Aspect Categories and Classes of Temporal Properties

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

ABSTRACT
Generic categories of aspects are described, and their potential value is explained. For some categories, broad classes of syntactically identifiable temporal properties, such as safety, liveness, or existence properties, are guaranteed to hold for a system with any aspect of the category woven into it, if the property was true in the system without the aspect. Thus classes of properties preserved by the aspect are defined. Moreover, relatively simple verification techniques are shown to hold for some classes of temporal properties over systems augmented with some other categories of aspects. Verification of new properties added by the aspects is also considered.
Each category is defined in terms of the semantic transformation it makes to the state graphs of underlying systems. A generic procedure to identify syntactically when an aspect belongs to a category is described and related to existing code analysis systems that use static code analysis and dataflow techniques. The definitions of categories, identification procedures, and lemmas about property classes provide the needed foundations that justify and motivate automatic code analysis modules to identify aspect categories. The categories enable simpler proofs of correctness than would otherwise be possible, and exploiting their characteristics can aid in software development.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features – control structures.
General Terms
Languages, Verification.
Keywords
Aspect specification, spectative, regulative, invasive, aspect categories, dataflow analysis, aspect verification.

1 INTRODUCTION

Aspect-oriented programming seeks to isolate cross-cutting concerns in aspect modules that are then woven with other aspects or with more standard object-oriented classes. Already in the earliest examples of aspects, it was clear that some are more complicated than others as far as the relation to the underlying program with which they are woven. The classic examples of logging or adding performance monitoring influence the underlying program less than aspects treating overflow of integer computations, or restricting access to some methods to favored users. In turn, those are less closely tied to the computations of the underlying program than aspects that compute and apply discount prices to an underlying on-line bookshop or that change the designs of buildings in an architecture support program in order to enforce handicapped access regulations.

In [27] and in some earlier works on superimpositions, spectative, regulative, and invasive types of aspects were suggested to describe categories of aspects such as those above. The *spectative* aspects only gather information about the system to which they are woven, usually by adding fields and methods, but do not otherwise influence the possible underlying computations. On the other hand, *regulative* aspects change the flow of control (e.g., which methods are activated in which conditions) but do not change the computation done to existing fields, while *invasive* aspects do change values of existing fields (but still should not invalidate desirable properties).

In this paper these categories of aspects are extended and made more precise. An analysis is given of how properties are affected by aspects in each category. The classes of properties are described as those satisfying generic temporal logic formulae, and have previously been used for purposes related to refinement and proof techniques. Both the influence of the aspect on classes of properties that already were true in the underlying system, and the difficulty of verifying new properties added by the aspect are considered. The categories and lemmas enable establishing some properties automatically without expensive proofs using model checkers. For some other properties and categories the verification is not immediate, but simpler proofs of correctness can be used than would otherwise be possible. Aspect categories can aid in software development by encouraging the modularity that is identified in the categories presented here.

The view taken in this paper is of a main existing system to which one or more aspects can be applied, as in AspectJ [18]. When the entire system is considered to be a composition of aspects, as in the HyperJ approach [24], similar considerations can be applied. The implications of our results for interference among aspects is considered in Section 8, but otherwise the relations between a given system and a single aspect are treated. In the continuation, the system before an aspect is woven into it is termed the *original* or the *underlying* system. After an aspect has been woven into it, the result is referred to as the *augmented* system. The aspect categories are defined in terms of restrictions on the semantic changes between the state graphs of the underlying and the augmented system. These are then used to justify claims about classes of temporal logic properties. A particular aspect generally cannot be directly checked for the semantic restrictions. Instead, for most aspect languages, static analysis with dataflow techniques or a rich type system can be used to identify subsets of the categories. Here, generic dataflow techniques for identification are presented for each category, and shown to guarantee the semantic definitions. In Section 9, related approaches and analysis techniques are surveyed, and their connections to the categories defined here are explained. As new techniques are developed, they can be shown to identify the appropriate semantic categories, so that the associated lemmas on classes of properties will automatically hold. Thus this paper can be seen as providing a semantic foundation and implications for temporal logic properties over static analysis for aspects.

```
public class Rational{
        private int       numerator      = 0;
        private int       denominator   = 1;
        public int getNumerator() {...}
        public void setNumerator(int numerator) {...}
        public int getDenominator() {...}
        public void setDenominator(int denominator) {...}
        public Rational add(Rational r) {...}
}
public class RationalExam{…
        private static String getInput() {...}
        private static Rational randomRational() {...}
        private Rational getAnswerToAddQuest(Rational r1, Rational r2) {...}
        private boolean checkAnswer(Rational answer, Rational correctresult) {...}
        public void doExam() {...}
}
```
Figure 1. Outline of an online exam system

In this paper a running example is used of aspects over an underlying system that manipulates fractions, and generates and checks online exams for students with simple arithmetic exercises for fractions. In Figure 1, the outline of this system is given, as a minimal basis on which to add aspects. The RationalExam class initiates exams, generates questions, accepts answers, and checks whether an answer is correct, all using the Rational class. The doExam method is activated externally. In later sections aspects are added to compile the results, restrict use of the system, and reduce the fractions.

In order to investigate the properties of a system and categorize aspects applied to it, the systems to which aspects are woven as well as the aspects themselves will be assumed to have *specifications*. These are descriptions of the desirable properties of the system. Often the exact nature of the specifications will not need to be available, but rather the class of property described will be sufficient, e.g., safety or invariant properties. Note that specifications do not describe all properties of the system, only those seen as important and positive. Such properties should be maintained (perhaps in a modified form) even if the system is augmented with aspects, or even if an aspect is combined with other aspects. For example, if a system has been shown to properly complete each request submitted externally, and an aspect is added to monitor performance, we do not want the combined system with the aspect to occasionally crash in midtreatment of a request. In the system with fractions, a desirable invariant property might be that the denominator of every fraction is positive, and any aspects added might be expected to maintain that invariant. What can change completely are the properties of the system *not* seen in the specification. The form of such specifications using syntactic forms of temporal logic assertions is described in Section 2.

