictionaryService) context.getService(refs(0));
5.   if (dictionary.checkWord(word)) {
6.     ...
```

Figure 8. A naïve example of using a service

Without surprise, when we instantiate threads in Figure 6 with the two bundles in Figure 8, JPF reports a `NullPointerException` as shown in Figure 9 (in a simplified format). The exception is due to the interference of the producer bundle: after the consumer thread gets a non-null `ServiceReference`, the producer thread unregisters the dictionary service so that the query to the dictionary service returns null.

```
1. ...producer registers DictionaryService
2. ServiceReference() refs=context.getServiceReferences(...);
3. if (refs != null) {
4.   ... producer unregisters DictionaryService
5. DictionaryService dictionary =
      (DictionaryService) context.getService(refs(0));
6. if (dictionary.checkWord(word)) {
7. JPF reports NullPointerException
```

Figure 9. Error trace for the application in Figure 8

To avoid this problem, the tutorial proposes several scenarios to go around it. One approach is to subscribe service events by implementing `ServiceListener` and track the status of the dictionary service, as shown in Figure 10.

In the revised scenario, two variables are created to hold service information and are shared between threads: `m_ref` is designed to hold the service reference to the dictionary service, and `m_dictionary` is a cache to the dictionary service object. The hook `ServiceChanged()` is called by the base framework when a service event occurs: it sets `m_ref` and `m_dictionary` by looking up the service repository in the base framework when a service is registered, and refreshes `m_ref` and `m_dictionary` when a service is unregistered (i.e. set `m_ref` and `m_dictionary` to null and reassign the service with the highest rank to them).

```

(1) Use dictionary service in the consumer thread:
1. context.addServiceListener(this,...);
2. ServiceReference() refs=context.getServiceReferences(...);
3. if (refs != null) {
4.     m_ref = refs(0);
5. m_dictionary=(DictionaryService)context.getService(m_ref);
6. }
7. if (m_dictionary!=null) {
8.     if (m_dictionary.checkWord(word)) {
9.         ...

(2) Implement ServiceListener.serviceChanged():
10. if (event.getType()==ServiceEvent.REGISTERED) {
11.     if (m_ref == null) {
12.         m_ref = event.getServiceReference();
13.         m_dictionary=(DictionaryService)
            context.getService(m_ref);
14.     }
15. } else if (event.getType()==
            ServiceEvent.UNREGISTERING) {
16.     if (event.getServiceReference() == m_ref) {
17.         context.ungetService(m_ref);
18.         m_ref = null;
19.         m_dictionary = null;
20.         ServiceReference() refs=
            context.getServiceReferences(...);
21.         if (refs != null) {
22.             m_ref = refs(0);
23.             m_dictionary=(DictionaryService)
                context.getService(m_ref);
24.         }
25.     }
26. }

```

Figure 10. Subscribing service events to avoid the stale references problem

However, JPF reports an error trace when we check the above application, as shown in Figure 11.

```
...producer thread registers DictionaryService
(1.3) if (refs != null) {
(1.4)   m_ref = refs(0);
...producer thread unregisters DictionaryService
...base framework calls serviceChanged(UNREGISTERING)
(2.18) m_ref = null;
(2.19) m_dictionary = null;
(1.5)   m_dictionary =
(DictionaryService)context.getService(m_ref);
...JPF reports NullPointerException: call getService() on null
pointer..
```

Figure 11. Error trace for the application in Figure 10

The error is the result of concurrent access to `m_ref` by the three threads: the producer/consumer bundle, and the base framework. It happens when a valid service reference `m_ref` is about to be used to query a service in the framework, the base framework resets `m_ref` and subsequently causes the query to fail (i.e. the service reference may not be null when looking up a service in the base framework).

The result is somewhat surprising because this scenario has been advertised in a tutorial to tackle the stale references problem, and is actually used in many real projects. For example, by the same modeling framework, we have found a similar problem between Console bundle and Desktop bundle included in the Knopflerfish distribution. Such examples are evidence of the severity of concurrency errors in practice and the effectiveness of our modeling framework to tackle a class of these errors.

Interestingly, the stale references problem is known to the OSGi alliance, and a helper class, `org.osgi.util.tracker.ServiceTracker`, is included in the OSGi specification to help track valid services (OSGi 2005). It implements a fairly

involved service listener to track registered services. However, the specification doesn't propose any approach completely immune to the stale references problem. In fact, it acknowledges that one may still suffer from the same problem even when using a `ServiceTracker`. The tutorial in the Knopflerfish framework (Knopflerfish 2007) gives an example to leverage this helper, as follows (it is slightly modified by adding line 4 to avoid a shallow `NullPointerException` when there is no dictionary service):

```
1. ServiceTracker tracker = new ServiceTracker(...);
2. tracker.open();
3. DictionaryService s=(DictionaryService)tracker.getService();
4. if (s!=null) {
5.     s.checkWord (...);
```

Figure 12. An example to use the `ServiceTracker`

For the above implementation, JPF reports an assertion exception as in Figure 13 (in an abstract format). This error is caused by the second type of the stale references problem: the reference to the `DictionaryService` object is valid till line 4; however, after the producer bundle unregisters the dictionary service, it actually points to an obsolete object and behaves in an unpredictable way, e.g. it may throw exceptions due to using non-existing resources.

```
...producer bundle registers DictionaryService
1. ServiceTracker tracker = new ServiceTracker(...);
2. tracker.open();
3.DictionaryService s=(DictionaryService)tracker.getService();
4. if (s!=null) {
...producer bundle unregisters DictionaryService
5.   s.checkWord (...);
6. JPF reports an assertion error
```

Figure 13. Error trace for the application in Figure 12

It is worth pointing out that we adapt the models in the above examples from real OSGi applications: the modifications mainly involve the removal of unrelated statements and the removal of exception try-catch block to expose exceptions. Since the programming interface of the regular OSGi applications (i.e. no system bundles, not touching the framework core infrastructure) are created with a *Facade design pattern*, and we keep these APIs unchanged, developers can create a model for an OSGi application like developing the OSGi application itself, and thus should have little difficulty in building new models for OSGi applications or adapting their applications into models. These modeling efforts are minimal compared with the construction of the modeling framework.

3.4. Proposed Solutions

In this section, we discuss potential approaches to address the stale references problem, and use the formal model to explore candidate solutions.

The stale references problem is caused by the very nature of the indirection and lookup scheme of services in the OSGi specification: there is no assurance that a

reference (i.e. a pointer) to a service object can be cached for future usage. To guarantee no stale references, there ought to be a scheme that guarantees the validity of a service object throughout all of the period of its invocation, i.e., (1) a service reference is valid (2) the use of the service object is exclusive (i.e. the service can't be unregistered during this period) (3) the use of the service is short (i.e. cause no blocking), to avoid hanging the whole system.

In the above three conditions, (3) has to be guaranteed by developers who develop the consumer bundle. We show paradigms to assure (1) and (2).

In Figure 14, we define a class `BasicService` that is to be extended by any service class. It requires the collaboration of the framework and the consumer bundle. In the framework, while unregistering a service, `service_lock` assures the exclusive access to the service object, and the service validity can be faithfully reported by calling `service.isValid()`. We can leverage this infrastructure to correct the three flawed examples in the previous section.

```
(1) A service to be extended by all services:
1. class BasicService {
2.   public volatile boolean valid = false;
3.   public Object service_lock = new Object();
4.   public final void setValid(boolean b) {
5.     valid = b;
6.   }
7.   public final boolean getValid() {
8.     return valid;
9.   }
10.}

(2) Collaboration in Framework:
unregister() {
  synchronized (service.service_lock) {
    service.setValid(false);
    ...proceed unregistration like original framework...
  }
}
```

Figure 14. Collaboration between OSGi applications and the framework

Solution 1: For the example in Figure 8, we add synchronization on `service_lock` and check service validity (i.e. `service!=null && service.getValid()`) before invoking a service's function.

We check this modified model with the modeling framework and find no error. The paradigm to use a service in this example is, always exclusively access a service and get a fresh copy of a service each time.

Solution 2: For the example in Figure 10, we set the framework's option to broadcast an `unregister_service` event before actually unregistering a service, and add synchronization on `service.service_lock` and check service validity before invoking a service's function.

We check this modified model with the modeling framework and find no error. The paradigm for this example is, with a service listener, always exclusively access a service and assure the `unregister_service` event is delivered and processed synchronously and before actually unregistering in the framework.

Solution 3: For the example in Figure 12, we add synchronization on `service_lock` and check service validity before using a service achieved from a `ServiceTracker`.

We check this modified model with the modeling framework and find no error. The idea for this example is, with a `ServiceTracker`, always exclusively retrieve a service and check its validity at each time of invocation.

There are subtle issues in the above three paradigms. For example, although they are all free from deadlock in our verification, they need fine-grained arrangement in synchronization, which may be difficult in a complex application. Also, getting a fresh copy of a service by looking up in the framework each time carries an overhead in performance. It goes beyond the scope of this dissertation to propose a generic, fine-grained concurrency control protocol for the service management.

CHAPTER IV

TAILORING A MODELING FRAMEWORK WITH AOP ASSISTANCE

In this section, I point out the inflexibility and crosscutting concerns using only Object-Oriented techniques when tailoring a modeling framework, with examples from the OSGi domain. Thereafter, I show that one can modularize these crosscutting concerns with Aspect-Oriented techniques, thus boosting the usability of a modeling framework.

4.1. Dilemma of Performance and Modularity

In section 3, we have shown that our modeling framework can be applied to OSGi applications to discover the stale reference problem. However, we arbitrarily decided not to check the permission before executing a bundle. The question is, what modifications to the modeling framework do we need to make when we want to add this feature?

On the other hand, the overall model based on the modeling framework is not necessarily the most efficient that a developer can reasonably come up with. There are fields and statements that are required for a particular application but totally unrelated to another application. For example, we have included the module for bundle listener management and bundle event dispatch in the modeling framework; however, an OSGi application not subscribing to the bundle events (like the example in Figure 8) doesn't

need the bundle listener module presented in the modeling framework. One of such compromises is shown in Figure 15.

Figure 15 shows some fragments of three Java classes, `BundleImpl`, `Framework` and `Listeners`. They are part of the core infrastructure of the modeling framework: `BundleImpl.start()` describes the internal changes when starting a bundle, including checking execution permission, invoking `BundleActivator.start()` callback and broadcasting bundle starting events; `BundleImpl.stop()` describes the internal changes when stopping a bundle, including checking execution permission, invoking `BundleActivator.stop()` callback and broadcasting bundle stopping events; `Framework.secure` stores the security policy, and `Listeners.bundleListeners` stores a list of mapping between bundles and their listeners. In particular, the fields and statements in the **bold** font are only required when the security feature is desired, and the fields and statements highlighted with **grey color** are only required when the bundle event influences the OSGi application. Since the state space explosion problem is so severe, we would rather remove these fields and statements unless they are really required. However, as shown in the above, the relevance of fields and statements vary according to the applications, thus there is no way to define a modeling framework with *just* enough details. For the simple example in Figure 15, we need to make 8 modifications for the security feature and 5 modifications for the bundle listener routines in different classes; for a particular OSGi application, we will potentially make modifications in $5+8=13$ places for each of the $2^2=4$ possible combinations. When the full modeling framework (core infrastructure only) is

considered, we have 23 modifications for the security feature, 16 modifications for the bundle listener feature, and we will have to make $23+16=39$ modifications for each of the 4 possible combinations. Even worse, for a framework with rich features like OSGi, it is a daunting task to make modifications for a particular combination of features: we have identified other features like the framework listener feature, the service listener feature, the bundle context validator. Each of these features involve modifications similar to the above two features; therefore, for a particular application, we have to make about 90 potential modifications for one of the $2^5=32$ possible combinations of features. And again, these modifications involve fields in different classes and statements in different methods and classes, break the encapsulation of an application framework and make the framework difficult to understand and reuse. In practice, the developer faces a dilemma: whether to use the modeling framework in a white-box style and have scattered code here and there, or to use the modeling framework in a black-box style but with more details than necessary. We set out to study this problem in the rest of this dissertation.


```

class BundleImpl {
    PermissionOps secure;
    ...
    synchronized public void start() {
        check permission for bundle management;
        switch (current state) {
            case INSTALLED: throw error;
            case RESOLVED:
                state = STARTING;
                broadcast a bundle starting event;
                create an associated BundleContext;
                check permission for starting a bundle;
                instantiate the BundleActivator;
                invoke start() method of activator;
                state = ACTIVE
                broadcast a bundle started event;
            case ACTIVE: return;
            case STARTING, STOPPING, UNINSTALLED:
                throw error;
        }
    }
    synchronized public void stop() {
        check permission for bundle management;
        switch (current state) {
            case INSTALLED, RESOLVED:
                check permission to set persistent;
            case ACTIVE:
                check permission to stop a bundle;
                state = STOPPING;
                broadcast a bundle stopping event;
                invoke stop() method of activator;
                state = RESOLVED
                broadcast a bundle stopped event;
            case STARTING, STOPPING, UNINSTALLED:
                throw exception;
        }
    }
}
class Framework {
    PermissionOps secure;
    ...
    Framework(Object m) {
        initialize secure by security policy;
        ...
    }
}
class Listeners {
    HashSet bundleListeners = new HashSet();
    ...
}

```

Figure 15. A Dilemma of efficiency and modularity

4.2. Tailoring the Modeling Framework to the Right Abstraction Level

The addition of fields and statements according to the needed features can be modularized by Aspect-Oriented techniques. The hypothesis is that we can use *inter-type definition (ITD)* to add the missing fields, pick *joinpoints* to catch the execution points to update the system states and use *advices* to carry out the actual updates. Thereafter, we follow the general steps in the below to carry out framework-based formal modeling:

- 1) Define a generic modeling framework with the least details
- 2) Define each feature of the system in an aspect
- 3) Choose a particular combination of features by including the corresponding aspects for weaving.
- 4) Specialize the modeling framework with slots, hooks and advices.

We now show the procedure of how we apply the above techniques to build formal models for OSGi applications based on a generic modeling framework. We first aspectize the example in Figure 15, as shown in Figure 16.

In Figure 16, the aspect `BundleListenerAJ` (line 1-16) modularizes the scattered code for the feature of bundle listening. Using ITD, it declares a field `bundleListener` that has been removed from the class `Framework` in the original framework, which holds a collection of mappings between bundles and their listeners. The pointcut `Monitor_BundleStateChange` catches all changes to the bundle states, i.e. a bundle state is represented as the field `BundleImpl.state`. When a bundle state is changed, various bundle events are broadcast: if the bundle state is set to

`Bundle.STARTING` (line 7), a bundle `STARTING` event is broadcast by calling the routine `bundleChanged(BundleEvent)` (i.e. `bundleChanged()` is a helper function that is moved from the original framework into `BundleListenerAJ`, which is omitted in Figure 15); if the bundle state is set to `Bundle.ACTIVE`, a bundle `STARTED` event is broadcast to notify all subscribers that the subject has been started; if the bundle state is set to `Bundle.STOPPING`, a bundle `STOPPING` event is broadcast to notify subscribers that the subject is being stopped; if the bundle state is set to `Bundle.RESOLVED`, a bundle `STOPPED` event is broadcast to notify subscribers that the subject has been stopped.

The aspect `PermissionAJ` (line 17-39) modularizes the changes needed for the security feature. It inserts the field `perm` back to the class `Framework` (line 18), to represent the security policy. The security policy is initialized in the framework constructor (line 21). The pointcut `Monitor_BundleStartStop` catches the joinpoints when a bundle is about to start or stop, and the corresponding advice calls the security policy's `checkexecutionAdminPerm()` method to verify the permission for execution. The pointcut `Monitor_BundleStartPersistent()` catches the joinpoints when a bundle is set to persistent, and the corresponding advice calls the security routines to carry out the actual verification. The pointcut `Monitor_BundleStart0()/Monitor_BundleStop0()` catches the joinpoint when a bundle is actually started/stopped, and the corresponding advice calls a security to check the execution permission.

