

Research Statement

Raffi Khatchadourian

Overview

Software is typically conceived in order to deal with a set of *concerns*, i.e., units of functionality the system is responsible for realizing. In a rapidly changing world, we often tend to demand more and more from our software, requiring systems to cope with a wide variety of increasingly complex concerns. Such complexity can lead to software implementations that are difficult to verify, comprehend, maintain, and evolve. These activities are important in all classifications of software, but especially ones that are considered *critical*, that is, having real-time constraints or whose correct functioning is of the utmost importance.

In an effort to combat these complexities, software is conventionally built in a modular fashion with each unit corresponding to a specific concern. As an example, consider an international online shopping application that is concerned with placing and managing orders, shipping products, mediating auctions, etc. Suppose that a change in the sales tax rate has occurred in a discrete country and that the checkout functionality of the application should account for this change. Consequently, we would expect that the implementation of a particular module, say the *shopping cart*, to be altered so that upon checkout, if the purchasing customer resides in that country then the sales tax rate applied is that of the new value. Such a modification is considered *localized* in that the policies governing a specific concern are implemented in a single module, and that evolving the software in respect to that concern does not affect unrelated portions of the software. This ability to perform localized modifications is crucial especially in large, sophisticated systems, as it requires minimal alterations to the source code and thus reduces the probability of introducing new defects.

However, not every kind of concern can be isolated in such a manner. To contrast the aforementioned scenario, imagine that our hypothetical shopping application is also concerned with the logging of events (e.g., orders, shipping, quantity fluctuations, sales trends) that occur during its execution. It is easy to recognize that such a concern applies to *many* portions of the software, and that any potential adjustment to this concern will *not* result in a source code modification that is localized. Concern implementations of this kind are *scattered* throughout the software in that changing any policies governing that concern requires altering the implementations of all modules to which it applies. In relation to our example, suppose that we notice that the output of logged events becomes too verbose, making it difficult to analyze. Ergo, we would like to alter our event logging policy so that only certain, more interesting events are logged in hopes of reducing the amount of event records. Producing this modification, however, requires examining each place in each module where logging should apply and transforming the implementations at these points so that our new system behaves in a way such that events are logged conditionally. Worse yet, concerns like event logging are characteristically *tangled* with other concerns in that understanding a single, isolated concern implementation requires filtering through the implementation of the event logging concern. Again in relation to the example given above, altering the shopping cart module in order to account for the tax rate change would require mentally separating and categorizing each program statement implementing the checkout functionality with those implementing the event logging functionality. Clearly, such an activity would make it difficult to understand the substance of each concern as it is realized in the software, verify the correctness of each implementation, and to evolve the software effectively. Concerns of this sort are denoted as *crosscutting concerns* in that they inherently *crosscut* traditional modular boundaries.

Although providing facilities to help encapsulate implementations of *core* functionality (e.g., a shopping cart), traditional software development techniques prove inadequate in properly encapsulating, separating, and localizing the implementations of crosscutting concerns. As a result, these concern implementations become *scattered* and *tangled* with the software's main logic, having detrimental effects on key facets of the software development process. Aspect-Oriented programming (AOP) [13] has

emerged in order to provide the proper machinery to help encapsulate crosscutting concern implementations by allowing developers to specify “aspects” as programming language constructs. An aspect is a single, modular programming unit that corresponds to a particular crosscutting concern (e.g., event logging) and consists of two essential parts, the code that *realizes* the concern (i.e., “what to do”) and a set of well-defined points in the software’s execution designating where the code applies (i.e., “when to do it”). Since its inception, various authors [5, 16, 17, 18] have shown how aspects may be used to write *localized* implementations for such important, crosscutting concerns as process synchronization, event logging, data persistence, exceptional situation handling, etc. Despite its increasing popularity, a number of researchers [1, 2, 3, 14, 15] have raised serious questions about the problems AOP poses for modular reasoning. That is, while AOP allows for modular *implementations* of crosscutting concerns, it is yet unclear if AOP can be *reasoned* about in a modular way. In this context, “reasoning” refers to both informal checks, including desk-checks of source code, and formal proofs of the correct functioning of software by either developers themselves or by automated tools that assist programmers in developing error-free software. The problem is that, by separating software in this fashion we are essentially creating *two* different systems, a *baseline* system (also known as the “base-code”) that solely includes the core logic and an *augmented* system which is the result of applying aspects (an activity called *advising*) that alter the behavior of the baseline. Indeed, the ability of an aspect to change the behavior of the base-code that it advises, which is the very reason for much of the power of AOP, is also what causes difficulties for reasoning about the behavior of such software. Since the addition of the aspect changes the behavior of the base-code, whatever reasoning we may have done about the base-code may no longer be valid. In fact, we may be forced to reason about the entire system, accounting for the interleaved execution of various pieces of advice with the base-code. My proposal involves developing an approach to reasoning about the base-code and the aspects that seems to offer important advantages over the existing approaches. A preliminary version of this work was presented at SPLAT '07 [11], OGSS-CISE '07 [8], and AOSD-SS '07[7]. Intermediate progress appears in [19].

