Diagnosis of Harmful Aspects Using Regression Verification

Shmuel Katz
Computer Science
The Technion
Haifa Israel
katz@cs.technion.ac.il

ABSTRACT
Aspects are intended to add needed functionality to a system or to treat concerns of the system by augmenting or changing the existing code in a manner that cross-cuts the usual class or process hierarchy. However, sometimes aspects can invalidate some of the already existing desirable properties of the system. This paper shows how to automatically identify such situations. The importance of specifications of the underlying system is emphasized, and shown to clarify the degree of obliviousness appropriate for aspects. The use of regression testing is considered, and regression verification is recommended instead, with possible division into static analysis, deductive proofs, and aspect validation using model checking.

Static analysis of only the aspect code is effective when strongly typed and clearly parameterized aspect languages are used. Spectative aspects can then be identified, and imply absence of harm for all safety and liveness properties involving only the variables and fields of the original system. Deductive proofs can be extended to show inductive invariants are not harmed by an aspect, also by treating only the aspect code. Aspect validation to establish lack of harm is defined and suggested as an optimal approach when the entire augmented system with the aspect woven in must be considered.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features –, *control structures.*

General Terms
Languages, Verification.

Keywords
Aspects, desired specification properties, noninterference, preventing harm, regression verification, aspect validation.

1. INTRODUCTION
Like all modularity and language concepts, aspects are intended to improve the development of complex systems. On the code level, Aspect-Oriented Programming (AOP) languages provide notations to separately declare and repeatedly apply aspects that cross-cut the usual class structure of object-oriented systems. Using AOP has already been shown in numerous case studies to isolate the treatment of concerns that otherwise are scattered throughout the system, and tangled with code treating a variety of application issues. However, it is clear that sometimes such augmentations of systems can make properties that previously held for the system become untrue in the combination of the system with the aspect.

Such changes in the properties of the system could be a proper outcome of applying the aspect if the property is considered undesirable, such as that the system deadlocked in certain situations, or that messages were visible to any other observer in the computer. On the other hand, in general there is no way to linguistically prevent aspects from invalidating some properties that *are* desirable. This could occur either inadvertently, or maliciously. An example of the former could be when an aspect intended to treat overflow of variables, by mistake also causes the system to deadlock. An example of the latter could be when a system with private fields that guarantee some level of privacy is augmented by an aspect that provides public methods for reading the values of those very fields, in order to expose their contents, thereby violating the desired level of privacy.

In order to identify and treat such situations, the systems to which aspects are woven need to be augmented with *specifications.* These are descriptions of the desirable properties of the system. Note that they do not describe *all* properties of the system, only those seen as important and positive. Such properties should be maintained even if the system is augmented with aspects, or even if an aspect is combined with other aspects. What *can* change are the properties of the system not seen in the specification. The form of such specifications is described in Section 2.

Treatment of harmful aspects also requires a rethinking of the degree of *obliviousness* needed by an aspect-oriented notation. Obliviousness has traditionally [2][3] been seen as a desirable feature of Aspect-oriented notations. Although several definitions are possible, all imply that the underlying system does not have to prepare any hooks, or in any way depend on the intention to apply an aspect over it. The application of an aspect adds new features to a system, but the system without the aspect has its own specification and is correct relative to that specification, without needing any aspects.

Obliviousness is clearly important in dynamically evolving systems, where the aspects may not even have been thought of when the original system was created. It also is appropriate when a system can have many variants, some with one collection of aspects, and some with another, each configured for a user's particular needs. This is one of the potential uses of aspects to allow more flexible components, configurable on demand.

However, a total obliviousness to aspects prevents treating such malicious aspects as the one that reveals values intended to be kept private. Who prevents the application of such an aspect, on the language/system level (as opposed to locking the source in a safe, and physically preventing access to it)?

If specifications are available, a middle ground is possible, where a system is oblivious to the particular aspects to be applied to it, but still can restrict new aspects to those that do not violate its specification (or at least some parts of its specification). An aspect will be considered harmful if it invalidates any desired properties of the system to which it is applied. This will be more precisely defined and justified in Section 2.