```

1. public privileged aspect BundleListenerAJ {
2.   private HashSet bundleListeners = new HashSet();
3.   pointcut Monitor_BundleStateChange(BundleImpl b, int nstate):
4.     set(int BundleImpl.state) && args(nstate) && target(b);
5.   after(Bundle b, int nstate): Monitor_BundleStateChange(b, nstate) {
6.     BundleEvent be;
7.     if (nstate==Bundle.STARTING) {
8.       bundleChanged(new BundleEvent(BundleEvent.STARTING, b));
9.     } else if (nstate==Bundle.ACTIVE) {
10.      bundleChanged(new BundleEvent(BundleEvent.STARTED, b));
11.    } else if (nstate==Bundle.STOPPING) {
12.      bundleChanged(new BundleEvent(BundleEvent.STOPPING, b));
13.    } else if (nstate==Bundle.RESOLVED) {
14.      bundleChanged(new BundleEvent(BundleEvent.STOPPED, b));
15.    }
16.  } }
17. public aspect PermissionAJ {
18.   PermissionOps perm = null;
19.   pointcut FConstructor(Object m): call (Framework.new(Object)) &&
20.   args(m);
21.   after(Object m): FConstructor(m)
22.   { initialize the secure policy; }
23.   pointcut Monitor_BundleStartStop(BundleImpl bundle):
24.     (call (void start()) || call (void stop())) && target(bundle);
25.   before(BundleImpl bundle): Monitor_BundleStartStop(bundle)
26.   { perm.checkExecuteAdminPerm(bundle); }
27.   pointcut Monitor_BundleStartPersistent(BundleImpl bundle):
28.     execution (void startOnLaunch(..)) && this(bundle);
29.   before(BundleImpl bundle): Monitor_BundlePersistent(bundle)
30.   { check permission to set persistent; }
31.   pointcut Monitor_BundleStart0(BundleImpl bundle):
32.     execution (void BundleImpl+.start0()) && this(bundle);
33.   before (BundleImpl bundle): Monitor_BundleStart0(bundle)
34.   { check permission to actually start a bundle; }
35.   pointcut Monitor_BundleStop0(BundleImpl bundle, boolean b):
36.     execution (BundleException BundleImpl+.stop0(boolean))
37.     && this(bundle) && args(b);
38.   before (Bundle bundle, boolean b):
39.   Monitor_BundleStopACTIVE(bundle, b)
40.   { check permission to actually stop a bundle; }
41. }

```

Figure 16. Modularizing the crosscutting concerns in Figure 15 with aspects

When we need a variation of the modeling framework that supports the security feature, we simply include the aspect `PermissionAJ` in AspectJ weaving, which is as

trivial as a command-line option. Similarly, we can generate a variation of the modeling framework that supports the bundle listening feature by including the aspect `BundleListenerAJ` in AspectJ weaving. Overall, to enable a feature, we simply add the corresponding aspect to AspectJ weaving. In this way, a model developer doesn't have to know about the internals of the modeling framework, and is able to use the modeling framework in a black-box style.

CHAPTER V

MODEL CHECKING ASPECTJ PROGRAMS

Two concerns arise due to the very nature of AspectJ programs when model checking AspectJ programs, namely, native code and performance overhead. This chapter explores the causes and solutions to these concerns.

5.1. Native Code

The concerns of native code arise due to two factors: 1) The aspect weaving uses *reflection* to gain contextual information and create proper pointcuts designators. 2) AspectJ programs require a runtime library, which may invoke *native methods* that are not directly checkable with JPF. We have to address these concerns before we are able to check an AspectJ program with JPF.

Some factors mitigate the severity of the concerns. First, the aspect weaving uses *reflection* mainly at the weaving time, rather than the runtime. Second, the runtime library of AspectJ programs is relatively small (~180K bytes) and only a small portion of it invokes native methods. Therefore, it is possible to create a small abstraction library to check AspectJ programs. While it seems trivial to resolve such “accidental” invocation of system-level functions, JPF can’t check AspectJ programs without such step. In this

section, we briefly show some examples to demonstrate the necessity and the approach to construct an abstraction for the runtime library.

In Figure 17 it shows an example of the AspectJ runtime library that causes a runtime exception to JPF. The exception happens when the statement in the **grey** area is executed: the reason is that `java.lang.ClassLoader.loadClass(String)` is a system-level function invocation and JPF can't track such information. While theoretically it can be implemented via MJI, it is fairly involved and one cannot reasonably expect a typical programmer to be able to do so. Instead, we can simply ignore the different classpath and stick to the default system class loader, as shown in Figure 18. The function `java.lang.Class.forName(String)` is also a system-level invocation; however, JPF already provides its peer implementation in MJI form, which creates an instance of the class and registers the instance in the runtime environment of JPF's host VM.

```
Class makeClass(String s) {
    if (s.equals("")) return null;
    Class ret = (Class)prims.get(s);
    if (ret != null) return ret;
    try {
        ClassLoader loader = getLookupClassLoader();
        if (loader == null) {
            return Class.forName(s);
        } else {
            return loader.loadClass(s);
        }
    } catch (ClassNotFoundException e) {
        return ClassNotFoundException.class;
    }
}
```

Figure 17. Sample code in AspectJ runtime library that raises an exception

```
Class makeClass(String s) {
    if (s.equals("")) return null;
    Class ret = (Class)prims.get(s);
    if (ret != null) return ret;
    try {
        return Class.forName(s);
    } catch (ClassNotFoundException e) {
        return ClassNotFoundException.class;
    }
}
```

Figure 18. A reimplementaion that uses only the system classpath

In the rest of this section, we show an example that implements a peer method via MJI. Consider the following code from `org.aspectbench.runtime.internal` in the *ajc* runtime library: The function `set()` in Figure 19 invokes `java.lang.System.getProperty(String, String)`, which reads the named property from the operating system. Again, such function invocation breaks the close environment of JPF and needs to be taken care of. No peer method for `getProperty(String, String)` is implemented; therefore, the invocation to the `set()` method will cause a runtime exception to JPF (this is accurate as of 03/2007; the current release of JPF does provide another implementation by pre-fetching all needed information through configurations).

We can use the MJI scheme in JPF to create a peer method to be executed in place of `System.getProperty(String, String)`, as shown in Figure 20. The peer class follows the naming convention of Java Native Interface (JNI). After the peer class is

installed, a peer method gets executed when JPF decides that its associated model method is invoked. In this case, when `System.getProperty(String, String)` is called, the host VM will execute `JPF_java_lang_System.getProperty__Ljava_lang_String_2Ljava_lang_String_2__Ljava_lang_String_2()` correspondingly. In particular, the host JVM gets the environment information (line 8) and stores the result in a special area (line 10), or stores a default value in the special area if the fetched information is null (line 11-12). Thereafter, JPF reads the result from this special area as if it is read from the system environment. After the installation of this peer class, JPF reports no exception when it invokes `System.getProperty(String, String)`.

```
private static boolean set() {
    if (System.getProperty(
        "org.aspectbench.DontUseCflowThreadLocal",
        "false").equals("true"))
        return false;
    ...
}
```

Figure 19. The runtime library code that raises a runtime exception

As a result to date, we have created an MJI abstraction library and customized the AspectJ runtime library, to help resolve native methods directly or indirectly introduced by AspectJ. This covers the basic AspectJ primitives except *cflow*, *cflowbelow* and *percflow* constructs. We also leverage the integrated JUnit test utilities in JPF to create

a regression test set, to verify that all AspectJ primitive constructs (except *cflow*-related constructs) are checkable with JPF.

```
1. public static int getProperty__Ljava_lang_String_2Ljava
   _lang_String_2__Ljava_lang_String_2 (MJEnv env, int clsObjRef, int
   keyRef, int defRef) {
2.     int r = MJEnv.NULL;
3.     if (keyRef != MJEnv.NULL) {
4.         String k = env.getStringObject(keyRef);
5.         String defaultString = env.getStringObject(defRef);
6.         if (k==null)
7.             return MJEnv.NULL;
8.         String v = System.getProperty(k);
9.         if (v != null)
10.            r = env.newString(v);
11.        else if (defaultString!=null) {
12.            r = env.newString(defaultString);
13.        }
14.    return r;
15.}
```

Figure 20. Peer method for `System.getProperty(String, String)`

5.2. Performance Overhead

The performance overhead is a more severe concern when adopting AOP techniques for model checking. It is reflected as the different search space of between a Java program and its AspectJ counterpart. In addition to fields and statements inserted as intended (i.e. the same fields and statements as in their pure Java counterpart), the aspect weaving introduces instances of aspects and extra statements that have no correspondence in its Java counterpart. These fields and statements are required for internal use, such as aspect management and contextual information. These additions

may increase the number of reachable states and raise the risk to run out of resources (time and memory) before the verification completes. The performance overhead can be demonstrated with the following code fragments.

Figure 21 shows a Java program `TwoThreadResourceJava.java` and a corresponding AspectJ program `TwoThreadAJ.aj` that are driven by the skeleton `TwoThread.aj`. The intention of the program, as demonstrated by the Java program, is to keep track of the usage of the shared resource: the counter is increased when entering a critical section, and decreased when exiting the critical section. The AspectJ version, `TwoThreadResourceAspectJ`, creates a *joinpoint* designator before and after the critical section, and advises increasing and decreasing the counter when appropriate (for simplicity, we assume that there is only one resource and one counter, and use *issingleton* version of aspects).

Model checking the above two programs reveals that the state space is increased due to AOP techniques. Using the number of new states as the measurement of state space, from Table 1, we can see that the *issingleton* version increases the state space by 19% (i.e. 194 new states vs. 163 new states), while the *perthis* version nearly triples the state space (i.e. 465 new states vs. 163 new states).

```

class TwoThreadResourceJava implements TwoThreadResourceInt {
    int value=0;
    void inc() { value++; }
    void dec() { value--; }
    public void consume() {
        inc();
        dec();
    }
}

public class TwoThread extends Thread {
    TwoThreadResourceInt resource;
    public TwoThread(TwoThreadResourceInt e){
        resource = e;
    }
    public void run() {
        resource.consume();
    }
    public static void main(String args()) {
        TwoThreadResourceInt resource;
        TwoThread t1 = new TwoThread(resource);
        TwoThread t2 = new TwoThread(resource);
        t1.start();
        t2.start();
    }
}

```

```

public class TwoThreadResourceAspectJ implements
TwoThreadResourceInt {
    public void consume() { }
}

public aspect TwoThreadAJ {
    public int TwoThreadResourceAspectJ.value=0;
    void TwoThreadResourceAspectJ.inc() { value++; }
    void TwoThreadResourceAspectJ.dec() { value--; }
    pointcut pc_consume(TwoThreadResourceAspectJ resource):
        execution (public void consume()) && this(resource);
    before(TwoThreadResourceAspectJ resource): pc_consume(resource) {
        resource.inc();
    }
    after(TwoThreadResourceAspectJ resource): pc_consume(resource) {
        resource.dec();
    }
}

```

Figure 21. A Java program and the equivalent AspectJ program

Table 1. A comparison of search space: Java and AspectJ versions

Java version	AspectJ (<i>issingleton</i>)	AspectJ (<i>perthis</i>)
states: new=163, visited=178, backtracked=340, end=31 instructions: 3930	states: new=194, visited=220, backtracked=413, end=33 instructions: 4917	states: new=465, visited=588, backtracked=1052, end=51 instructions: 13324

The root of increased state space can be speculated by inspecting the woven bytecode. With *-dava* option, the *abc* compiler allows the output in Java source code. A snapshot of the woven Java source code for the *issingleton* version is shown in Figure 22.

From Figure 22, we can see that apart from intended code defined in advice, extra variables, statements and methods are inserted for aspect management. The extra statements do not increase the state space as long as they don't access a shared variable, thanks to the *partial order reduction (POR)* algorithm used in JPF (i.e. when POR is enabled, if a bytecode instruction is not scheduling relevant, JPF continues execution without creating a new state). However, some statements do access a shared variable. For example, the singleton aspect provides a public static method `aspectOf()` to allow the access of the aspect instance via a singleton pattern, which is stored as a public static variable `abc$issingletonInstance`. The `aspectOf()` is invoked from the `consume()` method, as a need for aspect management. Furthermore, the `consume()` method is invoked from both threads, as part of the driving skeleton. Therefore, `abc$issingletonInstance` is shared among two threads, and the particular

statements that read/write it will incur additional interleaving. Consequentially, there are more states in the aspect version than its Java counterpart.

```

public class TwoThreadAJ {
    public static final TwoThreadAJ abc$issingletonInstance;
    private static Throwable abc$initFailureCause;
    public static TwoThreadAJ aspectOf() throws
org.aspectj.lang.NoAspectBoundException {
    TwoThreadAJ theAspect;
    theAspect = abc$issingletonInstance;
    if (theAspect != null) {
        return theAspect;
    }
    throw new NoAspectBoundException("...", abc$initFailureCause);
}
...
}
public class TwoThreadResourceAspectJ implements
myabc.perform.auxiliary.TwoThreadResourceInt {
    public void consume() {
        TwoThreadAJ r1;
        r1 = null;
        try {
            r1 = TwoThreadAJ.aspectOf();
            this.myabc$perform$TwoThreadAJ$inc$6();
        } catch (Throwable r2) {
            if (r1 == null) {
                TwoThreadAJ.aspectOf();
            }
            this.myabc$perform$TwoThreadAJ$dec$8();
            throw r2;
        }
        this.myabc$perform$TwoThreadAJ$dec$8();
    }
}
...
}

```

Figure 22. The woven output (in Java format) for the *issingleton* version

The increase of the state space depends on the degree of extra interleaving. For AspectJ programs with *perthis* and *pertarget*, the number of states can be doubled or

even tripled. Therefore, it is worthy to look at the methodologies to reduce the performance overhead due to the adoption of AOP techniques.

5.3. Reducing the Performance Overhead

Ideally, while enjoying the modularity provided by AOP techniques, we don't want to sacrifice the performance since the state space explosion is already a severe problem. We have come up with several methodologies to reduce the extra interleaving introduced by aspect weaving. In this section, we address the methodology we have adopted and briefly discuss several other possibilities to argue for our approach.

Theoretically, each extra statement added by AOP techniques has the potential to increase the state space. An intuitive approach to reduce the states is to make each contiguous, aspect-specific execution as an atomic block. This doesn't change the semantics of the Java programs, namely, it should render as many errors as the Java version. There are several heuristics to decorate the atomicity blocks, for example, using the line number for a bytecode, or modify the AspectJ compiler to directly specify the atomicity blocks. These approaches are tied to the implementations of a particular AspectJ compiler.

Another general approach is based on the observation that under the POR analysis, an inserted statement won't increase the state space as long as the corresponding bytecode instructions are not scheduling relevant. Therefore, as long as we are able to force aspect-specific bytecode instructions as scheduling irrelevant, there is no increase of state space. In our observation, a major source of interference is aspect-specific shared

variables, which may be excluded from the POR analysis in two approaches: directly exclude them from the POR analysis, or execute the methods that access them atomically.

We will focus on the last methodology in this section, which has been adopted in our case studies.

5.3.1. Atomicity of Aspect-Specific Methods

For convenience, we call a class field that is irrelevant to application code but used only internally for aspect management as *aspect-specific variables*. Similarly, we call a method that contains no application-specific code but statements that are internally used by the aspect management *aspect-specific methods*. As mentioned, aspect-specific variables are the major source of extra interleaving; our goal is to eliminate the extra interleaving caused by *aspect-specific variables*.

Since aspect-specific methods carry no application code, executing them atomically won't omit errors caused by the application code. The remaining questions are: 1) Are aspect-specific variables accessible only through aspect-specific methods? 2) Can we pre-determine aspect-specific methods before model checking? 3) Can we execute aspect-specific methods atomically during model checking? The first two conjectures can be validated by inspecting aspect generation code in the AspectJ compiler and the AspectJ programs after weaving. We answer these three questions in this section, mainly based on the *abc* compiler and verified by the *ajc* compiler.

To be shared among threads, an aspect-specific variable must be a class field. They are generated for the aspect management in several situations, as discussed below.

issingleton An aspect may be defined as *issingleton* (by default), where an instance of the specific aspect is created and accessed through a singleton pattern. Figure 23 shows the source code from the *abc* compiler that generates an *issingleton* aspect.

From Figure 21 and Figure 22, we can see that for an *issingleton* aspect, only two field variables are generated, `abc$issingletonInstance` and `abc$initFailureCause`. The variable `abc$initFailureCause` is used to cache an exception thrown during the static aspect class initialization so that it can be re-thrown thereafter. If we ignore the passing of the exception instance (i.e. this assumption is reasonable, since it is less common to share the exception instance elsewhere, and it is often deemed as a bad practice to let exception-handling code change the control flow), the only source for extra interference is the variable `abc$issingletonInstance`.

From Figure 21 and Figure 22, we can further see that the only aspect-specific methods generated to access the aspect instance `abc$issingletonInstance` are `aspectOf()`, `hasAspect()` and the static aspect initialization method. Since the static class initialization is always synchronized, we don't have to worry about its access to the shared aspect instance; we only need to make sure that the other two methods, `aspectOf()` and `hasAspect()`, are executed atomically.

Figure 22 also answers the second question, "Can we pre-determine aspect-specific methods before model checking?": for an *issingleton* aspect, the *abc* compiler always generates these two aspect-specific methods with the same names. To tell JPF that these methods are to be executed atomically, we need to identify them in a JPF configuration file `jpf-attribute` as:

```
<full-class-name> aspectOf true  
<full-class-name> hasAspect true
```

The `<full-class-name>` is the canonical class name for the woven aspect, including its package and class name. In general, they can be expressed as a regular expression that can be recognized by `gov.nasa.jpf.jvm.ConfigAttributor` (one has to configure JPF to set `vm.attributor.class = gov.nasa.jpf.jvm.ConfigAttributor` instead of the default attributor). This attributor will be checked when JPF loads a class, so that JPF can set an *atomic* mark for each method in the class.

The atomic execution of a method that has been marked as *atomic* can be implemented as a VM listener through the JPF extension scheme. The algorithm relies on JPF's implementation of the on-the-fly partial order reduction, as shown in Figure 24.

Figure 24 shows the current implementation of the on-the-fly partial order reduction algorithm in JPF. It uses a global variable, `isFirstStepInsn`, to indicate whether the current bytecode instruction is the first instruction to be executed after the last step. Upon the execution of a bytecode instruction, if the instruction is the first instruction, it will be always executed; or, if it is scheduling relevant and not excluded from the POR analysis, a choice generator is created and JPF breaks the inner loop, marking the end of the execution step (so that a potentially new state will be reached if not within an atomic block); otherwise, this instruction is either scheduling irrelevant or has been explicitly excluded from the POR analysis, so it should be executed without breaking a POR analysis step.