Methodologies

Fortunately, AO programs share some resemblance in their reasoning challenges to concurrent programs. Consider a concurrent program with two parallel processes that share some variables that either of them may read or write. Standard modular reasoning would require us to reason about each process independently of the other and then combine the results of the two reasoning tasks in an appropriate manner to arrive at the behavior of the whole program. But since the two processes will be interleaved during execution, whatever conclusions we may have drawn about each of them when reasoning about them independently may not, in fact, be valid. In effect, the actions of each process may interfere with the other process thereby invalidating whatever results we may have established by reasoning about that other process. This is rather similar to the situation in AOP. Suppose, for example, that the base-code contains an assignment statement, assigning a specific value vv to a particular instance variable xx . When reasoning about this base-code, we might have established an assertion following the assignment, that states that the value of xx would, in fact, be equal to vv . Suppose now we add an aspect that includes a piece of *after*-advice that applies at a *set-join* point of the base-code and that the variable xx is one of the affected variables. Now, immediately following the execution of the assignment of vv to xx , the after-advice would execute and, possibly, assign a new value to xx before returning control to the base-code. At this point, the assertion we previously established in the base-code is no longer satisfied. In other words, the aspect has interfered with the base-code.

The rely-guarantee approach [6, 20] addresses the interference problem in parallel programs in an effective and modular fashion. In my current research, I explore the feasibility of adapting this approach to reasoning about AO programs. The rely-guarantee approach has proven extremely successful in reasoning about concurrent and distributed programs; and this approach, appropriately modified, may help address many of the reasoning problems induced by AOP. Specifically, I explore the key differences between the challenges associated with reasoning about concurrent programs, how the rely-guarantee approach deals these challenges, and how such an approach can be suitably adapted to provide an effective and useful reasoning system for AOP. Notably, there are several key disparities between the

situation in AOP and in that of concurrent programs. While in the case of parallel programs, two processes are typically designed hand-in-hand, in an AO program, the base-code is often written without any particular aspect in mind. In the same way, an aspect may not be written for the sole application to a certain base system. In adapting the rely-guarantee approach, we must take care to make it possible for each component (base-code or aspect) provider to reason about a single component independent of the environment in which it will be deployed. For instance, it should be possible for the base-code provider to reason about the base-code independent of particular aspects. Similarly, it must also be possible for the aspect provider to reason about the behavior of the aspect independent of a particular base system to which it may be applied.

Such a task is especially difficult since we must be able to draw useful conclusions about the behavior of each component (base and aspect) that will remain valid even after an introduction of *any* additional component. In other words, we would like our base-code reasoning to remain valid prior to the addition of any applicable advice, and likewise we would desire that any reasoning we have performed about a particular aspect to remain valid despite the addition of either a class *or* aspect, considering that an aspect itself may also be open to advice either from another aspect or possibly from the aspect itself. To achieve this, in the case of base-code, for example, the reasoning will essentially be parameterized over all possibly applicable aspects; that way, as an AO system evolves with various aspects being incorporated, altered, and removed, the base-code provider is not required to rebuild the associated reasoning. In fact, aspects being used in this fashion is consistent with the true spirit of AO systems, unequivocally that they possess a “plug-n-play” nature. In this sense, just as aspects augment a baseline system, deriving conclusions about the overall system (i.e., the base-code plus the aspects) will be obtained by “plugging-in” the behavioral specifications of the actual applicable aspects. We denote this activity as *enrichment* in that the addition of aspects are intended to *enrich* the behavior of the base-code they advise; indeed, it is the possibility of such enrichment that is the source of much of AOP’s power. For this purpose, we introduce the concept of *join point traces (JPTs)*. A JPT is used to record the various interactions of the base-code and aspects. Then, specifications of the AO system are in terms of assertions involving not just instance and local variables (and other parameters) but also in terms of the state changes recorded on the JPT. Finally, the reasoner *composes* the JPT-based specification with the actual behaviors of each component using an associated rule of composition to arrive at the resulting enriched behavior of the composed system.