The paths open to diagnosis of harmful aspects are usual testing, static code analysis similar to that done by type-checkers, and use of formal methods, both deductive verification and model checking. We shall consider all possibilities. The type of augmentation or change made by an aspect is another dimension that can determine the best way of preventing harm. The three basic divisions [6] are to *spectative* aspects that only gather information about the system to which they are woven, usually by adding fields and methods, but do not influence the possible underlying computations otherwise, *regulatory* aspects that change the flow of control (e.g., which methods are activated in which conditions) but do not change the computation done to existing fields, and *invasive* aspects that do change values of existing fields (but still should not invalidate desirable properties).

Yet another question is whether only the aspect module itself must be analyzed, independently of any system to which it may be woven, or whether an entire system augmented by an instance of the woven aspect is the object of analysis. In the continuation, the former is called *aspect code* analysis, and the latter *augmented system* analysis. The system before an aspect is woven into it is termed the *original* system.

The focus on preventing harmful effects of aspects is unusual, but as will be shown, does allow a uniform treatment. Such a treatment is more difficult when the new properties to be established by the aspect also need to be taken into consideration. Taking a medical analogy, the basic principle should first be, as in the doctor's Hippocratic oath: "Do no Harm."

2. SPECIFICATIONS OF ASPECTS AND SYSTEMS

A full treatment of aspects and their compositions clearly does deal with the specifications of the aspects themselves, and not just of the underlying system. In a HyperJ view, the entire system is composed of such aspects, or concerns. However, such specifications are often difficult to construct. Aspects on a code level are typically described by defining *joinpoints* where changes are to be made, and *advice*, with code to augment or replace what is done at the original joinpoint. Note that joinpoints may be defined as dynamically determinable events, and not merely locations in code or method calls.

As already defined in earlier works[6], specifications of aspects need to describe both what is assumed about any object or method in the basic system to which the aspect may be applied (and in general, what must be true at each joinpoint identified by the aspect), and, on the other hand, what is required to be true after the advice is applied, if the needed assumption indeed holds at the joinpoint. For each joinpoint and advice segment of code, the advice assumes some property of the system, and guarantees some property when it finishes. Such an assume-guarantee structure for aspects has already been recognized in [1], and [7], and is essential for describing the added value of an aspect. The overall properties added by the aspect can also be globally described. Since many aspects deal with so-called non-functional concerns like availability, fault-tolerance, security, or persistence, providing their specifications is that much more difficult.

Here, however, we concentrate on simply avoiding harm, and thus are not interested in what new properties are promised by the aspect. Only the specification of the system to which the aspect is woven is needed to prove the absence of harm. Since that specification usually deals with basic functional properties, it is more amenable to a description in standard temporal logic, and/or using precondition/postcondition pairs around methods or functions.

The obliviousness of systems to aspects is reflected in that usually the underlying system does not make assumptions of any kind about the possible aspects that may be applied. The existence of a specification of the desired properties that hold for the basic system provide a way to weaken obliviousness while maintaining the desired characteristics of extensibility and flexibility to add new unanticipated aspects. The specification of the basic system, in addition to restricting the implementation of the system, also can restrict future aspects, either by default —guaranteeing that all the desired properties in the basic specification will be maintained after weaving an aspect--- or in a more restricted version, where only some of the original desired properties are designated as unchangeable. Thus, for purposes of avoiding harm, the only requirement of the aspect is that the desired properties of the basic system expressed in its specification remain true when the aspect code is woven into the basic system and the augmented system is then executed.

Although not essential to the arguments in this paper, temporal logic provides a convenient formal notation for describing properties of execution sequences. In the simplest version G stands for 'globally' meaning from now on in the sequence of states, and F stands for 'in the future', meaning that eventually there is a state. Thus an assertion G(p => Fq) means that in every state, if p is true then eventually there will be another state with q. If p represents "a request has been made", while q is "a response is given", this corresponds to a specification that every request has a later response. Note that a counterexample to such an assertion would involve showing a computation with a state where p is true, but which never has a later state with q true. Whatever specification notation is used, it should not allow expressing assertions about immediately following states (using, for example, the "next-state" temporal modality X), since such assertions are known to be sensitive to any refinements or additions, and will be violated by any aspect that adds computation at problematic points. Thus we require a "stutter-free" version of temporal logic [5].