```

public void fillInSingletonAspect(Aspect aspct) {
    SootClass cl = aspct.getInstanceClass().getSootClass();
    SootField instance = new SootField("abc$issingletonInstance", cl
        .getType(), Modifier.PUBLIC | Modifier.STATIC | Modifier.FINAL);
    cl.addField(instance);
    SootField instance2 = new SootField("abc$initFailureCause", RefType
        .v("java.lang.Throwable"), Modifier.PRIVATE | Modifier.STATIC);
    cl.addField(instance2);
    generateSingletonAspectOfBody(cl);
    generateSingletonHasAspectBody(cl);
    generateSingletonClinitBody(cl); }
private void generateSingletonAspectOfBody(SootClass cl) {
    if (Modifier.isAbstract(cl.getModifiers())) return;
    SootMethod aspectOf = cl.getMethod("aspectOf", new ArrayList());
    Body b = Jimple.v().newBody(aspectOf);
    aspectOf.setActiveBody(b);
    SootClass nabe = Scene.v().getSootClass(
        "org.aspectj.lang.NoAspectBoundException");
    ...
    StaticFieldRef ref = Jimple.v().newStaticFieldRef(
        Scene.v().makeFieldRef(cl, "abc$issingletonInstance",
            cl.getType(), true));
    Chain units = b.getUnits();
    units.addLast(Jimple.v().newAssignStmt(theAspect, ref));
    Stmt newExceptStmt = Jimple.v().newAssignStmt(nabeException,
        Jimple.v().newNewExpr(nabe.getType()));
    Stmt ifStmt = Jimple.v().newIfStmt(
        Jimple.v().newEqExpr(theAspect, NullConstant.v()),
        newExceptStmt);
    units.addLast(ifStmt);
    Stmt returnStmt = Jimple.v().newReturnStmt(theAspect);
    units.addLast(returnStmt);
    units.addLast(newExceptStmt);
    List typelist = new LinkedList();
    typelist.add(RefType.v("java.lang.String"));
    typelist.add(RefType.v("java.lang.Throwable"));
    SootMethodRef initthrowmethod = Scene.v().makeConstructorRef(nabe,
        typelist);
    StaticFieldRef causefield = Jimple.v().newStaticFieldRef(
        Scene.v().makeFieldRef(cl, "abc$initFailureCause",
            RefType.v("java.lang.Throwable"), true));
    Stmt assigntocause = Jimple.v().newAssignStmt(failureCause,
causefield);
    units.addLast(assigntocause);
    List arglist = new LinkedList();
    arglist.add(StringConstant.v(cl.getName()));
    arglist.add(failureCause);
    ...
}

```

Figure 23. Aspect generation code for *issingleton* aspect (from (Avgustinov 2005))

```
1. do {
2.   if not in atomic block {
3.     advance current choice generator
4.     context switch if next choice is a thread choice
5.   }
6.   isFirstStepInsn = true;
7.   do {
8.     fetch next instruction
9.     notify vm listener to execute next instruction
10.    executeInstruction(JVM);
11.    notify vm listener next instruction executed
12.    if (a choice generator is created) {
13.      break;
14.    } else {
15.      isFirstStepInsn = false;
16.    }
17.  } while (there are more instructions);
18. } while (atomicLevel > 0);
19.
20. executeInstruction() {
21.   if (isFirstStepInsn && isSchedulingRelevant() && field is not
22.     excluded) {
23.     set the current choice generator
24.     return;
25.   }
26.   actually execute the instruction
27. }
```

Figure 24. On-the-fly partial order reduction algorithm in JPF
(Summarized from source code in (Visser 2003))

We can simply implement the atomic execution of an “atomic” method based on the existing infrastructure of the POR analysis and JPF’s search extension scheme, as shown in Figure 25.

```

1. public class AtomicMethodVMLListener implements VMLListener {
2.     public void executeInstruction(JVM vm) {
3.         Instruction insl = vm.getLastInstruction();
4.         MethodInfo method = insl.getMethodInfo();
5.         ThreadInfo thread = vm.getCurrentThread();
6.         if (method.isAtomic()) {
7.             if (!method.isClinit()) {
8.                 String name = method.getCompleteName();
9.                 if (name.startsWith("java.") || name.startsWith("javax.") ||
10.                    name.startsWith("sun.")) {
11.                     if (atomic_standard) {
12.                         thread.isFirstStepInsn = true;
13.                     }
14.                 } else {
15.                     thread.isFirstStepInsn = true;
16.                 } } }
17.     }

```

Figure 25. The actual implementation of atomic execution of a method

JPF is extensible with the Observer design pattern: one can implement the `gov.nasa.jpf.jvm.VMLListener` interface (e.g. `AtomicMethodVMLListener` in Figure 25) and adds it as a listener to the host VM of JPF. JPF will instantiate the customized `VMLListener` and invoke callback functions at certain interesting points, e.g. before/after the execution of a bytecode instruction, before/after the advancement of a search. For the class `AtomicMethodVMLListener` in Figure 25, the method `VMLListener.executeInstruction(JVM vm)` is invoked right before the actual execution of a bytecode instruction (line 9-10, Figure 24). If a method is marked as atomic (line 6), we mark each of its bytecode instruction as the first instruction in an execution step, to force the atomic execution until it reaches a bytecode instruction of a non-atomic method (line 11, 14). There is some subtleness though: line 7 excludes such atomicity for static class initialization since it is inherently synchronized, and line 8-12

allows the flexibility to exclude some Java routines from atomic execution (this is implementation-specific: JPF, by default, marks all methods in standard Java libraries as atomic).

With the mark of the atomic methods and the plug-in of the AtomicMethodVMLListener, run JPF to re-check the model in Figure 21, then we get the following result:

```
states: new=163, visited=178, backtracked=340, end=31
instructions:      4747
```

From this result, we can see that the *issingleton* version has the same number of new states and visited states as its Java counterpart!

perthis An aspect may be decorated as *perthis*, where an instance of the specific aspect is created and associated with each object that is the currently executing object at any of the specific pointcut. Figure 26 shows a re-implementation of the two threads example in Figure 21, with the *perthis* modifier.

For the two threads problem, such implementation differs in that an instance of the aspect is created and associated for each resource, namely, a counter only tracks the access of the associated resource instead of all resources.

```

public aspect TwoThreadPerThisAJ
perthis(pc_consume(TwoThreadPerThisResourceAspectJ)) {
    public int TwoThreadPerThisResourceAspectJ.value=0;
    void TwoThreadPerThisResourceAspectJ.inc() {
        value++;
    }
    void TwoThreadPerThisResourceAspectJ.dec() {
        value--;
    }
    pointcut pc_consume(TwoThreadPerThisResourceAspectJ resource):
        execution(public void consume()) && this(resource);
    before(TwoThreadPerThisResourceAspectJ resource):
pc_consume(resource) {
    resource.inc();
}
    after(TwoThreadPerThisResourceAspectJ resource):
pc_consume(resource) {
    resource.dec();
}
}

```

Figure 26. Counting access to each individual resource

Figure 27 shows the Java source code after weaving the *perthis* aspect in Figure 26. A class field, `myabc_perform_TwoThreadPerThisAJabcPerThisField`, is added to store the instance of the aspect. This is the shared variable that causes the extra interleaving and the subsequent increased states shown in Table 1. In the generated code, this field is only accessed (read and/or write) by five methods: a *get* method, a *set* method, a static *aspectOf* method, a static *hasAspect* method and a static *bind* method. The *get/set* method protects the field from direct access like a normal Object-Oriented program. The static *bind* method is used to initialize the aspect instance for a specific object (i.e. a resource in the two threads example). The static *aspectOf* method returns the corresponding instance of a particular object. The static *hasAspect* method checks whether the particular object has already had an associated instance of the aspect. The

static *bind* method creates and associates an instance of the aspect to a specific object. Furthermore, these five methods are routines for aspect management and carry no application code, namely, they are *aspect-specific* methods as defined. Therefore, if we execute these methods atomically, the Aspect-Oriented implementation should be as efficient as their Java counterpart.

Inspecting the *abc* compiler's source code responsible for *perthis* aspect generation confirms our speculation in the above. Furthermore, there are patterns to name the five methods. As shown in table 2, the *aspectOf* method, the *hasAspect* method, and the *bind* method have the fixed regular expressions regardless of the aspect's name, while the *get/set* methods are prefixed with the full name of the aspect class (with "." replaced with "_").


```

public class TwoThreadPerThisAJ implements
myabc.perform.TwoThreadPerThisAJ$abc$PerThis {
    private transient TwoThreadPerThisAJ
myabc_perform_TwoThreadPerThisAJ$abc$PerThisField;
    public static TwoThreadPerThisAJ aspectOf(Object theObject$17)
throws org.aspectj.lang.NoAspectBoundException {
        TwoThreadPerThisAJ perInstance$21;
        if (theObject$17 instanceof TwoThreadPerThisAJ$abc$PerThis) {
            perInstance$21 = ((TwoThreadPerThisAJ$abc$PerThis)
theObject$17).myabc_perform_TwoThreadPerThisAJ$abc$PerThisGet();
            if (perInstance$21 != null) {
                return perInstance$21;
            }
        }
        throw new NoAspectBoundException();
    }
    public static boolean hasAspect(Object theObject$22) {
        if (theObject$22 instanceof TwoThreadPerThisAJ$abc$PerThis &&
((TwoThreadPerThisAJ$abc$PerThis)
theObject$22).myabc_perform_TwoThreadPerThisAJ$abc$PerThisGet() !=
null) {
            return true;
        }
        return false;
    }
    public TwoThreadPerThisAJ
myabc_perform_TwoThreadPerThisAJ$abc$PerThisGet() {
        return myabc_perform_TwoThreadPerThisAJ$abc$PerThisField;
    }
    public void
myabc_perform_TwoThreadPerThisAJ$abc$PerThisSet(TwoThreadPerThisAJ
fieldloc$4) {
        myabc_perform_TwoThreadPerThisAJ$abc$PerThisField = fieldloc$4;
    }
    public static void abc$perThisBind(Object theObject$26) {
        TwoThreadPerThisAJ$abc$PerThis castedArg$27;
        if (theObject$26 instanceof TwoThreadPerThisAJ$abc$PerThis) {
            castedArg$27 = (TwoThreadPerThisAJ$abc$PerThis) theObject$26;
            if
(castedArg$27.myabc_perform_TwoThreadPerThisAJ$abc$PerThisGet() ==
null) {
                castedArg$27.myabc_perform_TwoThreadPerThisAJ$abc$PerThisSet(new
TwoThreadPerThisAJ());
            }
        }
    }
    ...
}

```

Figure 27. The woven output (in Java) for the *perthis* version

Table 2. Names of the five aspect-specific methods (*perthis*)

Type	Pattern	Examples
get	<aspect>\$abc\$PerThisGet	myabc_TwoThreadPerThisAJ\$abc\$PerThisGet
set	<aspect>\$abc\$PerThisSet	myabc_TwoThreadPerThisAJ\$abc\$PerThisSet
aspectOf	aspectOf	aspectOf
hasAspect	hasAspect	hasAspect
bind	abc\$perThisBind	abc\$perThisBind

If we configure JPF to mark these five methods as atomic and use the `AtomicMethodVMLListener` we created in the previous section, we get the following result:

```
states: new=163, visited=178, backtracked=340, end=31
instructions: 8000
```

Compared with the statistics in Table 1, now we have reduced the search space to the same size as its Java counterpart, which is about 1/3 of the search space without such reduction.

pertarget An aspect may also be decorated as *pertarget*, where an instance of the specific aspect is created and associated with each object that is the target of the joinpoint of any pointcut.

The aspect generation code for *perTarget* aspect in the *abc* compiler is almost identical to the code for *perThis* aspect: it also inserts a class field to store an instance of the aspect, and adds the five aspect-specific methods to access the aspect instance (Table 3 shows the patterns for the names of the five aspect-specific methods for *perTarget* aspect). Similarly, after we mark these methods as atomic in JPF's configuration file, we are able to reduce the state space to its Java counterpart.

Table 3. Names of the five aspect-specific methods (*perTarget*)

Type	Pattern	Examples
get	<aspect>\$abc\$PerTargetGet	myabc_TwoThreadPerTargetAJ\$abc\$PerTargetGet
set	<aspect>\$abc\$PerTargetSet	myabc_TwoThreadPerTargetAJ\$abc\$PerTargetSet
aspectOf	aspectOf	aspectOf
hasAspect	hasAspect	hasAspect
bind	abc\$perTargetBind	abc\$perTargetBind

5.3.2. Other Methodologies of Reduction

We have also tested other methodologies to reduce the performance overhead caused by AOP techniques. We briefly describe their theory, implementations and cons here, which serve as a justification of our approach in the previous section.

“Hiding” the aspect instance In the previous discussion, our goal is to reduce the extra interleaving caused by the addition of the aspect instance. One may immediately come up with another approach to reduce the performance overhead: exclude the added aspect instance from the POR analysis. This can be as simple as a small modification in JPF’s configuration file: add the aspect instance to the exclude fields of the POR analysis.

This approach is simple and totally removes the state transitions when accessing the aspect instance. However, there are certain circumstances where we don’t want to ignore such interferences. For example, each aspect class has a public, static `aspectOf()` method to retrieve the instance of the associated aspect (i.e. it is the only aspect instance for *issingleton* aspect, or the per-object aspect instance for *perthis* and *pertarget* aspect). One may enforce synchronization rules based on the retrieved aspect instance: blindly excluding the aspect instance from the POR analysis may lead to omission of some reachable states and potential errors. An interested reader can refer to (Hannemann 2002) to see such an example when implementing an Aspect-Oriented Observation Pattern.

Executing aspect-specific instructions atomically (1) We have mentioned the idea to execute contiguous aspect-specific bytecode instructions atomically. The remaining question is how to identify the contiguous aspect-specific bytecode instructions.

One way to identify the aspect-specific bytecode instructions is to use the instructions’ line number information. Theoretically, the aspect-specific bytecode instructions, such as those for aspect management, have no corresponding Java source

code, either in the pure Java program or the AspectJ program. Similar to the AtomicMethodVMListener, we can implement a VMListener, such that right before the execution of a bytecode instruction, it checks the instruction's line number. If it is negative (i.e. it is not application code), execute the bytecode instruction atomically. An implementation of such reduction is shown in Figure 28.

Such implementation is straightforward. However, it relies on the "correctness" of the line number information and is closely tied to the implementation of the AspectJ compiler. For example, some aspect-specific bytecode instructions are mapped to the line of aspect declaration and thus have valid line numbers. For such case, the reduction based on line number information is not as much as the optimal approach.