In general, once an aspect has been shown to belong to one of the categories described, there are various semantic implications for the properties of a system augmented with such an aspect. In the best case, classes of properties true of the underlying system can be shown to be maintained automatically in the augmented one, without further proof. In other situations, properties can be established by analyzing only the aspect code. In the worst case, the entire system must be considered, but for certain properties easier proof methods can be used. The lemmas about aspect categories and types of temporal properties are justified using a semantic view of object systems and aspects based on state graphs. In Section 3, this view is presented and related to other definitions of the semantics of aspects. Section 4 explains that realtime and next-state properties are not covered by the analysis framework in this paper. In Sections 5-7, spectative, regulative, weakly invasive, and invasive categories of aspects are defined in terms of the semantic transformations of the graphs, syntactic checks are shown to guarantee aspects of the desired category, and classes of temporal properties are shown to be preserved or to be easier to prove for the category. In Section 8 interactions among multiple aspects are considered, Section 9 discusses related work, while Section 10 summarizes the results.

2 TEMPORAL LOGICS AND CLASSES OF PROPERTIES

2.1 Specifications of aspects

Specifications of aspects need to describe both what is assumed true of the underlying system at each joinpoint identified by the aspect (often, any object or method in the basic system to which the aspect may be applied), 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 (as well as possibly some properties during the aspect execution). Such an assume-guarantee structure for aspects has already been recognized in [9], [27], and [28], 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 such as availability, fault-tolerance, security, or persistence, explicitly providing their specifications is that much more difficult. Still, no such distinction is made here, and temporal logic can be used to specify such properties as well as functional ones. Even if some properties have not been formally expressed, for the purposes of property analysis described below, it is sufficient to identify to which well-known classes of specification properties the specific properties of interest belong, e.g., if they are invariant properties.

2.2 The semantics of temporal logic

Temporal logic provides a formal notation for describing properties of execution sequences, using temporal modalities to quantify over the execution sequences and the states in them from a reference state. In the simplest version of linear temporal logic (without existence properties), G stands for ‘globally’, i.e., from now on in the sequence of states, and F stands for ‘in the future’, i.e., eventually there is a state. Thus an assertion $G(p \Rightarrow Fq)$ 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. An assertion pUq means that p will hold in all states from now on until a state that satisfies q (and

there is a state satisfying q). An assertion Xp (in words: `next' p) is true in a state if p itself is true in the state immediately following.
Formally, in terms of an execution sequence (a sequence of states) σ, and an index in that sequence, i, we have
(σ, i) satisfies Gp iff $\forall j \geq i.\ p(\sigma(j))$
(σ, i) satisfies Fp iff $\exists j \geq i.\ p(\sigma(j))$
(σ, i) satisfies pUq iff $\exists j \geq i.\ q(\sigma(j)) \wedge \forall k.\ j > k \geq i.\ p(\sigma(k))$
(σ, i) satisfies Xp iff $p(\sigma(i+1))$
Similar past modalities are defined symmetrically. Linear temporal logic only has modalities defined in this format.

In branching temporal logic, such as CTL [4], there are also *path* quantifiers over all possible continuations from a state (denoted A) and some possible continuation (denoted E). Formally, such formulas are interpreted over an execution tree, i.e., a collection of execution sequences organized into a tree where common prefixes are written once as paths from the root.
When applied to systems, temporal logic formulae are interpreted over a Kripke structure semantics (see [5]) consisting of a state transition graph with nodes corresponding to the possible states reachable in executions of the system, and edges labeled by possible atomic actions of the system that transform the source state into the target. The states are labeled by values for every possible atomic assertion.

In Linear Temporal Logic (LTL), an assertion is true for a system iff it is true for every path through the Kripke structure starting at initial states, while for CTL, the formula must be true for the execution tree obtained by 'unwinding' the Kripke structure. Thus an LTL assertion Gp is equivalent to a branching temporal logic assertion AGp.

2.3 Classes of temporal properties

Classes of temporal properties were first defined by [22] and shown to correspond to a simple syntactic form, to a proof method, and to a complexity hierarchy.
Safety properties hold in every state, and may relate to the history of states up to the state being considered. Such properties describe what is allowed in the system, under what conditions, and what states cannot occur. This class includes precedence properties such as "a state satisfying P is always preceded by a state satisfying Q", as well as the invariant properties described below. As shown in [22], all safety properties can be expressed as a linear temporal logic assertion of the form Gp, where p is a predicate without other future modalities, relating only to the state variables or the past history leading to the state under consideration (a past predicate). The temporal modalities needed to express such past assertions are not given here since they are not needed in the continuation.

(**Global**) **Invariant** properties are true in every state (without reference to the computation history), and are the most common subclass of assertion in the class of safety properties. A weaker safety assertion common in object-oriented systems is known as a **class invariant**, and is required to be true initially, as well as before and after each method call of the class (but not while method calls are ongoing). A **partial correctness** assertion, intended to hold whenever the system terminates, is also a kind of invariant property. An invariant intended to be true at certain points, such as a class invariant or partial correctness assertion, can be transformed to a global invariant of the form "at(method-call) implies p" or "at(return) implies p" instead of only asserting p at the method-call or return point. Thus partial correctness and class invariants are merely special forms of global invariants.

Liveness properties are guaranteed to hold eventually for every possible execution. Example properties in this category are the successful termination of a kind of method activation, an assertion that every message sent is eventually received, or an assertion that some crucial event (e.g., a particular method call) will occur whenever some other event occurs first. Such properties always can be expressed using an eventuality (F or Until) modality not negated. Among the common combinations are Fp (to express "eventually there is a state in which p is true"), GFp (to express that p is infinitely often true in an infinite computation), FGp (to express that eventually, p becomes continuously true), or various combinations of these forms.