3. REGRESSION TESTING AND ITS LIMITATIONS

A straightforward approach to detecting harmful aspects would seemingly be the use of *regression testing*. The idea is simply to retest a system every time a new aspect is woven into it, to ensure that the test suite which previously was passed (and presumably captures the desirable outcomes that should be maintained) is still passed. Then the new properties to be added by the aspect could later be validated with new additional tests to be added to the test suite. This is the technique used by Extreme Programming (XP) [8] in place of having specifications, and is intended in XP to be applied to any significant change (e.g., a new version) in the system. However, there are several serious drawbacks to this approach when applied to aspects and their weaving:

First, regression testing is most easily applied to systems to which spectative aspects have been woven, where the aspects do not influence the computations of the underlying system at all. A regression test then could reasonably expect that the fields of the underlying system are unaffected by the augmentation of the aspect, so the results of the tests are unchanged. A violation is then trivially determined by comparing the results of the test, and can be inspected automatically. Yet when spectative aspects are used, it is more efficient to determine such situations using static analysis, as described in the next subsection. When the aspect is regulative or invasive, and thus *does* affect the computation, the results of the test will differ from the same test applied to the original system. They thus are often difficult to evaluate, and any violation cannot be determined automatically simply by detecting changes.

Second, this approach obviously relates to the entire augmented system, and retesting the entire system every time an aspect is applied is often unfeasible due to time or resource constraints. For a complex system, it seems overkill to activate the entire test suite even if an aspect with presumably small changes to only some of the objects and methods is added. Also, when aspects are taken from a library and bound to new systems, such a small investment in coding (binding the aspect to a system) hardly justifies an entire activation of the test suite. Moreover, if aspects are applied and removed dynamically, during run time, retesting is not realistic.

Third, and most significantly, the original tests obviously did not take into account the structure of the aspect or the influence it may have on the basic computation paths. Thus, for example when conditionals appear in aspect code, the original tests may be completely irrelevant, since new paths are generated and followed, and the tests may miss many computation paths. Precisely because of their cross-cutting nature, it is difficult to isolate parts of the test suite that still might be relevant, since many regular modules (e.g., classes) are affected by each aspect.

Therefore, simply using regression testing does not adequately treat harmful aspects. We thus turn to regression verification, which can be based on static type analysis, deductive verification, or model checking using aspect validation.

4. STATIC ANALYSIS

As noted above, when the aspect to be applied is spectative relative to the underlying system, it should often be possible to establish this fact using static analysis of code. As will be discussed, in some languages the aspect can be analyzed in isolation, while in others the augmented system must be considered. A spectative aspect does not change either the value of any field or the flow of method calls of the underlying system. New fields, methods, and even classes can be added, but the new model of computation has a very particular relation to the underlying one without the aspect. Each computation path has sections of original computation interleaved with sections of new computation. The result is always equivalent to temporarily suspending the underlying system, recording some information about it, computing new values not influencing the underlying system in any way, and then continuing as before.

Such a situation might be difficult to detect directly on the execution graph of the computation, but it is amenable to detection on the code level in some aspect languages, using standard type checking and data-flow techniques. The idea is that the locally defined fields of the aspect are the only ones computed by that aspect, and no assignments are made by aspect code to fields or to parameters that can be bound to fields, variables, or parameters of the basic system. The aspect code also cannot "redirect" the flow of execution, and simply adds to the previous system without skipping any of its computation.

This situation is amenable to syntactic detection by analyzing only the aspect if all bindings between fields or variables of the aspect and the basic system are made through parameters of the aspect. On the other hand, when arbitrary binding is possible, for example by using the same name in both code segments, then only when a specific binding has been made can the augmented system be analyzed to determine which elements are bound, and whether the aspect is spectative. In either case, dataflow techniques, such as the *uses* and the *defined-use* pairs of standard code optimization, can be employed to determine whether there is any influence of fields in an aspect on those of the basic system (the other direction is, of course, not a problem). The possibility of analyzing just the aspect is one argument in favor of clearly identifying parameters for weaving, rather than allowing free bindings that force analysis of the entire augmented system.