```
public void executeInstruction(JVM vm) {
    Instruction ins1 = vm.getLastInstruction();
    int line = ins1.getLineNumber();
    ThreadInfo thread = vm.getCurrentThread();
    if (line < 0) {
        thread.isFirstStepInsn = true;
    }
}
```

Figure 28. An implementation of NegativeLineVMListener

Executing aspect-specific instructions atomically (2) Another approach to execute contiguous aspect-specific instructions atomically is to add atomicity boundary before and after the contiguous aspect-specific instructions. A sample implementation is shown in Figure 29.

In Figure 29, line 4 loads the atomicity boundary class, `gov.nasa.jpj.jvm.AtomicBoundary`, into the workspace of Soot. Line 5 creates a reference to the method `AtomicBoundary.beginAtomic()`, line 6 creates an invoke statement to this reference, and line 7 adds this statement as the first statement of a contiguous block of aspect-specific statements. Similarly, line 11 creates the reference to `AtomicBoundary.endAtomic()`, line 12 creates the invoke statement and line 13 adds it to the end of the block of aspect-specific statements. Meanwhile, one also needs to use the MJI scheme to begin the atomic execution and end the atomic execution at JPF's runtime.

```
1. SootClass atomicBoundary = null;
2. Stmt beginS = null, endS = null;
3. if (ifCustomized) {
4.   atomicBoundary =
      Scene.v().loadClassAndSupport("gov.nasa.jpj.jvm.AtomicBoundary");
5.   SootMethodRef beginR = Scene.v().makeMethodRef(atomicBoundary,
"beginAtomic", new ArrayList(), soot.VoidType.v(), true);
6.   beginS = Jimple.v().
      newInvokeStmt(Jimple.v().newStaticInvokeExpr(beginR));
7.   units.addFirst(beginS);
8. }
9. ...
10.  if (ifCustomized) {
11.    SootMethodRef endR = Scene.v().makeMethodRef(atomicBoundary,
"endAtomic", new ArrayList(), soot.VoidType.v(), true);
12.    endS = Jimple.v().newInvokeStmt(
Jimple.v().newStaticInvokeExpr(endR));
13.    units.addLast(endS);
14.  }
```

Figure 29. Adding atomicity boundary to aspect-specific code

However, decorating all blocks of aspect-specific statements with atomicity boundaries is a tedious task and tied to a specific version of the AspectJ compiler (which prevents one from using a newer version of AspectJ compiler). We are not using this approach in our experiments.

5.3.3. Aspect-Specific Methods in *ajc*

While we carry out the above analysis and experiments based on *abc*, many of the results can be equally applied to *ajc*. For example, the *ajc* compiler uses a naming convention similar to the one in the *abc* compiler. While the *ajc* compiler doesn't provide an option to generate corresponding Java source code like the “-dava” option in the *abc* compiler, one can learn about the names of the aspect-specific methods using *Byte Code Engineering Library* (BCEL 2006) or the source code of *ajc*. I also implement a simple tool that is based on BCEL, which can read field names, method names and line numbers of bytecode instructions from a Java class file (Chen 2008). As an example, using these methodologies, I have identified the following regular expressions to catch the names of the five aspect-specific methods for *perthis* aspect:

```
aspectOf
hasAspect
.*?per.*?Get.* true
.*?per.*?Set.* true
.*?per.*?Bind.* true
```

We can configure JPF with the above regular expressions so that these methods are marked as “atomic” methods. If we compile the example in Figure 26 with *ajc* and plug `AtomicMethodVMLListener` implemented in Figure 25 into JPF, the search space of

this setup is the same as the search space of its Java counterpart. Similarly, we can also reduce the search space for aspect variations *issingleton* and *pertarget* to their Java counterparts.

CHAPTER VI

VERIFYING ASPECT-ORIENTED DESIGN PATTERNS

In the previous sections, we have shown that an AspectJ program has a larger search space than its pure Java counterpart, if without additional search heuristics. In addition to worsening the state space explosion problem, such an increase of state space may well be an indication of added concurrency problems. In this section, we investigate such concurrency issues in the context of generic design techniques, using JPF extension we have created insofar. We first point out the lack of synchronization during aspect instantiation and show a consequent flaw existing in a common Aspect-Oriented programming paradigm. We then introduce a generic programming model to avoid such a trap, verified with JPF (together with the abstraction library we have created). We then examine more Aspect-Oriented reimplementation of design patterns in the literature, showing the bugs we have found and providing patches as needed.

6.1. Aspect-Oriented Design Patterns

A design pattern names a generic solution to a common problem at an abstract level and describes its consequence (Gamma 1995). It is one of the definite characters of a framework and promotes design reuse. Researchers have studied the Aspect-Oriented design patterns. Hannemann et. al. showed that in the 23 GoF design patterns, there are

17 cases that have modularity improvements with Aspect Oriented techniques (Hannemann 2002). The modularity benefits are concluded *as better code locality, reusability, composition transparency and (un)pluggability*. Since design patterns have summarized the generic solutions to common problems, they have a good chance to be reused and shall be checked strictly to guarantee the correctness.

Meanwhile, there are also other common programming patterns of AspectJ, which are similar to design patterns in the sense that they are abstract solutions to common problems but additionally require Aspect-Oriented features (we may broadly call them Aspect-Oriented design patterns). These design techniques are often taught in programming books and literature to demonstrate the superiority of Aspect-Oriented techniques. However, these programming schemes might not have been carefully inspected in multi-threaded environments: Concurrent programming is already difficult and error-prone in pure Java programs, but it is even harder to reason AspectJ programs without the knowledge of AspectJ weaving. In fact, we do discover flaws in some well-accepted programming paradigms, using the model checking facility we have built. Furthermore, it seems *unreasonable* to require an AspectJ programmer to have deep knowledge about aspect weaving to write a correct AspectJ program. For this reason, we propose a simple, generic programming model that is free from the extra concurrency problem due to aspect instantiation. We start this section with a well-accepted but actually flawed programming paradigm.

6.2. A Classic Example: Using AOP for Concurrency Control

AOP techniques can be used to enforce the *pre-* and *post-* condition of a function invocation in a modular way. One of such use is to assure the proper order to access a critical section, on top of a possibly non-thread-safe solution. A classic example is to modularize the “multiple read, single write” policy using AspectJ (Colyer 2004). The true intention (i.e. the equivalent pure Java implementation) is shown in Figure 30, and the corresponding AspectJ version (slightly modified from (Colyer 2004)) is shown in Figure 31. Notice that in Colyer’s original example, the implementation of `ReadWriteLock` is based on Doug Lea’s concurrency library (Lea 1999); to avoid burying the true cause of errors with implementation details, I use a simpler yet still popular alternative that is based on an integer counter. But the problem I reported here can be replicated with both `FIFOReadWriteLock` and `WriterPreferenceReadWriteLock`, the two main implementations of `ReadWriteLock` in Lea’s library. It is also worth stressing that this concurrency problem is independent to AspectJ compilers: It can be reproduced with both *abc* and *ajc*, the two mainstream AspectJ compilers.

In Figure 30, `main()` is the test harness that creates and starts various reader/writer threads. The only purpose of the reader thread is to execute the `read()` method of a shared resource (line 30), and the only action of the writer thread is to execute the `write()` method of the shared resource (line 40). For this test case, we have one instance of `Resource` (line 3), which is shared among the reader threads and the writer threads. The “multiple read, single write” policy is enforced by acquiring a read (or write) lock at the entrance of the `read()` method (or the `write()` method), and

releasing a read (or write) lock at the exit of the `read()` method (or the `write()` method), as shown in line 16-25. Each resource is attached with such a read/write lock.

Figure 31 shows a popular implementation of the read/write lock: It uses an integer counter to track the current status of the lock. To acquire a *read* lock, the counter must initially be non-negative (i.e. the lock is not currently in the *write* state), as guarded by line 6-10; to acquire a *write* lock, the counter must be zero (i.e. the lock is currently in the *idle* state, neither *read* nor *write*), as guarded by line 28-32. Each time when a read operation is done and a read lock is released, the counter decreases by one (line 19); when a write operation completes and a write lock is released, the counter is set to 0 (line 43). Notice that the fields and statements in the grey area are not required for the actual implementation of the read/write lock; instead, they articulate part of the correctness property to be checked by JPF: The tracking counter must be in a consistent state, i.e. it must be no less than -1 at any time, no less than 0 to allow “read” operation, equal to 0 to allow “write” operation. Of course, JPF can also check whether this program may be deadlocked.

If we run JPF to check the formal model in Figure 30 and 31 with two read threads and one write thread, the verification completes without finding any error (i.e. no runtime exception, no deadlock). Increasing the number of participating reader threads and writer threads doesn't lead to any flaw. Therefore, we conclude that the baseline example has no concurrency flaw.

```
1. ...
2. public static void main(String args()) {
3.     Resource resource = new Resource();
4.     for all reader thread
5.         reader_thread = new ReaderThread(resource);
6.     for all writer thread
7.         writer_thread = new ReaderThread(resource);
8.     start all readers/writer threads
9. }
10. ...
11. public class Resource {
12.     ReadWriteLock lock = new ReadWriteLock();
13.     private void readCriticalSection() { ... }
14.     private void writeCriticalSection() { ... }
15.
16.     public void read() {
17.         lock.lockRead();
18.         readCriticalSection();
19.         lock.unlockRead();
20.     }
21.     public void write() {
22.         lock.lockWrite();
23.         writeCriticalSection();
24.         lock.unlockWrite();
25.     }
26. }
27. class ReaderThread extends Thread {
28.     private volatile Resource r;
29.     public void run() {
30.         r.read();
31.     }
32.     public ReaderThread(Resource resource) {
33.         r = resource;
34.     }
35. }
36.
37. class WriterThread extends Thread {
38.     private volatile Resource r;
39.     public void run() {
40.         r.write();
41.     }
42.     public WriterThread(Resource resource) {
43.         r = resource;
44.     }
45. }
```

Figure 30. An example to assure the “multiple read, single write” policy

```

1.class ReadWriteLock {
2. private volatile int readers = 0;
3. public void lockRead() {
4.     synchronized (this) {
5.         // System.out.println(this + " before lockRead: " + readers);
6.         while (readers==0) {
7.             try { wait(); } catch (InterruptedException ex) {
8.                 System.out.println(this + " Interrupted in lockRead!");
9.             }
10.        }
11.        if (!(readers>=0))
12.            {throw new RuntimeException(this+ " readers=" + readers);}
13.        readers++;
14.        // System.out.println(this + " after lockRead: " + readers);
15.    } }
16. public void unlockRead() {
17.     synchronized (this) {
18.         // System.out.println(this+" before unlockRead:"+ readers);
19.         readers--;
20.         if (readers==0) { notifyAll(); }
21.         //if (!(readers>=0))
22.         // { throw new RuntimeException("readers=" + readers); }
23.         // System.out.println(this + " after unlockRead: "+readers);
24.     } }
25. public void lockWrite() {
26.     synchronized (this) {
27.         // System.out.println(this + " before lockWrite: "+readers);
28.         while (readers!=0) {
29.             try { wait(); } catch (InterruptedException ex) {
30.                 System.out.println("Interrupted in lockWrite!");
31.             }
32.        }
33.        if (!(readers==0))
34.            { throw new RuntimeException("readers=" + readers); }
35.        readers = -1;
36.        // System.out.println(this + " after lockWrite: "+ readers);
37.    } }
38. public void unlockWrite() {
39.     synchronized (this) {
40.         // System.out.println(this+" before unlockWrite: "+readers);
41.         //if (!(readers==0))
42.         // { throw new RuntimeException("readers=" + readers); }
43.        readers = 0;
44.        notifyAll();
45.        // System.out.println(this+" after unlockWrite: "+readers);
46.    } }
47.}

```

Figure 31. An implementation of lock for shared read and exclusive write

Figure 32 and Figure 33 show an Aspect-Oriented implementation corresponding to the example in Figure 30, which leverages the same `ReadWriteLock` class in Figure 31. Figure 32 defines a generic, abstract aspect to enforce the “multiple read, single write” rule, which can be instantiated to advise the access control of a particular class. An instantiation of this abstract aspect for a particular `Resource` class is shown in Figure 33. Notice that the `Resource` class allows `read()` and `write()` operations but has no concurrency control; therefore, to create a thread-safe resource, one has to additionally enforce access control rules using external lock mechanisms, which often involves scattered code and breaks the natural class hierarchy. With AspectJ, such enforcement is modularly defined in the abstract aspect `MultipleReadersSingleWrite` and can be conveniently reused by aspect inheritance (i.e. code reuse as well as design reuse). This is a classic example to show the advantages of AOP techniques: they can be used to enforce concurrency control in a modular way (Colyer 2004).

If we run JPF with the extension we created in the previous section to check the AspectJ version (with two read threads and one write thread), surprisingly, JPF reports error traces in this popular programming example, including an error trace that leads to a deadlock condition and an error trace that leads the counter into an inconsistent state (e.g. the counter is -2). The error is more intuitive if we uncomment the “`System.out.print(...)`” statements in Figure 31. We show the printout along the deadlock error trace in Figure 34 and a simplified error trace in Figure 35; an interested reader can refer to (Chen 2008) to replay the full error traces. We briefly explain the cause of the error in the below.

```

1. public abstract aspect MultipleReadersSingleWriter
2.   perthis ( readMethod() || writeMethod() ) {
3.     private volatile ReadWriteLock rw = new ReadWriteLock();
4.     protected abstract pointcut readMethod();
5.     protected abstract pointcut writeMethod();
6.     before(): readMethod() { rw.lockRead(); }
7.     after(): readMethod() { rw.unlockRead(); }
8.     before(): writeMethod() { rw.lockWrite(); }
9.     after(): writeMethod() { rw.unlockWrite(); }
10. }

```

Figure 32. A generic aspect to enforce the “multiple read, single write” policy

```

1. public class Resource {
2.   public void read() {
3.     readCriticalSection();
4.   }
5.   public void write() {
6.     writeCriticalSection();
7.   }
8.   public void readCriticalSection() {...}
9.   public void writeCriticalSection() {...} }
10. public aspect ReadersWriterAspect extends
MultipleReadersSingleWriter
11.   perthis( readMethod() || writeMethod() ){
12.   public ReadersWriterAspect() {
13.   }
14.   protected pointcut readMethod():
15.     execution(public * Resource+.read());
16.   protected pointcut writeMethod():
17.     execution(public * Resource+.write());
18. }

```

Figure 33. Instantiating MultipleReadersSingleWriter aspect

Figure 34 shows the printout along the trace to deadlock, which indicates two instances of `ReadWriteLock` have been created for the same resource: One has the hashcode `@aaf4` and the other has `@aa8b`. This is not only semantically incorrect (i.e. we have only one resource, and there should be only one instance of `ReadWriteLock` for this resource), but also leads to a deadlock condition. As shown in Figure 35, the first reader thread enters the read state and sets the counter to 0; thereafter, a new aspect and the associated resource and lock are created and override the old ones, which causes the counter to be “reset” to 0; subsequently, the first thread proceeds to unlock the read lock, decreases the counter by 1 (now the counter is -1) and runs to its end. The second reader thread and the writer thread proceed but both get stuck when they try to acquire the lock, since the counter is -1 and no thread can bring it out of this status. The system thus enters a deadlock condition.