Both safety and liveness are categories that make assertions about every possible computation of a system. In a branching temporal logic, there is an A quantifier on the outer level, and no other path quantifiers. As noted above, there are also branching temporal logic **Existence** properties that can express assertions about possible computations of a system (e.g., there is a path that reaches an interrupt). Syntactically, these are equivalent to properties with an E path quantifier not negated, relating to the existence of a computation among the possible executions of the system. There are branching time generalizations of the safety and liveness properties above that have been shown to be maintained under certain types of model abstractions [7].
Instead of writing "for every object r of a class, r.fieldname" in assertions, just the fieldname is used when clear from the context. So G(denominator >0) means that the assertion is true in every object of class Rational.

3 THE SEMANTICS OF ASPECTS AND OBJECT SYSTEMS

3.1 Approaches to aspect semantics

In order to define and reason about the categories of aspects and their connections to classes of properties, the form of an object-oriented system, and its semantics must be defined. There are several formal semantic definitions of aspects and aspect systems, using, e.g., denotational [32], operational (or so-called *small step* semantics) [15], process calculus [3], and functional [31] approaches. In principle, any of them could be used to justify the claims connecting categories of aspects to the correctness of types of temporal properties. Most of them assume a simplified object or functional base language, in order to concentrate on a semantic definition for the new aspect construct.

3.2 State machine semantics for object systems

Here we adopt the state graph semantic view, because it is most appropriate for verification of temporal properties, and is used by software model checkers such as Bandera [14]. This view can be seen in the UML statechart semantics, where each class is accompanied by a statechart (equivalent to a hierarchical state graph) expressing the possible states and transitions for each instance (object) of that class.

A node in the state graph of an object gives specific values to the variables, fields, and control state of that object, and the edges (transitions) describe the effect of executing an enabled step from that node. A system is then described by the cross product of the state graphs of the active objects, linked by potential method calls among them. Each object has a designated node in its state graph corresponding to the present values of the object. The cross product of these designated nodes defines a mapping that provides the present values of variables, fields, locations, and internal stacks of method activations along with identification of which actions are presently enabled in the system, for each object. This can be seen as a continuation semantics describing both the immediate values and the potential continuations for each point during an execution. For simplicity, the term *variables* is used to refer to all of the state components that are given values in each state.

The semantic meaning of a system (either underlying or augmented with aspects) at any point during its execution is defined as the expansion of the cross product of state graphs described above, to a single computation state graph, where the nodes are particular values for the existing variables at that point (what is usually called a state), and the arrows are transitions that correspond to the atomic actions of the system. Note that this does not exclude transitions that extend the state graph by adding new object occurrences, or shrink it by discarding objects that have been finished. Issues of inheritance and polymorphism can complicate the definition of the state graph, but are orthogonal to the question of adding aspects. The system and semantic state graph are often organized as a reactive system [22] where each external activation leads to a finite computation corresponding to a transaction. A maximal sequence of states in the graph from an external activation to a rest state is known as a *trace*. Although such a state graph may have an infinite number of states (if there are variables with infinite domains, such as the integers), abstraction techniques can be used to create finite-state versions that are used in model-checking.

3.3 Aspect semantics as state graph transformations

An aspect declaration and its binding (known as *weaving*) to an underlying system define *joinpoints* or events of the underlying system where the aspect is to be activated and aspect code (known as *advice*) is to be executed. The joinpoints are described syntactically using *pointcut* declarations. Semantically, the weaving of an aspect to a system transforms the graph of the original system to that of the augmented one. Thus the semantics of an aspect is defined as a transformer of state graphs: given the graph of the underlying system, it yields the graph of the augmented one. Note that this semantics is neutral about how aspects and weaving are actually implemented in an aspect language: systems that in-line aspect code, those that capture events and transfer control at run time, as well as languages with other known implementation techniques, are all equally valid.

When an aspect is woven, for each join point in the original system, a transition to the beginning of the state machine of the corresponding advice is added. The transitions and states generated by applying the aspect advice code are part of the subgraph following the joinpoint transition. However, note that objects of the original system not related to the joinpoint continue to have enabled operations which can be interleaved with those of the aspect code, and that appear in the state graph of the augmented system as transitions from states generated by aspect operations (or as separate components that have an implicit cross product of local states).

The semantic picture is clearly influenced by whether the aspect code is intended to interrupt the underlying system *before*, *after*, or *around* an event of that system. (Note that the terminology of AspectJ is used here, although other aspect languages have similar ideas.) If the aspect advice code is intended to execute *before* the event that defines the occurrence of a joinpoint, then the transition to the advice code is from the state before the transition corresponding to the joinpoint event, and the joinpoint event transition itself is removed at that point, to be inserted later where the aspect advice completes. The intention is that the code of the underlying program will be continued after the advice completes, but it is not at all clear that the continuation is from the local state that previously held before that continuation. Thus the issues raised by where to reconnect are discussed as the types of aspects are examined in greater detail. The considerations for an aspect intended to occur *after* an event defined by a pointcut are similar, with the transition to the beginning of the advice state graph added from the state after the event in the underlying system. When the *around* option is used and the *proceed* statement occurs in the advice, it is equivalent to a *before* advice, followed by an *after* advice. When *proceed* does not occur, the advice does not continue from the point in the code at which it was interrupted, so there is no return arrow at all.
A fragment of a generic state graph of an underlying system is seen in Figure 2(a), where *x* represents the state of the original system, while the augmented graph after weaving an aspect before the *P* transition is seen in 2(b). As will be shown later, assuming that *w* is