Showing that an aspect is spectative is one way to guarantee that all safety and liveness properties involving assertions only about variables, fields, and methods of the underlying system will not be influenced by the aspect (as already explained, without assertions about "next" states). However, it should be noted that properties such as "the value of a field is not visible outside the class" can be violated by spectative aspects, even when they were previously true. The problem is that the assertion of "not visible" involves both the original fields and methods *and* new fields or methods added by the aspect. As already noted in the Introduction, a (hidden) field X could be "made visible" by examining another field Y (added by the aspect) linked to X by an invariant, or by adding new public methods.

Such data-flow and type-safety techniques are always conservative, in that if successful, the spectative nature of the aspect is guaranteed, and the aspect can cause no harm for specification properties as described above. If the analysis does not establish that the aspect is spectative, it remains to be seen whether the aspect is actually harmful.

5. DEDUCTIVE PROOFS OF CLASSES OF TEMPORAL PROPERTIES

It is also possible to establish a lack of harm for either specific properties or entire classes of properties using deductive proofs only over the aspect code. For example, an invariant of the original system can often be shown to also be an invariant of the augmented system, even without analyzing in what situations the aspect code will be applied. This is true when the invariant I is what is known as "inductive," meaning that {I} s {I} can be shown for each individual step s. Note that it is sufficient to show that if the invariant is assumed before a step, it will again be true at the end of the step. In this situation, to establish that I is also an invariant of the augmented system, it is sufficient to check that each aspect action t also satisfies the same assertion {I} t {I}. Since I is already known to be an invariant of the original system, it actually *is* true of the augmented system whenever the aspect is first applied, even without analyzing the joinpoints. By induction, it is easy to see that I will hold whenever some t action is taken, so will be an invariant of the augmented system, even without rechecking the original code.

For example, consider a situation where x>y>0 is an invariant of a system, and an aspect has changes of the form

<complex> ➔ double (x,y),

where <complex> is a complex condition for applicability, and double(x,y) doubles the values of x and of y. Then we easily have {x>y>0} double(x,y) {x>y>0}, extending the invariant to the augmented system, even though only the aspect code was newly analyzed, and when it is applied was ignored.

It is also possible to prove that an aspect is "almost spectative" in that it might only abort an underlying system, but would not otherwise affect the computation of the original statespace. In such a situation, liveness properties of the underlying system might be harmed, but all safety properties are maintained.

Consider an aspect that treats overflow for variables in one part of a system with limited memory. An invariant of the underlying system that is also in its specification could be that x = y. However, this will no longer hold in the augmented system if x is treated for overflow, resulting in new assignments to x, while y is not. In this case the aspect has harmed the system by violating a safety assertion of its specification. On the other hand, if the aspect stops the system when overflow is detected, rather than continuing as above, then safety properties *are* maintained, as long as the system continues.

6. REGRESSION ASPECT VALIDATION FOR INVASIVE ASPECTS

The approach of aspect validation, first suggested in [4], can be specialized to detecting harmful aspects. The idea of validation is to prove that each individual weaving is acceptable, rather than having a single generic proof that the aspect always does no harm. It is effective when each weaving of an aspect triggers automatic generation of verification tasks that themselves are automatically checked, e.g., using a model checking tool. Note that the initial organization and set-up of the validation framework for each application aspect can be non-algorithmic and require human effort and invention. However, the validation associated with each weaving of the aspect does not require such intervention, and must be automatic.

Aspect validation is appropriate when we cannot successfully identify absence of harm syntactically or statically by analyzing the aspect code, and yet concluding about lack of harm for classes of properties and for every possible weaving of the aspect. Thus we are forced to turn to techniques that analyze the augmented system, rather than just the aspect code. Indeed, in general we need to know the binding of the aspect to the basic system, and the properties which are the desired ones of that system, before the absence of harm can be established. Then we need to verify that those properties hold of the augmented system. The automatic verification for each weaving is essential to make this approach feasible.

In many cases a software model checker can be used to generate a model checking task to be executed using a well-known tool such as SMV, Spin, or Java Pathfinder. One practical tool design and implementation for aspect validation was suggested in [4], where Bandera is used to generate input for standard model checkers directly from heavily annotated Java code. The annotations that express the specification of the original system are themselves given as aspects. These so-called *specification aspects* include parametric temporal properties, labels, predicates, and functions intended to annotate a system with its desired properties, as preparation of input for Bandera. The parameterization, and the fact that the annotation is kept as a separate module rather than being built into the original system allows the specification aspect to be applied both to the original system, and to one augmented with application aspects. Since annotating a system in preparation for a Bandera verification is a nontrivial task, using specialized notation and requiring human ingenuity, the reuse of the specification aspect is the key to making the approach practical.