```
reader threads: 2; writer threads: 1
ReadWriteLock@aaf4 before lockRead: 0
ReadWriteLock@aaf4 after lockRead: 1
ReadWriteLock@aa8b before unlockRead: 0
ReadWriteLock@aa8b after unlockRead: -1
ReadWriteLock@aa8b before lockRead: -1
ReadWriteLock@aa8b before lockWrite: -1
```

Figure 34. Printout of the `ReadWriteLock` before/after a lock/unlock operation

```

1.Td0: create the shared resource
2.Td1: pointcut readMethod()
3.Td1: if (no aspect is instantiated for resource)
4.Td1:   new ReadersWriterAspect() for resource, @aspect1
5.Td1:   new ReadWriteLock();
6.Td2: pointcut readMethod()
7.Td2: if (no aspect is instantiated for resource)
8.Td2:   new ReadersWriterAspect() for resource, @aspect2
9.Td2:   new ReadWriteLock();
10.Td1:  rw = new ReadWriteLock();
11.Td1:  readers = 0;
12.Td1: rw.lockRead(); increase readers to 1
13.Td2: @aspect2 override @aspect1, now readers is 0
14.Td1: readCriticalSection();
15.Td1: rw.unlockRead(); decrease readers from 0 to -1
16.Td1: reach to its end
17.Td2: rw.lockRead(); stuck here since readers is -1
18.Td3: rw.lockWrite(); stuck here since readers is -1

```

Figure 35. The simplified error trace for the AspectJ program in Figure 33

To explain such phenomenon, we first review the methodology of aspect instantiation adopted in the AspectJ compiler. Figure 36 shows the aspect instantiation routines that the *abc* compiler typically generates (the code is output by *abc* with `-dava` option). The static method `abc$perThisBind()` is invoked before any *perthis*-pointcut match, to assure that an instance of the current aspect has been properly initialized. The *perthis*- semantics requires that only one instance of the particular aspect is created for the subject of the particular pointcut: for this purpose, when the aspect instance is created, it is associated with the subject of *perthis*-pointcut by `xxxSet()` method and retrievable by `xxxGet()` method. However, since there is no synchronization control on the aspect instantiation within `abc$perThisBind()`, in

multi-threaded environment, it is likely that two threads calling `abc$perThisBind()` will pass the *null test* (line 5) before the `xxxSet()` operation completes (line 6). Subsequently, there is a possibility that two aspect instances will be created for the same subject in the matched pointcut and eventually one will override the other. With this in mind, now we are able to understand the error trace in Figure 35. After the main thread creates the shared resource (line 1), thread 1 (a read thread) executes the *before* advice for the matched pointcut `readMethod()` (line 2), and invoke `abc$perThisBind()` to assure that the aspect has been properly instantiated (line 3-5). Meanwhile, thread 2 also executes the *before* advice for the matched pointcut and invoke `abc$perThisBind()` to initiate the aspect instance. As discussed in the above, both threads pass the *null test* and create their own aspect instances. From line 10 to line 12, thread 1 completes aspect instantiation and acquires the read lock, correctly prints out the value of the counter (0, 1, respectively, in Figure 34). In line 13, thread 2 (the second read thread) finishes aspect instantiation and overrides the aspect value previously written by thread 1. Now the counter is reset to 0. In line 14-16, thread 1 continues, enters the critical section, decreases the counter (which is just initialized to 0 by thread 2) by 1 to unlock the read lock and proceeds to its end. Now the counter is -1. Thereafter, thread 2 proceeds to acquire the read lock and gets stuck in line 7 of Figure 31 since the counter is always -1. Similarly, thread 3 proceeds to acquire the write lock and gets stuck in line 29 of Figure 31 since the counter is always -1. Now the system enters a deadlock condition.

```

1. public static void abc$perThisBind(Object theObject$40) {
2.   ReadersWriterAspect$abc$PerThis castedArg$41;
3.   if (theObject$40 instanceof ReadersWriterAspect$abc$PerThis) {
4.     castedArg$41 = (ReadersWriterAspect$abc$PerThis) theObject$40;
5.     if (castedArg$41.myabc_pattern_rw2_aspectj_
ReadersWriterAspect$abc$PerThisGet() == null) {
6.       castedArg$41.myabc_pattern_rw2_aspectj_
ReadersWriterAspect$abc$PerThisSet(new ReadersWriterAspect());
7.     }
8.   }
9. }

```

Figure 36. Aspect instantiation (*perthis*)

Similarly, the counter may enter an inconsistent state because of the overriding of aspect instances: The first reader thread may set the counter to 1; then the writer thread may create a new aspect and “reset” the counter to -1; thereafter the first read thread may further decrease the counter to -2 by unlocking the read lock.

It is worth emphasizing that the reported error is caused by the lack of synchronization during aspect instantiation, and is orthogonal to the implementation of `ReadWriteLock`. For example, one may try an alternative implementation by relaxing the condition “while (readers== -1) {” in line 6 of Figure 31 to “while (readers < 0) {”; JPF still catches a deadlock condition for such alternative implementation. Furthermore, as mentioned previously, JPF also finds deadlock conditions if we replace the counter-based `ReadWriteLock` with `FIFOReadWriteLock` and `WriterPreferenceReadWriteLock` from Lea’s concurrency library. Moreover, this error doesn’t seem to be caused by an accidental mistake in the implementation of AspectJ compilers: JPF discovers the deadlock

condition for the sample in Figure 32 and 33, no matter it is compiled by *abc* or *ajc* (as a reminder, *abc* and *ajc* are the two mainstream AspectJ compilers). The lack of synchronization during aspect instantiation may be a deliberate design choice to avoid the synchronization overhead; it is thus the responsibility of AspectJ developers to assure the correctness on such unsafe infrastructure.

This catch of errors is somewhat surprising, as similar programming schemes are wide spread and are even taught as examples in programming books (Colyer 2004). Nevertheless, it is evidence of how difficult it is to eliminate concurrent errors in Aspect-Oriented, multi-threaded environment, and the effectiveness of our extension to JPF to battle this problem. We will see some more examples in the following.

6.3. A Generic Model to Avoid the Aspect Instantiation Problem

The source of the above aspect instantiation problem is the lack of synchronization during the creation of an aspect for a particular subject. Naturally, it can be eliminated by *issingleton* aspect (i.e. the default aspect instantiation method in AspectJ) instead of *perthis* aspect. Notice that the semantic of *perthis* is to create an aspect instance for each subject of a matched pointcut as a syntax sugar, i.e. to establish the association between the subject of the matched pointcut and the aspect instance (subsequently, it also associates the subject and the attributes of this aspect instance). Instead of *perthis*, we can establish such association by using another aspect feature, *Inter-type definition (ITD)*. In particular, we can remove attributes of an aspect and declare them as the attributes of the subject, using *ITD*. It is worth noticing that the *issingleton* aspect is exempted from the concurrency problem. As shown in Figure 37, an *issingleton* pattern is initialized with an

eager Singleton Pattern, i.e. the singleton instance is initialized statically, even before an instance of aspect is created. Since such instantiation is done statically and the static class instantiation is implicitly treated as *synchronized* by the Java Virtual Machine, there is no chance for the two threads to create two instances of the same aspect. Therefore, the aspect instantiation problem is naturally eliminated with *issingleton* aspect.

```
static {
    ReadersWriterAspect DavaTemp_abc$perSingletonInstance;
    DavaTemp_abc$perSingletonInstance = null;
    try {
        DavaTemp_abc$perSingletonInstance=new
        ReadersWriterAspect();
    } catch (Throwable catchLocal$4) {
        abc$initFailureCause = catchLocal$4;
    }
    abc$perSingletonInstance = DavaTemp_abc$perSingletonInstance;
}
```

Figure 37. Aspect instantiation (*issingleton*)

As an example, we show a reimplementation with *issingleton* aspect that is free from the above violation. As shown in Figure 38, the reimplementation closely resembles the original *perthis* implementation in Figure 32 and 33. The main changes are: In line 1, the aspect `MultipleReadersSingleWriter` is declared as *issingleton* (by default). In line 4, instead of declaring `ReadWriteLock` as a field of the aspect, we declare it as a field of `Resource`, the subject of the matched pointcut, via inter-type definition. Thereafter, to invoke `ReadWriteLock.lockRead()` in an advice, we

will have to pass a reference of `ReadWriteLock`, which is indirectly done by passing a reference of `Resource` (i.e. remember that `Resource` has a reference to its `ReadWriteLock` via `ITD`). Such arrangement requires changes to the signature of pointcuts and their advices, but in a minimal degree.

```

1. public abstract aspect MultipleReadersSingleWriter {
2.   protected abstract pointcut readMethod(Resource resource);
3.   protected abstract pointcut writeMethod(Resource resource);
4.   public volatile ReadWriteLock Resource.rw = new
   ReadWriteLock();
5.   before(Resource resource): readMethod(resource) {
6.     resource.rw.lockRead();
7.   }
8.   after(Resource resource): readMethod(resource) {
9.     resource.rw.unlockRead();
10.  }
11.  before(Resource resource): writeMethod(resource) {
12.    resource.rw.lockWrite();
13.  }
14.  after(Resource resource): writeMethod(resource) {
15.    resource.rw.unlockWrite();
16.  }
17.}
18. public aspect ReadersWriterAspect extends
   MultipleReadersSingleWriter {
19.   protected pointcut readMethod(Resource resource):
20.     execution(public * Resource+.read()) && this(resource);
21.   protected pointcut writeMethod(Resource resource):
22.     execution(public * Resource+.write()) && this(resource);
23.}

```

Figure 38. A reimplementaion of Figure 32 (without concurrency violations)

After these changes, we find no violation using JPF. Varying the number of reader threads and writer threads doesn't lead to the discovery of errors, either. Moreover, as the newer version (Figure 38) doesn't impose any restriction on the AspectJ program (Figure

32 and Figure 33), it is a generic solution to avoid the *perthis*-aspect instantiation problem demonstrated in this section.

It is worth noticing that in our experiments, this assertion holds for *pertarget* aspect instantiation as well (actually, *perthis* aspect instantiation shares most of its code with *pertarget* aspect instantiation in the *abc* compiler). Furthermore, although the above reasoning is based on the *abc* compiler, in our experiments, the above conclusion applies to the *ajc* compiler as well.

6.4. Model Checking Aspect-Oriented Design Patterns

In addition to the aspect instantiation problem, the intricacy of concurrency issues due to the application logic still applies. After paving the way for the aspect instantiation problem, we look at concurrency issues that are potentially raised by the application logic as well as the aspect instantiation problem. Hannemann et. al. re-implement the 23 GoF design patterns in AspectJ, to study the advantages of AOP techniques. In this section, we apply model checking to check their implementation (their source code can be found in (Hannemann 2002)). Notice that we only make minimal changes to their sample programs: the modifications are limited to fields and statements for the sole purpose of correctness property specification and driving the system under investigation.

6.4.1. Observer Pattern

Figure 39 shows a sample application that adopts the Aspect-Oriented Observer Pattern (slightly modified to add the correctness property specification). Compared with its Object-Oriented counterpart, it improves on *locality*, *reusability*, *composition*

transparency and *pluggability* (Hannemann 2002). For the purpose of explanation, we briefly go over the sample program. The aspect `ObserverProtocol` defines a generic observer protocol that can be specialized for any type of subjects and observers. To achieve this, `ObserverProtocol` maintains the mapping between a subject and the collection of its listeners via a hashtable `perSubjectObservers` (line 4). The function `getObservers()` returns the collection of observers for a particular subject; if no observer exists for the subject, an empty collection of observers is created instead of returning a *null* pointer (line 9-15). To add an observer (i.e. `addObserver()`), it first fetches the listener collection for the subject, and adds the observer to the particular collection (line 18). The function `countObservers()` returns the number of observers for a particular subject. For a particular publish-subscribe relationship, one can instantiate the abstract `ObserverProtocol` by filling the pointcut that matches subjects' changes (line 35). The test skeleton (line 45-50) creates two threads, whose sole purpose is to add observers to a particular subject, i.e. it adds color observers `s1` and `s2` to the subject `p`, and position observer `s3` and `s4` to the subject `p`. After the addition of the two observers in two individual threads, it asserts that there must be two observers for the subject `p`.

We run JPF to verify the above program. JPF reports a violation trace in some seconds, as shown in Figure 40 (in a simplified presentation). The error is due to the lack of synchronization in the access of the observer storage object, `perSubjectObservers`; more exactly, it is due to the lack of access control of the observer storage object for a particular subject. For the error trace in Figure 40, when

Thread 1 detects that no observer list exists for a particular subject, an observer list is created for the particular subject (line 12-13); meanwhile, Thread 2 detects that no observer lists exists for a particular subject and then creates an observer list for the particular subject, before the observer list created by Thread 1 is successfully stored in `perSubjectObservers`. Therefore, for the particular subject, when Thread 2 adds an observer, it actually overrides the existing observer list with a new observer list (which contains a new observer), instead of adding the new observer to the observer list.

There are several approaches to eliminate such concurrency error. For example, if we require that all auxiliary functions that access observer storage `perSubjectObservers` (e.g. `getObservers()`, `addObservers()` and `removeObservers()`) must be synchronized methods, then JPF doesn't find any such error.

```

1. public abstract aspect ObserverProtocol {
2.   protected interface Subject { }
3.   protected interface Observer { }
4.   private Hashtable perSubjectObservers;
5.   public int countObservers(Subject subject) {
6.     return getObservers(subject).size();
7.   }
8.   public List getObservers(Subject subject) {
9.     if (perSubjectObservers == null) {
10.      perSubjectObservers = new Hashtable();    }
11.     List observers = (List)perSubjectObservers.get(subject);
12.     if ( observers == null ) {
13.      observers = new LinkedList();
14.      perSubjectObservers.put(subject, observers); }
15.     return observers;
16.   }
17.   public void addObserver(Subject subject, Observer observer) {
18.     getObservers(subject).add(observer);
19.   }
20.   public void removeObserver(Subject subject, Observer observer) {
21.     getObservers(subject).remove(observer); }
22.   protected abstract pointcut subjectChange(Subject s);
23.   after(Subject subject): subjectChange(subject) {
24.     Iterator iter = getObservers(subject).iterator();
25.     while ( iter.hasNext() ) {
26.       updateObserver(subject, ((Observer)iter.next()));
27.     }
28.   }
29.   protected abstract void updateObserver(Subject sub, Observer obs);
30. }
31. public aspect ColorObserver extends ObserverProtocol{
32.   declare parents: Point implements Subject;
33.   declare parents: Screen implements Observer;
34.   protected pointcut subjectChange(Subject subject):
35.     call(void Point.setColor(Color)) && target(subject);
36.   protected void updateObserver(Subject subject, Observer observer){
37.     ((Screen)observer).display("screen updated:...");
38.   } }
39. public aspect ScreenObserver extends ObserverProtocol{
40.   declare parents: Screen implements Subject;
41.   declare parents: Screen implements Observer;
42.   ...
43. }
44. void main(String args()) {
45.   Thread thread1 = new ObserverThread(p, s1, s3);
46.   Thread thread2 = new ObserverThread(p, s2, s4);
47.   thread1.start(); thread2.start();
48.   thread1.join(); thread2.join();
49.   assert(ColorObserver.aspectOf().countObservers(p)==2);
50.   ... }

```

Figure 39. A sample application using Aspect-Oriented Observer Pattern

```

Th0: start thread 1 and thread 2
Th1: 11.    List observers = (List)perSubjectObservers.get(subject);
Th1: 12.    if ( observers == null ) {
Th1: 13.        observers = new LinkedList();
Th2: 11.    List observers = (List)perSubjectObservers.get(subject);
Th2: 12.    if ( observers == null ) {
Th2: 13.        observers = new LinkedList();
Th1: 14.        perSubjectObservers.put(subject, observers); }
Th1: 18.    getObservers(subject).add(observer);
Th2: 14.        perSubjectObservers.put(subject, observers); }
Th2: 18.    getObservers(subject).add(observer);
Th0: 52.    assert (ColorObserver.aspectOf().countObservers(p)==2);
Error!

```

Figure 40. Error trace in the original Aspect-Oriented Observer Pattern

6.4.2. Flyweight Pattern

Figure 41 shows a sample application that adopts the Aspect-Oriented Flyweight Pattern (slightly modified to add the correctness property specification). Compared with its Object-Oriented counterpart, it improves on *locality*, *reusability*, *composition transparency and pluggability* (Hannemann 2002). For the purpose of explanation, we briefly go over the sample program. The aspect FlyweightProtocol defines a generic flyweight protocol that enables not only design reuse but also code reuse of the Flyweight design pattern. The flyweight object can be created via the createFlyweight(Object) method (line 4) and retrieved via the getFlyweight(Object) method (line 5-13), and the association between an intrinsic key and a flyweight object is maintained through the hashtable flyweights (line 2). Since the exact type of a flyweight object is not known in the generic aspect

FlyweightProtocol, the flyweight creation method `createFlyweight()` is defined as abstract and has to be instantiated in a concrete sub-aspect of FlyweightProtocol (e.g. the concrete aspect FlyweightImplementation implements the `createFlyweight()` method, since only the specific application knows the type of a flyweight). The fields and statements in the grey area (line 18, line 21-27) are added to the original Flyweight implementation; their sole purpose is to establish the criteria that only one flyweight object is created for an intrinsic key. The test harness is defined in line 40-51: the only action of FlyweightThread is to get a flyweight object for a particular intrinsic key (for our example, it is the character 'c'). Two threads will get the flyweight object concurrently; ideally, there should be only one flyweight object created.

We run JPF to verify the Flyweight sample application. JPF reports a violation trace in seconds, as shown in Figure 42 (in an abstract presentation). The error is due to the lack of synchronization in the access of the flyweight storage object, `flyweights`. For the error trace in Figure 42, when Thread 1 detects that the flyweight storage object `flyweights` doesn't have the flyweight for a particular intrinsic key, it continues to create the flyweight for the key (Thread 1, line 9); meanwhile, Thread 2 also detects that the flyweight storage object `flyweights` doesn't have the flyweight for the intrinsic key yet and continues to create the flyweight for the key (Thread 2, line 9). Subsequently, there are two flyweight objects created for the same intrinsic key (the latter will override the first one), which may lead to data race in various situations.