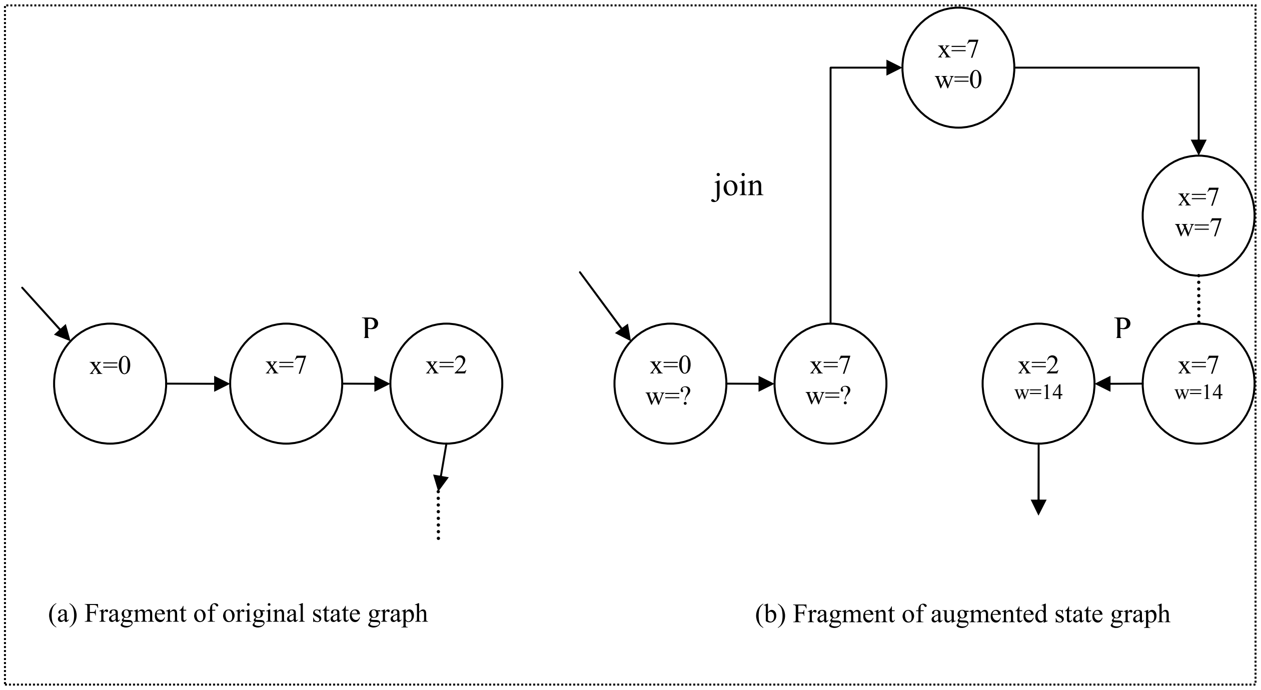
x=0
x=7
P
x=2
(a) Fragment of original state graph
join
x=0 w=?
x=7 w=?
x=7 w=0
x=7 w=7
P
x=7 w=14
x=2 w=14
(b) Fragment of augmented state graph

Figure 2. A fragment of underlying and augmented state graphs

local to the aspect, the situation described is typical of a spectative aspect because the *x* component is not changed by the aspect transitions, and the P transition at the end of the aspect transitions reconnects to the same state as in the original, if we ignore *w*. Note that in this example, there are no independent objects that would require a cross-product in the augmented state graph. In the augmented fragment, *w* appears in the states corresponding to those of the underlying system, but no assertion of the underlying system could have referred to it, since it was not defined there.
The influence of applying aspects from each category to an underlying system can now be considered in terms of the state graphs of the original and augmented systems. However, first the limits of this approach are described.

4 PROPERTIES NOT INHERENTLY PRESERVED

There are two types of properties which are not analyzed in the continuation: those not in the temporal logic defined earlier over the statespace of the underlying or the augmented system, and next-state properties. Both of these have properties that are not maintained, no matter which category of aspect is considered, and a more specialized analysis is needed which is different from the type of semantic reasoning used here.

4.1 Realtime properties

If an aspect or underlying system already explicitly contains clock variables, explicitly incremented in code, the framework here can relate to them. However, usually this is not the case, and realtime properties that relate to the elapsed time expected between two events are not expressible in the classes of temporal logic defined in Section 2, without special external variables. Time can be introduced either using special time or clock variables that automatically are increased with time values not explicitly in the code, or by extending temporal logic with additional operators. Thus the claims below that automatically extend properties of the original system to the augmented one do not apply to realtime properties. This is fortunate, since such properties can be influenced by aspects that have joinpoints between those events, and thus add the computation of the advice code to the previous computation of the underlying system. Clearly, some realtime properties true of the underlying system may not be true of the augmented one, no matter how loosely connected the aspect functionality is to that of the original system.

4.2 Next-state properties

Moreover, for any category of aspect, assertions about immediately following states that were true in the underlying system may not be true for the augmented one if an aspect can add advice (that generates new intermediate states). An assertion true in the underlying system such as $AG(p \Rightarrow Xq)$ (meaning, whenever p is true, q is true in the immediately following state), will not necessarily remain true if in the augmented system an aspect joinpoint can be added immediately after a state where p is true.

In general, assertions using the "next-state" temporal modality X are known to be sensitive to refinements or additions, and have therefore been considered problematic [21]. In fact, in order to achieve greater abstraction and robustness under refinements, it

has been suggested that a temporal logic should not be able to "detect stuttering" (i.e., repetitions of states). Lamport and others have shown that a temporal logic without X or the corresponding "immediately previous" operator cannot distinguish among sequences of states that only differ in the number of repetitions of states that appear. We thus have the following simple lemma:
Lemma 1: any property of the underlying system relating a state and its immediate successor or predecessor state (a "next-state property") is not automatically preserved in an augmented system when any aspect code can be applied at a joinpoint including the earlier state of the pair related.
This claim is straightforward because new intermediate states are always generated by the advice code.