One unfortunate consequence of such a flexible and adaptive reasoning system, however, is that specifications may become unwieldy. Accordingly, one key challenge of the approach is to reduce the amount of resulting complexity arising from the parameterized reasoning. In an effort to minimize this complexity, we leverage our notion of *specification pointcuts (s-pointcuts)*. Specification pointcuts are similar to normal pointcuts except that instead of associating code (i.e., advice) that is to be executed at the various join points included in the pointcut, the purpose of an s-pointcut is to specify properties of the interactions between components at those points. Specifically, each s-pointcut contains an associated *rely* and *guar* clause. The *rely* clause is an assertion specifying what the component provider is *relying on* to be true, and therefore should not be invalidated by the actions of any applicable advice at each join point. Conversely, the *guar* clause is an assertion identifying what the component provider, given that the *rely* clause is met, *guarantees* will be affected of the actions taken by the component itself at each join point. Each AO system is associated with an implicit default s-pointcut which encompasses all join points and whose constituent *rely* clause restricts *all* component interaction (i.e., $rely(\sigma, \sigma') \equiv (\sigma = \sigma')$ where σ denotes the state of the component immediately prior to a join point being advised, while σ' denotes the state of the component subsequent to execution continuing at the advised join point) and whose *guar* clause guarantees nothing (i.e., $guar(\sigma) \equiv \mathbf{true}$ where σ is as before). Although this default s-pointcut seems simultaneously both overly restrictive and lenient (in terms of each clause, respectively), component providers are allowed to explicitly declare additional s-pointcuts which, in the case of *rely*, relax the default constraint by disjunction of the *rely* clauses, and, in the case of *guar*, strengthen the default guarantee by conjunction of the *guar* clauses associated at each intersecting s-pointcut, respectively. The hope is that since the *rely* clause is used to restrict the kinds of interaction allowed at its corresponding join points, the complexity arising at the portion

of the JPT that is responsible for recording the interactions at these points is sufficiently tamed since only a limited amount of interaction is possible. The *guar* clause, on the other hand, is used to show that, upon composition of each component, the *rely* clauses accompanying each s-pointcut are indeed satisfied.

Goals

I aim to show that a modified rely-guarantee approach can enhance the effectiveness of current reasoning techniques as well as offer additional advantages. Prior approaches, such as [1, 3, 2, 4], have been made for restricting some of the useful facilities that AOP provides, and applying the proposed approach may considerably enhance their usefulness and expressiveness. My current research incorporates formalization of the approach (including proof rules), full development of the specification pointcut (*s-pointcut*) and join point traces (*JPTs*), which are at the heart of the proposal, a proof of soundness, and possible automated development tool integration. Current formalization utilizes an aspect calculi, thus, future work entails applying the formalization to a widely used AO language implementation such as AspectJ [12].

Auxiliary Interests

An auxiliary research interest of mine is techniques for automated refactoring of legacy Java source code, through the use of mechanisms such as static and dynamic analysis and type inferencing, to take advantage of new idioms and constructs that emerge with the evolution of programming languages and execution platforms. My interest lies in the creation of tools that successfully automate the source-to-source transformation necessary to facilitate the proper migration of existing systems to new technologies, thereby easing the costs of software maintenance over time. What makes creation of such automated tools challenging is that they should be created in a way so that they are generally applicable to all systems and situations. This often entails producing an approach that adopts a conservative methodology, however, defining the line of conservation is crucial for real-world applicability. Other interesting tasks of notable difficulty encompass preservation of semantics, various type-theoretical considerations, algorithmic efficiency, ease-of-use (i.e., using the tool should not be more difficult than conducting the refactoring manually), and inferring programmer intent (i.e., performing the refactoring as close as possible, if not better, to the way the refactoring would have been performed manually by the programmer).