Such an approach of aspect validation is possible when the original and the version augmented with aspects are ultimately given in the same notation. In practice, it has been used with AspectJ in the mode that generates source Java code for the system with its aspects, and could also be done when a Java bytecode verifier is available.

In essence, this is an incremental model checking task, if we assume that the properties in the specification of the orginal system were already verified using model checking. The task to be shown in order to verify that the new aspect causes no harm is simply to reprove the specification of the original system, but this time for the augmented system combining the original one with the aspect code bound to various joint points (including dynamic ones). We can and should reuse elements from the model checking of the original system in the model checking of the augmented one.

In particular, any model checking of the original system usually requires abstraction of the statespace to create a smaller model, in order to avoid the state-explosion problem that often prevents a successful verification, even though the model checking itself is algorithmic. Like the specification aspect, these abstractions, used to make the proof feasible in the available space and time, can be reused for the augmented version. If this should prove insufficient because an extensive new statespace is generated, of course new abstractions might be necessary. However, this would violate our goal of fully automatic validation. In case studies we have carried out, the abstractions needed for the original still lead to sufficient reductions in the augmented system, but further research is needed to determine whether this is generally the case.

As opposed to simple regression testing, this is a full verification, and thus will check the desired properties for whatever new paths might be introduced by the weaving of the aspect. However, using aspect validation for regression verification, and thus model-checking the entire augmented system, is clearly less desirable than proving once and for all (by only analyzing the aspect code) that an aspect cannot harm large classes of easily identifiable systems and specification properties.

7. SUMMARY

The goal of full specification and verification of aspect-oriented systems is still important. But even when specifications of aspects are difficult to express for non-functional concerns, and a full verification may be difficult, showing the absence of harm through regression verification is a valuable first step. A significant improvement in code reliability and quality can be obtained at a relatively low cost, especially when a specification of the underlying system is already available. A combined approach of static dataflow analysis, one-time deductive proofs, and aspect validation shows particular promise. Proper language design for aspects, with local variables and parameterization, can help extend the static analysis of only the aspect code to determine harmfulness or its absence, either for classes of properties and for every possible weaving, or reanalyzing only the aspect for each weaving. When analysis of the full augmented system is required, aspect validation is suggested.

This focus on harmful aspects also allows a weakening of obliviousness in a way that maintains extensibility, but does not allow (or at least diagnoses) malicious or inadvertent corruption of the desired properties of the underlying system.

ACKNOWLEDGMENTS

This work was done while the author was visiting at Lancaster University in the PROBE project, and is supported by UK Engineering and Physical Sciences Research Council (EPSRC) Grant GR/S70159/01. Valuable discussions with Awais Rashid are gratefully acknowledged.

8. REFERENCES

[1] Devereux, B., Compositional Reasoning about Aspects using Alternating-time Logic, FOAL 2003.

[2] Filman, R.E., and Friedman, D.P., Aspect-Oriented Programming is Quantification and Obliviousness, Workshop on Advanced separation of Concerns, OOPSLA 2000, October 2000.

[3] Filman, R.E., What is AOP, Revisited, Workshop on Advanced Separation of Concerns, 15th ECOOP, June, 2001.

[4] Katz, S., and Sihman, M., Aspect Validation Using Model Checking , Intl. Symposium on Verification in honor of Zohar Manna, Springer-Verlag, LNCS 2772, pp. 389-411. Also, early version in FOAL2003.

[5] Lamport, L., What Good is Temporal Logic?, IFIP 9th World Congress, 1983, pp. 657-668.

[6] Sihman, M. and Katz, S. Superimposition and Aspect-Oriented Programming, The Computer Journal, 46 (5), 2003, pp. 529-541.

[7] Sipma, H. B., A Formal Model for Cross-cutting Modular Transition Systems, FOAL 2003.

[8] eXtreme Programming homepage, http://www.extremeprogramming.org