5 SPECTATIVE ASPECTS

A spectative aspect can change the values of variables local to the aspect, but does not change either the value of any variable or the flow of method calls of the underlying system. New fields and methods can be added to existing classes of the underlying system, or new classes can be added, but these will not affect either the potential or actual actions of the original system. Each computation path of the augmented system has sections of original computation interleaved with sections of new aspect 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.

More precisely we have:
Definition: An aspect is **spectative** if the projection of the augmented state graph onto the state variables of the underlying system is identical to the underlying state graph, except that the projection contains additional repetitions of states connected by edges that correspond to aspect operations.

The repetitions of states in the projection represent the advice segments, which can affect variables local to the aspect, but not those of the underlying system. In Figure 2(b), the projection of the augmented system fragment onto *x* is identical to the original fragment in 2(a), except for repetitions of the *x*=7 state. Note that the projection of the augmented state graph could have several subpaths with repeated states branching from the state before a joinpoint state and reconnecting at the joinpoint state. This could be due to reading input values or branching in the aspect code. However, the set of traces (i.e., maximal sequences of states) of the projection is identical to the set in the original system except for repetitions of states, and the branching structure of the original is also maintained.

Such a situation might be difficult to detect directly on the state graphs that represent the semantics, and which, of course, are usually not generated in practice. However, detecting that an aspect is spectative is possible on the code level in most aspect languages, using standard type checking and data-flow techniques. For spectative aspects the local 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. Care must be taken in defining what is "local." A class and objects or fields declared within the aspect and only accessed there are clearly local. In addition, even if a class is declared globally, but objects are instantiated and the class methods are used only within the aspect, that class is *effectively local* to the aspect. Moreover, parameters of system methods that print values are also considered local to the aspect, because printing a value has no effect on other values printed. This means that assertions about what is *not* printed may be changed in the augmented system (although positive assertions about values printed in the original program are maintained, as will be shown). The aspect code cannot "redirect" the flow of execution, and simply adds to the previous system without skipping any of its computation. Moreover, the aspect code must be "wait-free", i.e., progress of its execution is not dependent on a condition being achieved in the underlying system.

This situation is easiest to detect 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 global analysis. The generic *detection procedure* to identify a spectative aspect is:

1. Identify which variables (including parameter names) of the aspect code are bound to variables in the underlying system (either by identifying actual-formal parameter pairs from an aspect instance using its declaration, or by identifying the same name in the aspect code and the underlying code, if that is sufficient to bind the variables). The set C denotes the aspect variables bound to an underlying system variable.
2. Check that no variable of C is assigned a value by the aspect (appears on the left-hand side of an assignment in advice of the aspect, or is an actual parameter that is assigned a value in an internal method call of the advice code).
3. Check that each aspect code segment (advice) terminates. Although generally an undecidable problem, syntactic special cases often hold, such as identifying straightline code (a basic block, in dataflow terms), or loop-free segments.
4. Check that the aspect code does not disable independent underlying operations and is wait-free.
5. Check that the code of the underlying system is resumed at each join-point identified and the enabling conditions for the underlying operations have not been affected. Thus there cannot be exceptions thrown in the aspect code that lead to abnormal termination or do not resume the underlying execution at the joinpoint from which it was interrupted, and *around* advice without a *proceed* is not allowed.

First we show that if the generic syntactic checks are made successfully, then the aspect is spectative. An aspect can be spectative if bound with *before*, *after*, or *around* including a *proceed*. Here we treat only the *before* case, since the others are similar. Recall that here, as in all subsequent lemmas until Section 8, an augmented system is treated where a single aspect is woven to an underlying system.

Lemma 2: If the detection procedure above has been applied successfully to an aspect and underlying code, the aspect is spectative.
Proof: From steps 1 and 2 it follows that the advice does not change the variables of the underlying system. Thus in the projection of the augmented system to the variables of the underlying one, each transition corresponding to operations of the advice code leaves the values of those variables unchanged. From step 3 it follows that the subsequences corresponding to aspect operations are finite. From step 4 it follows that the enabledness of transitions from objects independent of the joinpoint in the underlying system is not affected by the aspect operations. Finally, from step 5 it follows that after the last state resulting from an advice code operation, the location in the code of the program counter is at the operation that identified the joinpoint. Since the values of the variables of the underlying program are also unchanged, that operation will have the same effect as in the underlying system, so the reconnection arrow is to the same state as previously (see Figure 2). It thus follows that the projection of the augmented graph satisfies the conditions in the definition of a spectative aspect. □

```
public aspect ScoringAspect {
        // Inter-type declarations
        private int      RationalExam.correct = 0;
        private int      RationalExam.wrong  = 0;
        // Pointcuts
        pointcut checkingAnswer(RationalExam exam) :
                call(void RationalExam.checkAnswer(Rational, Rational))
                && target(exam) ;
        pointcut doingExam(RationalExam exam) :
                call (void RationalExam.doExam()) && target(exam);

        // Advice
        after(RationalExam exam) returning (boolean ok):
                checkingAnswer(exam) {
                if (ok)
                        exam.correct++;
                else
                        exam.wrong++;
        }
        after(RationalExam exam) : doingExam(exam) {
        System.out.println("You answered " + exam.correct +" correct answers");
        System.out.println("You answered " + exam.wrong +" wrong answers");
        }
}
```

Figure 3. Aspect for computing scores

Consider the aspect given in Figure 3. This aspect counts the correct and incorrect answers given to questions about fractions in locally declared fields *correct* and *wrong*. This aspect satisfies all of the conditions above, since it uses the parameters bound to the Exam object, but only changes the local fields, or prints values, and the advice code does not wait for any condition to hold. This aspect is therefore spectative. Note that even though the termination requirement in item 3 is undecidable in general, in this example there are no loops, and only simple system methods, so the termination is trivial.