My current research in this area comprises an automated refactoring of Java software which utilizes a popular legacy enumeration patterns to instead utilize the `Enum` construct introduced in Java 1.5, of which results appear in [9, 10]. Prior to the emergence of Java 1.5, programmers were required to employ various patterns to compensate for the absence of enumerated types in Java. Unfortunately, these compensation patterns lack several highly-desirable properties of the `Enum` construct, most notably, type safety. My proposal consists of a novel fully-automated approach for transforming legacy Java code to use the new enumeration construct. This semantics-preserving approach increases type safety, produces code that is easier to comprehend, removes unnecessary complexity, and eliminates brittleness problems due to separate compilation. At the core of the proposed approach is an interprocedural type inferencing algorithm which tracks the flow of enumerated values. The implementation of the proposal has been realized as an Eclipse IDE plug-in with a planned release with the standard distribution of Eclipse in March 2008. Current work necessitates the of augmentation of the approach to encompass grouping heuristics, analysis of reference types, and other analyses that capture and transform an increasing number of legacy enumeration patterns. Other ongoing research in this area involves refactoring of XML-based source code metadata to Java 1.5 annotations.

References

[1]J. Aldrich. Open modules: Modular reasoning about advice. In *Eur. Conf. Object-Oriented Programming*, 2005.

- [2]C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *International Workshop on Foundations of Aspect-Oriented Languages*, 2002.
- [3]C. Clifton, G. Leavens, and J. Noble. Mao: Ownership and effects for more effective reasoning about aspects. In *Eur. Conf. Object-Oriented Programming*, 2007.
- [4]D. Dantas and D. Walker. Harmless advice. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.
- [5]X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. SyncGen: An AOP framework for synchronization. In *Int. Conf. on Tools and Alg. for Construction and Analysis of Sys.*, 2004.
- [6]C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Prog. Lang. Sys.*, 1983.
- [7]R. Khatchadourian. Poster on Modular reasoning about Aspect-oriented programs: A rely-guarantee approach. At *2nd European Summer School on Aspect-oriented Software Development*, DISI, University of Genoa, Italy, July 2007.
- [8]R. Khatchadourian. Talk on Modular reasoning about Aspect-oriented programs: A rely-guarantee approach. At *Ohio Graduate Student Symposium on Computer and Information Science & Engineering*, University of Cincinnati, OH, April 2007.
- [9]R. Khatchadourian, J. Sawin, and A. Rountev. Automated refactoring of legacy Java software to enumerated types. Technical Report OSU-CISRC-4/07-TR26, Ohio State University, 2007.
- [10]R. Khatchadourian, J. Sawin, and A. Rountev. Automated refactoring of legacy Java software to enumerated types. In *IEEE International Conference on Software Maintenance*, 2007.
- [11]R. Khatchadourian and N. Soundarajan. Rely-guarantee approach to reasoning about aspect-oriented programs. In *International Workshop on Software Engineering Properties of Languages*, 2007.
- [12]G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Eur. Conf. Object-Oriented Programming*, 2001.
- [13]G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Eur. Conf. Object-Oriented Programming*, 1997.
- [14]G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *International Conference on Software Engineering*, 2005.
- [15]S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2004.
- [16]R. Laddad. *AspectJ in action*. Manning, 2003.
- [17]M. Lippert and C. Lopes. A study on exception detection and handling using AOP. In *International Conference on Software Engineering*, 2002.
- [18]A. Rashid and R. Chitchyan. Persistence as an aspect. In *International Conference on Aspect-oriented Software Development*, 2003.
- [19]N. Soundarajan, R. Khatchadourian, and J. Dovland. Reasoning about the behavior of aspect-oriented programs. In *IASTED International Conference on Software Engineering and Applications*, 2007.
- [20]Q. Xu, W. de Roever, and J. He. Rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 1997.