```

1. public abstract aspect FlyweightProtocol {
2.     private Hashtable flyweights = new Hashtable();
3.     protected interface Flyweight{};
4.     protected abstract Flyweight createFlyweight(Object key);
5.     public Flyweight getFlyweight(Object key) {
6.         if (flyweights.containsKey(key)) {
7.             return (Flyweight) flyweights.get(key);
8.         } else {
9.             Flyweight flyweight = createFlyweight(key);
10.            flyweights.put(key, flyweight);
11.            return flyweight;
12.        }
13.    }
14.}
15. public aspect FlyweightImplementation extends FlyweightProtocol {
16.     declare parents: CharacterFlyweight implements Flyweight;
17.     declare parents: WhitespaceFlyweight implements Flyweight;
18.     volatile int counter = 0;
19.     protected Flyweight createFlyweight(Object key) {
20.         char c = ((Character) key).charValue();
21.         if (c=='c') {
22.             counter++;
23.         }
24.         if (counter>=2) {
25.             System.out.println("counter=" + counter);
26.             throw new RuntimeException("counter=" + counter);
27.         }
28.         Flyweight flyweight = null;
29.         if (Character.isWhitespace(c)) {
30.             flyweight = new WhitespaceFlyweight(c);
31.         } else {
32.             flyweight = new CharacterFlyweight(c);
33.         }
34.         return flyweight;
35.     }
36.     public PrintableFlyweight getPrintableFlyweight(char c) {
37.         return (PrintableFlyweight) getFlyweight(new Character(c));
38.     }
39.}
40. class FlyweightThread extends Thread {
41.     public void run() {
42.         FlyweightImplementation.aspectOf().getPrintableFlyweight('c');
43.     } }
44.     void main(String args[]) {
45.         FlyweightThread t1 = new FlyweightThread();
46.         FlyweightThread t2 = new FlyweightThread();
47.         t1.start(); t2.start();
48.         ... }

```

Figure 41. A sample application that uses Aspect-Oriented Flyweight Pattern

```

Th0: start thread 1 and thread 2
Th1: 42. FlyweightImplementation.aspectOf().getPrintableFlyweight
Th1: 37. getFlyweight(new Character(c));
Th1: 6.  if (flyweights.containsKey(key)) {
Th1: 9.     Flyweight flyweight = createFlyweight(key);
Th2: 42. FlyweightImplementation.aspectOf().getPrintableFlyweight
Th2: 37. getFlyweight(new Character(c));
Th2: 6.  if (flyweights.containsKey(key)) {
Th2: 9.     Flyweight flyweight = createFlyweight(key);
Th2: 24. if (counter>=2) {
Th2: 26. throw new RuntimeException("counter=" + counter);
Error!

```

Figure 42. Error trace in the original Aspect-Oriented Flyweight Pattern

There are several approaches to eliminate such a concurrency error. For example, if we require that all auxiliary functions that access the flyweight storage `flyweights` (e.g. `createFlyweight()`, `getFlyweight()`) are synchronized, JPF doesn't find any error for this variation.

6.4.3. Singleton Pattern

Figure 43 shows a sample application that adopts the Aspect-Oriented Singleton Pattern (slightly modified to add the correctness property specification). Compared with its Object-Oriented counterpart, it improves on *locality*, *reusability*, and *pluggability* (Hannemann 2002). For the purpose of explanation, we briefly go over the sample program. The aspect `SingletonProtocol` defines a generic singleton protocol that can be specialized for any type of singletons. To achieve this, `SingletonProtocol` maintains the mapping between the singleton class (i.e. represented by the `Class` object

of the singleton class) and the sole instance of the class. The pointcut `call` (line 5) intercepts the creation of objects that are a subclass of the `Singleton` interface, unless it is explicitly excluded otherwise. To enforce the singleton policy for a particular class, the concrete aspect must declare `Singleton` as the parent of the particular class (line 14), and implement the exclusion aspect as needed. Line 18-45 creates a test harness for the Singleton pattern. The sole action of `SingletonThread` is to create a `Printer` object and save the newly created object in the array. Each time when the constructor of the `Printer` class is called, the static counter `objectsSoFar` is increased by one and the created object has the current counter as its unique, immutable id (line 35). In the sample application, since there shall be only one `Printer` object created, we assert that the `Printer` id must be 1 (line 32).

We run JPF to verify the above program. JPF reports a violation trace in no time, as shown in Figure 44 (in a simplified presentation). The error is due to the lack of synchronization in the access of the singleton storage object, `singletons`. For the error trace in Figure 44, when Thread 1 detects that no `Printer` object has been created (line 7), it proceeds to create the `Printer` object and attempts to save the newly created object in the slot identified by the `Printer` class; meanwhile, Thread 2 also detects that no `Printer` object has been created and decides to create another `Printer` object. Such interleaving leads to the state that two `Printer` objects are created, violating the restriction of Singleton design pattern.


```

1. public abstract aspect SingletonProtocol {
2.   private volatile Hashtable singletons = new Hashtable();
3.   public interface Singleton {}
4.   protected pointcut protectionExclusions();
5.   Object around(): call((Singleton+).new(..))
   && !protectionExclusions() {
6.     Class singleton =
thisJoinPoint.getSignature().getDeclaringType();
7.     if (singletons.get(singleton) == null) {
8.       singletons.put(singleton, proceed());
9.     }
10.    return singletons.get(singleton);
11.  }
12.}
13. public aspect SingletonInstance extends SingletonProtocol {
14.   declare parents: Printer implements Singleton;
15.   protected pointcut protectionExclusions():
16.     call((PrinterSubclass+).new(..));
17.}
18. class SingletonThread extends Thread {
19.   Printer() ps;
20.   int slot;
21.   public SingletonThread(Printer() printers, int index) {
22.     ps = printers;
23.     slot = index;
24.   }
25.   public void run() {
26.     ps(slot) = new Printer();
27.   }
28.}
29. class Printer {
30.   public Printer() {
31.     id = ++ objectsSoFar;
32.     assert(id==1);
33.   }
34.   ...
35.}
36. ...
37. main(String args()) {
38.   Printer() printers = new Printer(MAX);
39.   SingletonThread() threads = new SingletonThread(MAX);
40.   for (int i = 0; i < MAX; i++) {
41.     threads(i) = new SingletonThread(printers, i);
42.     threads(i).start();
43.   }
44.   ...
45. }

```

Figure 43. A sample application that uses Aspect-Oriented Singleton Pattern

```
Th0: start thread 1 and thread 2
Th1: 26.    ps(slot) = new Printer();
Th1: 7.     if (singletons.get singleton) == null) {
Th2: 26.    ps(slot) = new Printer();
Th2: 7.     if (singletons.get singleton) == null) {
Th1: 8.     singletons.put singleton, proceed());
Th1: 31.    id = ++ objectsSoFar;
Th2: 8.     singletons.put singleton, proceed());
Th2: 31.    id = ++ objectsSoFar;
Th2: 32.    assert(id==1);
Error!
```

Figure 44. Error trace in the original Aspect-Oriented Singleton Pattern

There are several approaches to eliminate such a concurrency error. For example, if we require that all auxiliary functions that access the singleton storage, `SingletonProtocol.singletons`, are synchronized (e.g. enclose line 7-9 in Figure 43 with `synchronized(singletons) { ... }`), JPF doesn't find any error for this variation.

CHAPTER VII

THE MODELING FRAMEWORK REVISITED

7.1. Stability of Models for OSGi Applications

In Section 3.1, we have discussed the Façade design pattern used in the OSGi framework. By this way, the OSGi applications are shielded from the details of a particular OSGi implementation and have better portability across various implementations from different vendors. At the modeling level, the OSGi framework also varies, not because of different vendors but because we need to select different combinations of features to minimize the search space. As shown in Figure 45, although we may change the complexity of the modeling framework by choosing a specific combination of aspects, it may not be needed to vary the OSGi applications: In a typical OSGi application that implements `BundleActivator.start(BundleContext)` and `BundleActivator.stop(BundleContext)`, the runtime environment about the OSGi platform is only known through the parameter `BundleContext`, which exposes a subset of framework functionalities and internal representations. As long as we keep unchanged the interfaces of those classes that are directly accessible by an OSGi application, the OSGi application remains the same for a modeling framework with varying details.

To ease model development, we require the same signatures for classes accessible from a `BundleActivator`. In this way, often an OSGi application can be directly used as a formal model, and it requires minimal efforts to adapt an OSGi application into its modeling counterpart.

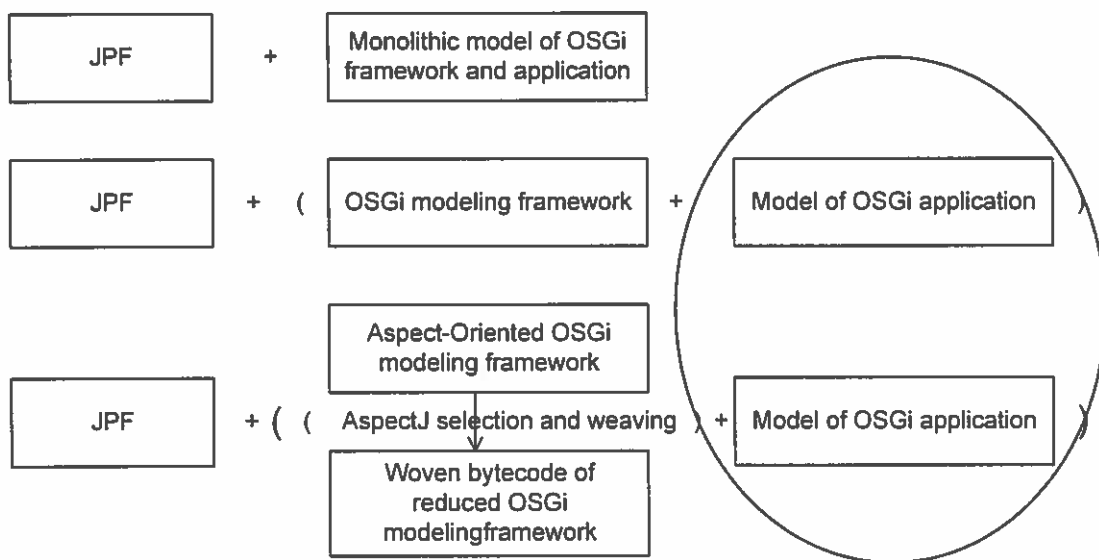


Figure 45. Models of OSGi applications with the Façade Pattern

7.2. Aspectizing the OSGi Framework at the Model Level

In Section 4.2, we use a motivating example to show the improvements of modularity by tailoring the modeling framework with AOP techniques. In Section 5, we develop the techniques to check AspectJ programs. In Section 7.1, we know that by a structural paradigm similar to a Façade pattern, we are able to shield an OSGi application

from varying details of a modeling framework. In this section, we describe a more complete aspectized OSGi framework. It is worthy to notice that while it is not new to aspectize a Java program, and there are many similarities for aspects at the modeling level and the application level, our aspectizing efforts are motivated for very different reasons and may be divided into very different categories.

We separate the following optional features from the core OSGi framework: `BundleListener`, `FrameworkListener`, `ServiceListener`, `Permission`, and `BundleContextValidator`. Each feature is specified by an aspect. In Section 4 we have discussed (fragments of) the aspects of `BundleListener` and `Permission`; now we will discuss the rest three aspects in the below.

7.2.1. `FrameworkListener`

Similar to `BundleListener` described in Section 4.2, the Knopflerfish framework also uses an Observer Pattern to publish framework events among subscribers. This can be modularized in an aspect, as `FrameworkListenerAJ` shown in Figure 46 and 47. In line 2, it uses ITD to add a field `frameworkListeners` to `Listeners` as the placeholder for framework event subscribers, which is exactly the same as the original Java implementation. Functions like `addFrameworkListener()`, `removeFrameworkListener()`, `frameworkError()`, `frameworkInfo()` and `frameworkEvent()` are helpers that are inserted via ITD to actually manipulate the subscriber list and broadcast various framework events among subscribers. These functions reside in `Listeners.java` in the original modeling framework and may be

invoked in different contexts. For example, an OSGi application may invoke `BundleContext.addFrameworkListener()`, which further invokes `Listeners.addFrameworkListener()` to add a framework event subscriber. As pointed out in Section 7.1, to minimize the changes required in adapting an OSGi application into a formal model, we shall keep the function interfaces (i.e. `BundleContext.addFrameworkListener()` in this case) that are directly accessible by an OSGi application unchanged. In this way, we are able to define pointcuts by these “fixed” signatures. Subsequently, in the generic, least framework, `BundleContextImpl` still has the function `addFrameworkListener()`, but it doesn’t actually invoke `Listeners.addFrameworkListener()` unless we include this aspect in the weaving. Similarly, we also create pointcuts and advices for `removeFrameworkListener()` in `BundleContextImpl.java` so that their implementations are only present if we include `FrameworkListenerAJ` in AspectJ weaving. Moreover, line 56-61 modularizes the management of framework events that are invoked from `ServiceRegistrationImpl.java`: if an exception is thrown when unregistering a service, an error message is broadcast to all framework event listeners. Finally, line 62-74 modularizes the management of framework events that is invoked from `ServiceReferenceImpl.java`: if an exception is thrown when getting a service (line 62-68) or ungetting a service (line 69-74), a framework error event needs to be broadcast to all event subscribers.