As another example, an aspect that treats the display of a shape manipulation program is often used as a case study for modularization using aspects, as opposed to a version without aspects that scatters the display updates in the object code. The aspect simply gathers information on the shapes, including join points that occur as shapes are changed or moved, and displays them using classes that are effectively local to the aspect. Since all of the display updating is now done in the aspect using the effectively local display class, the aspect is spectative relative to the underlying system performing shape manipulations. Even if the display object is used by other parts of the system, if an aspect only locally introduces and maintains a new field in the display, it is spectative.

As will be discussed in Section 9, in practical analysis systems for identifying spectative aspects, aliasing, inheritance, and polymorphism can significantly complicate the analysis. Dataflow and type-safety techniques such as those used in static analysis are always conservative, in that if successful, the spectative nature of the aspect is guaranteed. If the analysis does not establish that the aspect is spectative, but has not revealed a clear violation of the semantic definition, perhaps only a deeper semantic analysis is needed.

Now a key lemma about spectative aspects can be stated and proven, in terms of the classes of properties defined earlier and the underlying and augmented state graphs.
Lemma 3: If an aspect is spectative, all safety, liveness, and existence properties of the underlying system that are not next-state properties and involve assertions only about variables, fields, and methods of the underlying system will not be influenced by the aspect, and will also hold in the augmented system. Moreover, every such property true of the augmented system was already true of the underlying one.

Proof: By the definition of a spectative aspect, in terms of the variables of the underlying system, the projections to those variables of the new states due to the advice code of a spectative aspect are identical to the state before the joinpoint transition. The subsequences corresponding to states and transitions of the advice are finite and reconnect back to the state in the underlying program after the operation that activated the joinpoint transition (since a *before* advice is being considered). Because the assertions have no next-state properties, they are insensitive to repetitions of states, and thus any such linear or existence assertion true of the underlying system and involving only its variables is also true of the projection of the augmented graph. Since only variables in the projection appear in the assertions, they also hold in the augmented state graph itself.

No new properties involving only the variables of the underlying system can be added by the augmented one because, again, the projection of the state graph of the augmented system to the variables of the underlying one differs from the original underlying state graph only in the number of finite repetitions of states that already exist, and such differences cannot be distinguished without including a next-state property (i.e., a temporal formula with an **X** or the corresponding **"**previous**"** temporal operator). □

In the example, the underlying system could have invariant properties such as G(denominator>0) (i.e., in every state, the denominator is greater than 0). This will automatically hold for the augmented system with ScoringAspect, because it is spectative. A liveness property such as G(at(setNumerator) => F(at(setDenominator)) (i.e., setNumerator is always eventually followed by setDenominator) will also be automatically extended to the augmented system, if it was already true of the underlying system without the aspect.

However, it should be noted that implicitly scoped visibility properties such as "the value of a field is not visible outside the class" can be violated by spectative aspects, even when the properties were previously true of the underlying system. The problem is that the assertion of "not visible outside the class" when applied to the augmented system involves both the original variables, fields, and methods and new ones added by the aspect, and thus is different from the original assertion. Therefore Lemma 3 does not guarantee that such a property will be preserved in the augmented system. For example, within the aspect code, the value of a (hidden) field X of the underlying system could be "made visible" by examining another field Y (added by the aspect) that is given the value of X, or by adding public methods, both possible in a spectative aspect.

New properties of the augmented system that *do* involve both variables of the aspect and those of the underlying system are also easier to verify if the aspect is spectative:
Lemma 4: Invariant properties true of the variables of a spectative aspect, or connecting the aspect and underlying variables, can be established for the augmented system by separately analyzing the aspect variables not bound to variables of the underlying system in the aspect code and the other variables in the underlying code.

Proof: By the definition of a spectative aspect, variables of the underlying system and variables of the aspect bound to those of the underlying system are only changed in the underlying system. Moreover, variables of the aspect not bound to those of the underlying system are clearly changed only by the aspect advice code.

If variables of both the aspect and the underlying system are in the assertion, it is sufficient to consider separately whether the underlying variables and the aspect variables bound to them (which are only changed in the underlying system) maintain the invariant, and whether the `local' aspect variables (changed only in the aspect code) also maintain the invariant. □

Lemma 4 can allow decomposing the verification task for the augmented system to two smaller problems, especially if model-checking abstraction techniques are used. For the visibility property mentioned earlier, for spectative aspects it is sufficient to check only the aspect code in order to extend the property to the augmented system. Liveness properties added by the aspect, on the other hand, are closely connected to the liveness properties of the underlying system, because aspect advice is only executed when joinpoints of the underlying system are reached. Thus, in general, to establish new liveness properties involving the aspect, the augmented system as a whole must be considered. A simple exception is liveness properties involving only the aspect code segments, such as an assertion that each advice segment that is initiated will properly terminate without throwing exceptions.

6 REGULATIVE ASPECTS

Regulative aspects can affect the flow of control of the underlying system by restricting operations, delaying some operations, or preventing the continuation of a computation.

Definition: An aspect is **regulative** if the projection of the augmented state graph on the variables of the underlying system is identical to the state graph of the underlying system, except that some states are repeated (with new edges from aspect operations) and some edges are removed. States are ignored that become disconnected (unreachable) from the augmented state graph with entrance points (external method calls).

Note that, as for spectative aspects, a repeated state and added edge can branch off from an existing state. In a regulative aspect, that branch might lead to the repeated state, but with no continuation from there because edges have been removed. For example, in a regulative variant of Figure 2(b), there could be an additional edge from the *(x=7, w=0)* state after the join, to an identical state, but with no continuation edge. This would correspond to a branch in the aspect code that prevents the continuation. For regulative aspects, each trace of the projection is either a prefix of or the same as a trace in the original, except for repetitions of states. As before, the branching structure of the original is maintained, except for removed edges.