```

1. public privileged aspect FrameworkListenerAJ {
2.   private HashSet Listeners.frameworkListeners = new HashSet();
3.   void Listeners.addFrameworkListener(Bundle bundle,
FrameworkListener listener) {
4.     ListenerEntry le = new ListenerEntry(bundle, listener);
5.     synchronized (frameworkListeners) {
6.       frameworkListeners.add(le);
7.     }
8.   }
9.   void Listeners.removeFrameworkListener(Bundle bundle,
FrameworkListener listener) {
10.    synchronized (frameworkListeners) {
11.      frameworkListeners.remove(new ListenerEntry(bundle,
listener));
12.    }
13.  }
14.  void Listeners.frameworkError(Bundle b, Throwable t) {
15.    frameworkEvent(new FrameworkEvent(FrameworkEvent.ERROR, b,
t));
16.  }
17.  void Listeners.frameworkInfo(Bundle b, Throwable t) {
18.    frameworkEvent(new FrameworkEvent(FrameworkEvent.INFO, b,
t));
19.  }
20.  public void Listeners.frameworkEvent(final FrameworkEvent evt)
{
21.    ListenerEntry[] fl;
22.    synchronized (frameworkListeners) {
23.      fl = new ListenerEntry(frameworkListeners.size());
24.      frameworkListeners.toArray(fl);
25.    }
26.    for (int i = 0; i < fl.length; i++) {
27.      final ListenerEntry l = fl[i];
28.      try {
29.        ((FrameworkListener)l).frameworkEvent(evt);
30.      } catch (Throwable pe) { // Don't report Error events
again, since probably would go into an infinite loop.
31.        if (evt.getType() != FrameworkEvent.ERROR) {
32.          frameworkError(l.bundle, pe);
33.        }
34.      }
35.    }
36.  }
37.  pointcut Monitor_addFrameworkListener(BundleContextImpl bc,
FrameworkListener li):
38.    execution(void
BundleContextImpl+.addFrameworkListener(FrameworkListener))
39.    && target(bc) && args(li);

```

Figure 46. The aspect to enforce Observer Pattern for framework events 1-39

```

40. after(BundleContextImpl bc, FrameworkListener li):
41.     Monitor_addFrameworkListener(bc, li) {
42.         bc.framework.listeners.addFrameworkListener(bc.bundle, li);
43.     }
44. pointcut Monitor_removeFrameworkListener(BundleContextImpl bc,
FrameworkListener li):
45.     execution(void
BundleContextImpl+.removeFrameworkListener(FrameworkListener))
46.         && target(bc) && args(li);
47. after(BundleContextImpl bc, FrameworkListener li):
48.     Monitor_removeFrameworkListener(bc, li) {
49.         bc.framework.listeners.removeFrameworkListener(bc.bundle, li);
50.     }
51. pointcut removeBundleResources(BundleImpl bi):
52.     execution(void removeBundleResources()) && this(bi);
53. before(BundleImpl bi): removeBundleResources(bi) {
54.     bi.framework.listeners.removeAllListeners
(bi.framework.listeners.frameworkListeners, bi);
55. }
56. pointcut M_unregister0(Bundle b, ServiceRegistrationImpl sri):
57.     call (* ServiceFactory+.ungetService(Bundle,
ServiceRegistration, Object))
58.         && args(b, sri, ..) && withincode(void
ServiceRegistrationImpl+.unregister0());
59. after(Bundle b, ServiceRegistrationImpl sri)
throwing(Throwable ue): M_unregister0(b, sri) {
60.     sri.bundle.framework.listeners.frameworkEvent(new
FrameworkEvent(FrameworkEvent.ERROR, b, ue));
61. }
62. pointcut Monitor_getService(ServiceReferenceImpl si, Bundle b):
63.     call (Object ServiceFactory+.getService(Bundle,
ServiceRegistration))
64.         && target(si) && args(b, ..);
65. after(ServiceReferenceImpl si, Bundle b)throwing(Throwable pe):
66.     Monitor_getService(si, b) {
67.         si.registration.bundle.framework.listeners.frameworkError
(si.registration.bundle, pe);
68.     }
69. pointcut Monitor_SIunregister0(BundleImpl bl,
ServiceRegistration registration):
70.     call (* ServiceFactory+.ungetService(Bundle,
ServiceRegistration, Object))
71.         && args(bl, registration, ..) && withincode(boolean
ServiceReferenceImpl+.ungetService(..));
72. after(BundleImpl b, ServiceRegistration registration)
throwing(Throwable e): Monitor_SIunregister0(b, registration) {
73.     b.framework.listeners.frameworkError
(((ServiceRegistrationImpl)registration).bundle, e);
74. }}

```

Figure 47. The aspect to enforce Observer Pattern for framework events 40-74

It is worthy to stress the improvements in terms of code locality and composition transparency by implementing the Observer Pattern with AOP at the modeling level. Without AOP, when we want to turn on/off the feature of framework event management, it requires at least 12 modifications scattered in 6 files: Framework.java, ServiceRegistrationImpl.java, BundleContextImpl.java, BundleImpl.java, Listeners.java and ServiceReferenceImpl.java. However, with AOP, turning on/off this feature is as simple as including this aspect in AspectJ weaving! Again, such modularity and composition transparency are critical, considering the exponential increase of possible combinations of features.

It is worth noticing that the Aspect-Oriented framework event management can also be implemented by instantiating the Aspect-Oriented Observer Protocol from (Hannemann 2002) (described in Section 6), which allows code reuse as well as design reuse. However, for the ease of performance comparison with its Java counterpart, we implement it without using the existing library.

7.2.2. ServiceListener

As introduced in Section 2 and Section 3, the OSGi framework provides an important feature to allow sharing functionalities through services. To enable this, OSGi applications need to communicate with each other, which is implemented through the subscription to service events in an Observer Pattern. The management of service events can be modularized as an aspect, as shown in Figure 48 and 49.

```

1. public privileged aspect ServiceListenerAJ {
2.   private ServiceListenerState listeners.serviceListeners = new
   ServiceListenerState();
3.   void listeners.addServiceListener(Bundle bundle,
   ServiceListener listener, String filter) {
4.     try {
5.       serviceListeners.add(bundle, listener, filter);
6.     } catch (InvalidSyntaxException ex) {
7.       throw new RuntimeException(ex.toString());
8.     }
9.   }
10.  void listeners.removeServiceListener(Bundle bundle,
   ServiceListener listener) {
11.    serviceListeners.remove(bundle, listener);
12.  }
13.  public void listeners.serviceChanged(final ServiceEvent evt)
14.  { // broadcast service events }
15.  pointcut removeBundleResources(BundleImpl bi):
16.    execution(void removeBundleResources()) && this(bi);
17.  before(BundleImpl bi): removeBundleResources(bi) {
18.    bi.framework.listeners.serviceListeners.removeAll(bi);
19.  }
20.  private void removeAllListeners(Set s, Bundle b)
21.  { // remove all listeners }
22.  static boolean nocacheldap = false;
23.  pointcut Monitor_addServiceListener(BundleContextImpl bc,
   ServiceListener li, String filter):
24.    execution(void addServiceListener(..))
25.    && this(bc) && args(li, filter);
26.  after(BundleContextImpl bc, ServiceListener li, String
   filter):
27.    Monitor_addServiceListener(bc, li, filter) {
28.    bc.framework.listeners.addServiceListener(bc.bundle, li,
   filter);
29.  }
30.  pointcut Monitor_addServiceListener2(BundleContextImpl bc,
   ServiceListener li):
31.    execution(void
   BundleContextImpl+.addServiceListener(ServiceListener))
32.    && target(bc) && args(li);
33.  after(BundleContextImpl bc, ServiceListener li):
34.    Monitor_addServiceListener2(bc, li) {
35.    bc.framework.listeners.addServiceListener(bc.bundle, li,
   null);
36.  }

```

Figure 48. The aspect to enforce Observer Pattern for service events 1-36

```

37. pointcut Monitor_removeServiceListener(BundleContextImpl bc,
ServiceListener li):
38.     execution(void
BundleContextImpl+.removeServiceListener(ServiceListener))
39.         && target(bc) && args(li);
40. after(BundleContextImpl bc, ServiceListener li):
41.     Monitor_removeServiceListener(bc, li) {
42.         bc.framework.listeners.removeServiceListener(bc.bundle, li);
43.     }
44. pointcut Monitor_setProperties(ServiceRegistrationImpl sri):
45.     execution (void setProperties(Dictionary))
46.         && this(sri);
47. after(ServiceRegistrationImpl sri): Monitor_setProperties(sri) {
48.     sri.bundle.framework.listeners.serviceChanged(new
ServiceEvent(ServiceEvent.MODIFIED, sri.reference));
49. }
50. pointcut
Monitor_unregister_removeService(ServiceRegistrationImpl sri):
51.     call (void
ServiceRegistrationImpl+.unregister_removeService()) && target(sri);
52. before(ServiceRegistrationImpl sri):
Monitor_unregister_removeService(sri) {
53.     if (Framework.UNREGISTERSERVICE_VALID_DURING_UNREGISTERING) {
54.         sri.bundle.framework.listeners.serviceChanged(new
ServiceEvent(ServiceEvent.UNREGISTERING, sri.reference));
55.     }
56. }
57. after(ServiceRegistrationImpl sri):
Monitor_unregister_removeService(sri) {
58.     if (!Framework.UNREGISTERSERVICE_VALID_DURING_UNREGISTERING) {
59.         sri.bundle.framework.listeners.serviceChanged(new
ServiceEvent(ServiceEvent.UNREGISTERING, sri.reference));
60.     }
61. }
62. pointcut Monitor_ServicesRegister(BundleImpl bundle):
63.     execution (ServiceRegistration Services+.register(BundleImpl,
String(), Object, Dictionary)) && args(bundle, ..);
64. after(BundleImpl bundle) returning(ServiceRegistration sr):
Monitor_ServicesRegister(bundle) {
65.     bundle.framework.listeners.serviceChanged(new
ServiceEvent(ServiceEvent.REGISTERED, sr.getReference()));
66. }
67.}
68.class ServiceListenerEntry extends ListenerEntry { ... }
69.class ServiceListenerState { ... }

```

Figure 49. The aspect to enforce Observer Pattern for service events 37-69

In Figure 48 and 49, line 2-14 defines the basic access functions to manage service event subscription: line 2 adds a placeholder to store all service subscribers; line 3-12 adds routines to add/remove a service listener; line 13-14 adds routines to broadcast a service event. These definitions reside in `Listeners.java` in the original modeling framework. Line 15-21 adds routines to remove all service listeners in different contexts, which correspond to routines in `BundleImpl.java` in the original implementation. Line 23-43 modularizes service event management routines from `BundleContextImpl.java`. As explained, to minimize changes needed in OSGi applications, we keep the function interfaces of `BundleContextImpl` unchanged; the actual functionalities of these functions are implemented by the aspect. For example, line 23-36 actually adds a service listener as defined by the function interfaces, and line 37-43 actually removes a service listener as defined by the function interface. Line 44-61 modularizes service event management invoked in `ServiceRegistrationImpl`: in line 48, a `service_modified` event is broadcast after a service property is changed; in line 53-55 and line 58-60, depending on the configuration, a `service_unregistering` event is broadcast before or after a service is removed. Line 62-66 modularizes service event management invoked in `Services`: in line 65, a `service_registered` event is broadcast after a service is successfully registered in the OSGi framework. `ServiceListenerEntry` and `ServiceListenerState` are two helper classes used to manage service events.

Again, it is worth stressing the improvements of modularity and composition transparency: without AOP, to include the management of service events, one would

have to make modifications in at least 14 places across 6 files; with AOP, such modifications is as simple as plugging in the aspect in AspectJ weaving.

7.2.3. BundleContext Validator

The need to check the validity of a `BundleContext` instance arises at least in the following situation: a `BundleActivator` implementation may create a thread and pass an instance of `BundleContext` to it. If the thread is still alive when the ancestor bundle has been uninstalled, the instance of `BundleContext` becomes a dangling bundle context; invoking the functionality of a dangling bundle context has undefined consequences. To assure well-defined behaviors in face of an invalid bundle context, in the Knopflerfish implementation, it associates the ancestor bundle to the bundle context as an indicator of context validity.

Such validation of bundle context can be modularized in an aspect, as `BCValidator` shown in Figure 50. In line 4-19, it catches 15 joinpoints in `BundleContextImpl`, where the validity of the bundle context influences the outcome of the function invocation, i.e. it should throw an `IllegalStateException` if the bundle context is invalid. If there is a valid input stream, it should also close the input stream (line 33-44).

Again, without AOP, we would have to make modifications in 16 places in the source code of `BundleContextImpl.java`; using AOP, such modification is as simple as a configuration whether to include `BCValidator.aj` in AspectJ weaving.