In terms of code, regulative aspects prevent, restrict, or delay some of the actions that were possible in the underlying system. However, they cannot simply skip some steps in a transaction of the underlying program while continuing to other steps. Then a weakly invasive aspect (see Section 7) is needed. In a simple case, a regulative aspect might make fields or methods private that were previously publicly available. Then an external message activating the method would be denied in the augmented system, while an internal one would continue to execute as before. If requirements arise to restrict access to method calls that were originally unrestricted, a regulative aspect might add a parameter with a password or authorization key, along with aspect code that continues with the original method only if the password is authorized. When the restrictions are only to external method calls, as in the password authorization example above, the aspect belongs to the special category of an **externally regulative** aspect.

In general, regulative aspects enforce additional checks before allowing the activation of methods or actions that were not restricted in the underlying system. Thus, an aspect might terminate a system if overflow of an integer variable occurs, preventing continuations that were previously possible when overflow was ignored. In Figure 4 an aspect is given that restricts initiating an exam (by the method doExam, called only externally) to children over the age of 7. Note that the (age < MIN_AGE) test corresponds to a branch that terminates with no continuation, while the negation continues the original computation. The projection to the variables of the original system has a trace with repetitions of the initial state that has no continuation, and another one that is as in the original system, with repeated states. Thus it satisfies the conditions for a restrictive aspect.

Other examples include synchronization or scheduling aspects, such as one that enforces mutual exclusion among methods in different objects that were independent and could overlap without the aspect. In this case some actions are delayed until new synchronization or mutual exclusion conditions are satisfied An aspect that enforces mutual exclusion among instances of exams when they wish to print out summaries on the same printer (so the printer will be used exclusively by one or the other) is regulative in that one of the summaries is delayed until the other completes. The edges leading to states where the two printing tasks overlap are removed.

To detect that an aspect is regulative, we proceed as for a spectative aspect as far as determining that variables are independent of assignments in the underlying system, but are more liberal about the reconnection properties of the advice code and waiting for conditions during the advice. The generic detection procedure for a regulative aspect is:
1. Determine the set C of variables in the aspect that are bound to variables of the underlying system.
2. Check that no variable of C is assigned a value by the aspect.

```
public aspect AgeRestrictionAspect {
        private static final int  MIN_AGE      = 7;
        // Pointcuts
        pointcut doingExam(RationalExam exam) :
                call (void RationalExam.doExam()) && target(exam);
        // Advice
        void around(RationalExam exam) : doingExam(exam) {
                System.out.print("Hello, how old are you? ");
                int age = Integer.parseInt(getInput());
                if (age < MIN_AGE) {
                        System.out.println("You're too young for fractions");
                        return; //returns without doing the exam
                }
                proceed(exam); //proceeds with the exam when old enough
        }
```

Figure 4. An aspect restricting method activation

3. The aspect code may contain wait conditions and restrict execution of previously independent operations of the underlying system, but each advice should be shown to terminate if the wait conditions hold.
4. Check whether one of the following reconnection conditions hold:

- resume computation of the underlying system at the joinpoint,
- throw an exception that terminates the execution or directly terminate the execution of the entire system or of the present transaction

The possibility of preventing external calls to methods of the underlying system (e.g., by adding parameters and checks on them or making methods or fields private) is included in the above, as it is viewed as terminating a transaction before it can even begin.

Lemma 5: If the detection procedure above succeeds, the aspect is regulative.
Proof: For rules 1 and 2, the reasoning is as for spectative aspects. The liberal termination policy in rule 3 means that the sections of the graph resulting from aspect advice operations should be finite, but might deadlock or terminate, corresponding to removing edges to the continuation. The reconnection conditions ensure that no code of the underlying system is skipped unless the system terminates or at least completes its reaction to the most recent input (in a transaction view). Together these guarantee that the projection of the augmented state graph satisfies the conditions in the definition of a regulative aspect, and only removes edges or adds repetitions of states, including possible branching to repeated states with no continuation. □

In principle, the syntactic conditions above are too strong, and regulative aspects could be allowed to change some variables bound to those of the underlying system, but only under conditions that are hard to check statically. The elements of C that are given values by the aspects may be bound only to variables of the underlying system that are exclusively used in conditional statements, and the aspect assignments must lead to strengthening the condition under which a method is activated. That is, the aspect leads to choosing a method in fewer cases than previously (and never to choosing one more often). Some of the static analysis or type systems for aspects discussed in Section 9 also identify special cases of regulative aspects.

Lemma 6: If an aspect is regulative, all safety properties of the underlying system that are not next-state properties and involve assertions only about variables, fields, and methods of the underlying system will not be influenced by the aspect, and will also hold in the augmented system. Liveness and existence properties are not automatically preserved.
Proof: In terms of the state graph of an augmented system with a regulative aspect, relative to the state graph for the underlying system, the projection described for spectative aspects applies, except that some edges are removed. In particular, some potential entrance points (external method calls) are restricted or closed compared to the graph of the underlying system, and some arrows are simply removed, but no new ones are added except between repetitions of states in the original graph. Note that if an operation of the original computation were simply skipped, an arrow would be added from the state before the removed operation to the state now reached by doing a step in the continuation, which is forbidden for a regulative aspect. However, it is possible for

the aspect advice to delay some actions by waiting for a global condition involving underlying variables, as in the mutual exclusion example. The other actions that are meanwhile executed could have occurred by chance before the delayed one, so only some interleavings are eliminated, and no new traces are added relative to the underlying system, except for repetitions. The remaining computations maintain the partial order among operations that are not independent.

That is, in the semantic view, the state graph of a regulative aspect prunes edges from the computation graph of the underlying system (along with adding transitions and state only involving aspect variables, as for spectative aspects). The projection of the augmented graph onto the variables of the underlying system is an edge-pruned version of the original underlying computation graph, along with repeated states. Any safety property true of the original graph has the form AGp where p only relates to the history. Since the history of each state in the projection of the augmented system is identical to the history in the original system except for repetitions of states that already appear, and the assertion has no next-state operators, it will be true of the states in the augmented system if it was true of the states in the original one.