```

1. public privileged aspect BCValidatorAJ {
2.     pointcut p_isBCvalid(BundleContextImpl bc):
3.         within(BundleContextImpl) && (
4.             execution(String getProperty(String)) ||
5.             execution(Bundle installBundle(String) throws BundleException)
6.             || execution(Bundle getBundle())
7.             || execution(void addServiceListener(..))
8.             || execution(void removeServiceListener(..))
9.             || execution(void addBundleListener(..))
10.            || execution(void removeBundleListener(..))
11.            || execution(void addFrameworkListener(..))
12.            || execution(void removeFrameworkListener(..))
13.            || execution(void registerService(..))
14.            || execution(* getServiceReferences(..))
15.            || execution(* getAllServiceReferences(..))
16.            || execution(* getServiceReference(..))
17.            || execution(* getService(..))
18.            || execution(boolean ungetService(..))
19.        ) && this(bc);
20.     before(BundleContextImpl bc): p_isBCvalid(bc) {
21.         isBCvalid(bc);
22.     }
23.     private void isBCvalid(BundleContextImpl bc) {
24.         if (bc.bundle==null) {
25.             throw new IllegalStateException("This bundle context is no
26. longer valid");
27.         }
28.     }
29.     pointcut p_isBCvalid2(BundleContextImpl bc, InputStream in):
30.         execution(Bundle installBundle(String, InputStream) throws
31. BundleException)
32.         && args(String, in)
33.         && this(bc);
34.     before(BundleContextImpl bc, InputStream in):
35.         p_isBCvalid2(bc, in) {
36.             try {
37.                 isBCvalid(bc);
38.             } finally {
39.                 if (in!=null) {
40.                     try {
41.                         in.close();
42.                     } catch (IOException ex) {
43.                         // ignore
44.                     }
45.                 }
46.             }

```

Figure 50. The aspect to check the validity of a bundle context

7.3. Summary of Improvements

I summarize the changes needed to vary a feature of the OSGi modeling framework. As shown in Table 4, each row presents an optional feature: the first column is the number of modifications needed to add/remove this feature, and the second column describes the files to be modified to add/remove this feature. To come up with a particular combination of features, we would have to make $12+14+16+16+23 = 81$ modifications for one of the $2^5 = 32$ possible combinations, which cross the boundary of methods and classes of more than 9 files.

Table 4. Modifications needed to add/remove a feature

	Number of modifications	Files to be modified
Framework Listener	12	Framework.java, BundleImpl.java, BundleContextImpl.java, Listeners.java, ServiceRegistrationImpl.java, ServiceReferenceImpl.java
Service Listener	14	Listeners.java, Services.java, BundleContextImpl.java, ServiceRegistration.java, ServiceListenerEntry.java
Bundle Listener	16	Framework.java, Listeners.java, BundleContextImpl.java, BundleImpl.java, Bundles.java
BundleContext Validator	16	BundleContextImpl.java
Permission	23	Framework.java, Listeners.java, Services.java, BundleImpl.java, BundleContextImpl.java, Bundles.java, ServiceReferenceImpl.java, ServiceRegistrationImpl.java
Total	81	BundleImpl.java, Bundles.java, BundleContextImpl.java, Framework.java, Listeners.java, Services.java, ServiceReferenceImpl.java, ServiceRegistrationImpl.java, ServiceListenerEntry.java

It is a daunting task to make so many modifications that scatter across the modular boundary provided by Object-Oriented technologies. Furthermore, it breaks the encapsulation of a framework and forces a developer to use a framework in a white-box approach. On the contrary, as we have shown in Section 4 and 7, each of these optional features can be modularized as an aspect using Aspect-Oriented techniques, and the addition/removal of a feature is as simple as a command line option to include/exclude an aspect in AspectJ weaving. Apparently, the composition transparency of AOP techniques significantly eases the specialization of a modeling framework to the desired proper abstraction level: varying the abstraction level of a modeling framework is almost “free”!

In Section 5, I have shown the performance overhead and the optimization techniques to minimize such overhead when I introduce AOP techniques to model checking. In the rest of this section, I show the result of applying AOP techniques to reduce the modeling framework to various abstraction levels. Some of these results are summarized in Table 5.

Table 5. Comparison of state space of different configurations

	No feature	Service Added	Bundle Added	Framework Added	Bundle Context Validator Added	Max/Min
Example2	1094	2723	7206	8143	17098	15.63
Example2-2	1856	4958	12394	14674	24458	13.18
Example4	-	1249	3155	3495	5181	4.15

In Table 5, we conducted the performance comparison using three open-source programs from (Oscar). The first column corresponds to the search space without any feature (the intersection of the first column and the third row is empty as the Example4 requires at least the Service Listener feature); thereafter, each column has one more feature in the modeling framework than its preceding column, i.e. Service Listener, Bundle Listener, Framework Listener and Bundle Context Validator, respectively. The last column shows the ratio of search space obtained from full-featured modeling framework vs. the minimal possible modeling framework. From the table, we can see the general trend is that when more features are added, JPF needs more space before it finds the bugs. For the first row (Example2), the search space doubles when a redundant feature is added; a full-fledged modeling framework may need 15 times of search space compared with the minimum possible model! The verification demonstrates that we can effectively reduce the search space by selecting only related features (aspects), instead of using a fixed, monolithic modeling framework. This assertion is further validated in the second row: the search space of the base case is about 13 times of the least model, and its search space increases more than doubled when a redundant feature (ServiceListener or BundleListener) is added. The search space increases relatively moderately in the third row (Example4), but the full-fledged version still has a search space 3 times more than the minimum version. This could have a large impact on the completeness of the verification result.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

8.1. Summary

In this dissertation, I have shown that for applications in the OSGi domain, it is possible to build a modeling framework parallel to the OSGi specification to ease the model construction of OSGi applications. With such an approach, I am able to discover property violations in these applications through model checking, with relatively small efforts to formalize the applications; the modeling framework has the same slots and hooks for extensions as in the application framework.

Crosscutting concerns arise when I try to vary the details of a modeling framework to minimize the state space. Varying features of a modeling framework often requires modifications that break the Object-Oriented modularity and scatter in multiple functions and classes. To resolve such crosscutting concerns, I propose to use *Aspect-Oriented Programming (AOP)* techniques to modularize the needed changes. With AOP techniques, we pre-process a generic modeling framework with pointcuts, inter-type definition and other AOP constructs. The pre-processing step adds the only needed details to the bare essential. In this way, I am able to specify variations in an explicit and

modular way. Varying application details then becomes a simpler task of specifying and selecting the proper combination of aspects.

Two problems arise due to the adoption of AOP techniques, i.e. the native code and the performance penalty. I have created an abstraction library to resolve the native code in the AspectJ runtime library as needed, and came up with a test suite in JUnit for regression testing. The performance penalty is due to the internal variables used by aspect transformation and subsequent interleaving in a multi-threaded system. We thus made a distinction between model checking AspectJ programs in general and using AOP techniques to vary an existing formal model. For the latter problem, we presented several approaches to reduce the search space to a comparable size of its counterpart in pure Java implementation.

We have thus used the above technologies for rapid construction of formal models of OSGi applications. We are able to uncover several bugs in some benchmark OSGi applications with comparable search space. On the other hand, the extra interleaving of AspectJ programs suggests a potential data race. We uncovered bugs in the general Aspect-Oriented programming patterns that have been widely accepted and used. We showed a general solution to avoid data race during aspect instantiation for a class of applications under the current implementation of AspectJ compilers.

8.2. Lessons to Learn

Our experience in framework-based model construction with AOP assistance can benefit the software engineering research community in several ways.

First, for developers of formal models, our case studies propose a feasible methodology to build formal models in a domain-specific approach. By using a generic modeling framework, we are able to leverage modeling efforts from experts and avoid duplicated efforts in modeling commonalities in the particular domain. A modeling framework brings us merits of design reuse and code reuse, similar to the application level. By adopting AOP techniques, we are able to leverage the composition transparency in aspects weaving, thus solve the crosscutting concerns when varying the modeling framework. By designing the modeling framework with the Façade Pattern, we are able to encapsulate application models from varying details of the modeling framework.

We believe that with proper tool support, one can build a modeling framework that can be specialized for application logic as well as the abstraction level. In addition, specialization can be modularized with AOP techniques without unnecessary overheads. With the increased popularity of Aspect-Oriented techniques, we foresee that formal-model developers should have a flat learning curve to framework-based model construction with AOP assistance.

Second, for authors of model checking tools, our case studies may spark the ideas to build special AOP tools at the model level. We have shown that to use a regular AspectJ compiler at the model level, one has to customize the AspectJ runtime library to build some replacement for system-level methods. We also show that the extra fields (e.g. aspect instances) and statements (e.g. aspect instantiation) inserted by AspectJ weaving may lead to undesired interleaving and performance overhead, and one has to apply additional optimization techniques to eliminate such overhead. While the AspectJ

runtime library and the aspect-specific fields and statements evolve over time with sound justifications (e.g. clarity, consistency, flexibility and ease of garbage collection for aspect instantiation, as pointed out in (Colyer 2004)), their benefits may not outweigh the complexity to eliminate performance overhead at the model level. It is thus important to make a distinction between using AOP techniques to ease model construction, and model checking Aspect-Oriented programs in general. For our purpose of using AOP techniques to vary a modeling framework, we don't need a full-fledged AspectJ compiler – a simplified AspectJ compiler with basic AOP functionalities may suffice for our purpose. For example, the following AOP primitives suffice for our case studies: Intertype definition (adding class attributes, changing inheritance hierarchy), pointcut (call, execution, within, withincode, if, this), advice (before, around, after), and issingleton aspect instantiation. While it is hard to say that such a subset would suffice all needs in aspectizing formal models, it is critical to balance the benefits and the cost (e.g. performance overhead, complexity of woven bytecode) when choosing AOP features to create a special tool at the model level. For example, we may create a special AspectJ compiler that has no management of aspect instantiation and directly insert the advised code before and after the designated joinpoint. In this way, the bytecode woven from the AspectJ version is exactly the same as its pure Java counterpart, which is exactly the effect we are looking for!

Third, we also have some lessons to share with the authors of AOP tools and AOP developers. For the consideration of performance, the authors of AOP tools may deliberately omit the synchronization control of aspect instantiation. This is thoughtful,

just like the authors of the JDK library to omit the synchronization control for basic utility classes in the `java.util` package (for example, `java.util.PriorityQueue` is not thread-safe). However, the JDK library evolves to provide a thread-safe version of queue class (e.g. `java.util.concurrent.BlockingPriorityQueue`); analogously, it is desired to have a different aspect instantiation that is thread-safe. The authors of AOP tools may take such concerns into consideration and provide AOP primitives that are safe in a multi-threaded environment. On the other hand, our work also helps programmers to develop robust AspectJ programs. Many (if not most) AspectJ programmers are not aware of the lack of synchronization during aspect instantiation (the severity of this problem can be evidenced by the examples in Section 6), nor do they know the solution to avoid this trap. We have shown a general pattern to transform *perthis* and *pertarget* aspect instantiation into *issingleton* style, which is thread-safe as verified by model checking. AspectJ programmers can follow our examples to develop robust AspectJ programs that are free from concurrency problems caused by aspect instantiation.

Many of our results also directly benefit users in particular domains. For example, our aspectized modeling framework of OSGi can be directly reused by formal-model developers to check OSGi applications. Our abstraction library for the AspectJ runtime library can be reused to enable model checking AspectJ programs by JPF. Our optimization techniques can be reused when one needs to reduce the search space of AspectJ programs. Those bugs discovered in our examples (e.g. OSGi applications,

Aspect-Oriented design patterns) also help developers to build robust applications for their purpose.

8.3. Future Work

Among the success to uncover bugs in OSGi applications, there remain several open problems. First, not all AspectJ primitives have been resolved. In particular, to support the following AspectJ primitives, *cflow* and *percflow*, the AspectJ compiler needs to access the calling context at the bytecode level and use a stack-like structure to keep the related information. Using the current implementation as it is raises runtime exceptions. These remain to be done to fully support all AspectJ features. Second, I have used an aspect to present each feature, and show that we can select a particular combination of features by choosing a particular combination of aspects. Similar to the application level, there remain open problems about interleaving among different aspects. For example, the behavior of a feature (aspect) may be changed when another feature is present. Such issues of feature confliction remain an open problem to us. Furthermore, researchers have used program analysis techniques to slice fields and statements unrelated to the correctness criteria. I am interested in applying similar analysis techniques to automate the discovery and construction of aspects at the model level.

BIBLIOGRAPHY

- ABC, abc: The AspectBench Compiler for AspectJ. <http://abc.comlab.ox.ac.uk/>. Accessed 5 September 2007
- AJC, The Eclipse AspectJ project web site. <http://www.eclipse.org/aspectj>. Accessed 19 March 2007
- Andoni, A., Daniliuc, D., Khurshid, S., Marinov, D.: Evaluating the “small scope hypothesis”. Technical Report MIT-LCS-TR-921, MIT CSAIL (2003)
- Avustinov, P., Christensen, A., Hendren, L., Kuzins, S., Lhotak, J., Lhotak, O., Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Abc: an extensible AspectJ compiler. In: Proc. the 4th Int. Conf. on Aspect-Oriented Softw. Development, Chicago, Illinois, pp. 87-98 (2005)
- Barjaktarovic, M.: The state-of-the-art in formal methods. <http://www.cs.utexas.edu/users/csed/FM/docs/StateFM.pdf>. Accessed 1 March 2006. Copyright of Wilkes University and Rome Research Site (1998)
- BCEL, The Byte Code Engineering Library (BCEL) project web site. <http://jakarta.apache.org/bcel/>. Accessed 1 September 2007
- Chen, Z., Fickas, S.: The plain old television in a smart apartment. In: Proc. the 1st Int. Conf. on Collaborative Computing: Networking, Applications and Worksharing (CollaborateComm '05), San Jose, California (2005)
- Chen, Z., Fickas, S.: do no harm: model checking eHome applications. In: 1st Int. Workshop on Softw. Eng. of Pervasive Computing Applications, Systems and Environments (SEPCASE '07), Minneapolis, Minnesota (2007)
- Chen, Z.: The source code for the manuscript, including the aspectized OSGi modeling framework, the extension to JPF, and the examples of Aspect-Oriented design patterns. Accessible <http://www.cs.uoregon.edu/~zbchen/dissertation/source/> (2008).
- Colyer, A., Clement, A., Harley, G., Webster, M.: Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison-Wesley, Upper Saddle River, New Jersey (2004)

- Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.:
Bandera: extracting finite-state models from java source code. In: Proc. the 22nd Int.
Conf. on Softw. Eng., Limerick, Ireland, pp. 439-448 (2000)
- Demartini, C., Iosif, R., Sisto, R.: dSPIN: a dynamic extension of SPIN. In: Proc. the 6th
SPIN Workshop, Toulouse, France, Lecture Notes in Computer Science, vol. 1680,
pp. 261-276. Springer, Berlin (1999)
- Deng, W., Dwyer, M., Hatcliff, J., Jung, G., Robby: Model-checking middleware-based
event-driven real-time embedded software. In: Proc. the 1st Int. Symp. on Formal
Methods for Components and Objects (FMCO '02), Leiden, The Netherlands, pp.
154-181 (2002)
- Dennis, G., Chang, F., Jackson, D.: Modular verification of code with sat. In: Proc. the
Int. Symp. on Softw. Testing and Analysis, Portland, Maine, pp.109-120 (2006)
- Dwyer, M., Avrunin, G., Corbett, J.: Patterns in property specifications for finite-state
verification. In: Proc. the 21st Int. Conf. on Softw. Eng., Los Angeles, California,
pp. 411-420 (1999)
- Dwyer, M., Hatcliff, J., Robby, Joehanes, R., Laubach, S., Pasareanu, C., Zheng, H.
Visser, W.: Tool-supported program abstraction for finite-state verification. In: Proc.
the 23rd Int. Conf. on Softw. Eng., Toronto, Ontario, Canada, pp. 0177-0188 (2001)
- Dwyer, M., Hatcliff, J., Hoosier, M., Ranganath, V., Robby, Wallentine, T.: Evaluating
the effectiveness of slicing for model reduction of concurrent object-oriented
programs. In: Proc. 12th Conf. on Tools and Algorithms for the Construction and
Analysis of Systems (TACAS '06), Vienna, Austria, pp. 73-89 (2006)
- Equinox, The Eclipse Equinox project web site. <http://www.eclipse.org/equinox/>.
Accessed 1 September 2007
- Fayad, M., Schmidt, D., Johnson, R.: Building Application Frameworks: Object-Oriented
Foundations of Framework Design. John Wiley & Sons, New York (1999)
- Fayad, M., Schmidt, D., Johnson, R.: Implementing Application Frameworks: Object-
Oriented Frameworks at Work. John Wiley & Sons, New York (1999)
- Feather, M., Fickas, S., Razermera, A.: Model-checking for validation of a fault
protection system. In: Proc. IEEE Int. Symp. on High Assurance Systems
Engineering, Boco Raton, Florida, pp. 32-41 (2001)
- Felix, The Apache Felix project web site. <http://felix.apache.org/site/index.html>.
Accessed 18 March 2007

- Fickas, S., Pataky, C., Chen, Z.: DuckCall: tracking the first hundred yards problem. In: Proc. 8th Int. ACM SIGACCESS Conf. on Computers and Accessibility, Portland, Oregon, pp. 283-284 (2006)
- Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
- Garlan, D., Khersonsky, S., Kim, J. S.: Model checking publish-subscribe systems. In: Proc. 10th Int. SPIN Workshop on Model Checking of Softw., Portland, Oregon. Lecture Notes in Computer Science, vol. 2648, pp. 166-180. Springer, Berlin (2003)
- Griesmayer, A., Bloem, R., Hautzendorfer, M., Wotawa, F.: Formal verification of control software: a case study. In: Proc. the 18th Int. Conf. on Innovations in Applied Artificial Intelligence, Bari, Italy. Lecture Notes in Computer Science, vol. 3533, pp. 783-788. Springer, Berlin (2005)
- Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: Proc. the 17th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications, Seattle, Washington, pp. 161-173 (2002)
- Hatcliff, J., Deng, X., Dwyer, M., Jung, G., Ranganath, V.: Cadena: an integrated development, analysis, and verification environment for component-based systems. In: Proc. the 25th Int. Conf. on Softw. Eng., Portland, Oregon, USA, pp. 160-173 (2003)
- Heitmeyer, C., Kirby, J., Labaw, B., Archer, M., Bharadwaj, R.: Using abstraction and model checking to detect safety violations in requirement specifications. IEEE Trans. Softw. Eng., 24(11), 927 – 948 (1998)
- Holzmann, G.: The model checker spin. IEEE Trans. Softw. Eng., 23(5), 279-295 (1997)
- Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Boston, Massachusetts (2003)
- Iosif, R.: Exploiting heap symmetries in explicit-state model checking of software. In: Proc. 16th IEEE Conf. on Automated Softw. Eng., San Diego, California, pp. 254-261 (2001)
- Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. and Methodology (TOSEM), 11(2), 256-290 (2002)
- JUnit, The JUnit project web site. <http://www.junit.org/>. Accessed 1 March 2007.
- Knopflerfish, The Knopflerfish framework project web site. <http://www.knopflerfish.org/>. Accessed 21 September 2007

- Lamsweerde, A.: Formal specification: a roadmap. In: Proc. Int. Conf. of Softw. Eng. (The Future of Softw. Eng.), Limerick, Ireland, pp. 147-159 (2000)
- Lea, D.: Concurrent Programming in Java: Design Principles and Patterns (2nd edition). Addison-Wesley, Reading, Massachusetts (1999)
- Lethbridge, T., Laganier, R.: Object-Oriented Software Engineering. McGraw-Hill Education, Maidenhead, Berkshire, England (2001)
- Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. John Wiley & Sons, Chichester, West Sussex, England (1999)
- Markosian, Z., Mansouri-Samani, M., Mehlitz, C., Pressburger, T.: Program model checking using design-for-verification: NASA flight software case study. In: Proc. IEEE Aerospace Conf., Big Sky, Montana, pp.1-9 (2007)
- Mehlitz, P., Visser, W., Penix, J.: The JPF Runtime Verification System. <http://sourceforge.net/projects/javapathfinder>. Accessed 4 June 2006
- Nielson, F., Nielson, H., Hankin, C.: Principles of Program Analysis. Springer-Verlag, New York (1999)
- Oscar, The Oscar tutorial web site. <http://oscar-osgi.sourceforge.net/>. Accessed 18 September 2007
- OSGi, The OSGi Alliance: OSGi Service Platform Core Specification (Release 4). <http://www.osgi.org/>. Accessed 15 November 2005 (2005)
- Robby, Dwyer, M., Hatcliff, J.: Bogor: an extensible and highly-modular model checking framework. In: Proc. the 4th Joint Meeting of the European Softw. Eng. Conf. and ACM SIGSOFT Symp. on the Foundations of Softw. Eng. (ESEC/FSE '03), Helsinki, Finland, pp. 267-276 (2003)
- Smith, R., Avrunin, G., Clarke, L., Osterweil, L.: PROPEL: an approach supporting property elucidation. In: Proc. the 24th Int. Conf. on Softw. Eng., Orlando, Florida, pp. 11-21 (2002)
- Taghdiri, M., Jackson, D.: A lightweight formal analysis of a multicast key management scheme. In: Proc. the 23rd IFIP Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE). Lecture Notes in Computer Science, vol. 2767, pp. 240-256. Springer, Berlin (2003)
- Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Softw. Eng., 10(2), pp. 203-232 (2003)

- Wayt Gibbs W.: Software's chronic crisis. Scientific American.
<http://www.cis.gsu.edu/~mmoore/CIS3300/handouts/SciAmSept1994.html>.
Accessed in March 1 2007 (1994)
- Xie, F., Browne, J.: Verified systems by composition from verified components. In: Proc. the 9th European Softw. Eng. Conf. held jointly with 11th ACM SIGSOFT Int. Symp. on Foundations of Softw. Eng., Helsinki, Finland, pp. 277-286 (2003)

)

)

)

)

)

)

)

)

)

)

)