Liveness or existence properties need not be preserved by regulative aspects, since states previously reached may be inaccessible both in the augmented system and in the pruned graph with repeated states that is the projection of the augmented system. Thus a computation that existed in the underlying system may be interrupted in the augmented one, and a state that eventually occurred in the original may not appear in the pruned version. □

Recall that for spectative aspects, no new properties involving only the variables of the underlying system can be added in the augmented one. However, for regulative aspects, besides maintaining safety properties already true, there can be new safety properties even involving only the variables of the underlying system.
Lemma 7: An augmented system with a regulative aspect can have additional safety properties involving only the variables, fields, and methods of the underlying system that were not true in the underlying system.
Proof: As seen above, in an augmented system with a regulative aspect some system states of the original system can be unreachable. Thus, for example, new invariants may hold for all reachable states of the augmented system even if they did not hold for some states of the underlying one, because the problematic states become unreachable in the augmented system. In fact, one of the reasons for weaving a regulative aspect into an underlying system is to eliminate problematic states (that violate desired invariants) by making them unreachable in the state graph of the augmented system. □

In Figure 5(a), a fragment of an original state graph with state variables *X* and *Y* shows independent operations a and b, perhaps activated from different objects. The operation a increments *X* by one, while b does the same for *Y*. In the augmented version, a state component of the aspect, *S*, is added, to restrict which operations are allowed, yielding the augmented state graph in Figure 5(b). This fragment satisfies the conditions for a regulative aspect, because in the projection of 5(b) to *X* and *Y*, the first state of the original is repeated, and the edge labeled b is removed, making the state (*X*=1, *Y*=1) unreachable, so it can be removed. In the augmented version, the invariant G(*X* > *Y*) is true, while in the original it is not.

For an example of a regulative aspect that adds a mutual exclusion property by introducing a local mutex object, the invariant G(~(in(crit1) /\ in(crit2))) (meaning, in every state, we are not both in critical section 1 and in critical section 2 at the same time) becomes true in the augmented system even though it did not hold in the original system. States where it did not hold in the original system have been eliminated from the state graph of the augmented one by preventing the transitions that lead to a violation.
Lemma 8: If a regulative aspect only restricts external method calls from outside the original system and is thus externally regulative, then existential properties may not be preserved, but all safety and liveness properties preserved by a spectative aspect are also preserved for a resultant augmented program with such a regulative aspect.

Proof: In this case, only potential computations are limited, but all safety and liveness properties within the system (about the state graph of the original or augmented system) are maintained. Since the environment cannot be forced to actually make a particular method call even in the original system, there could not have been a liveness property that guaranteed its occurrence. □

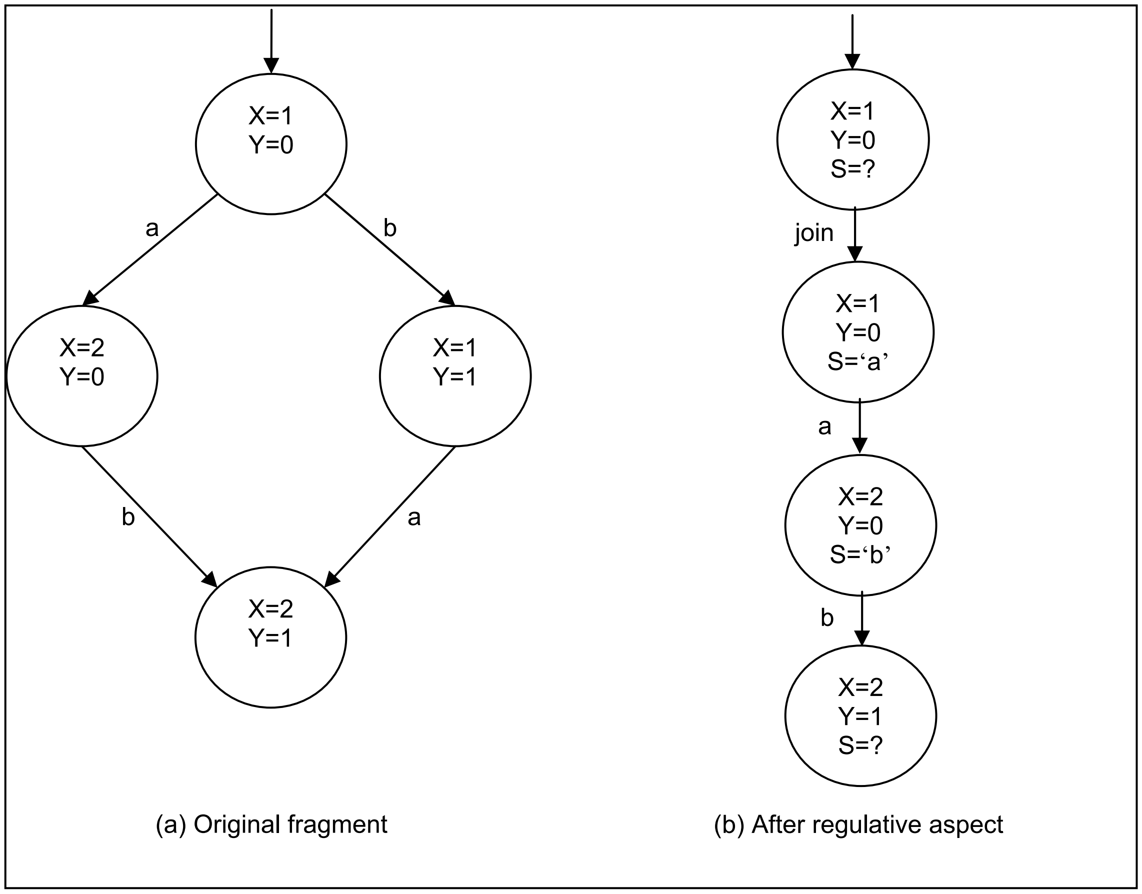

Figure 5. State graph fragments for a regulative aspect

For example, the aspect restricting initialization of an exam on fractions is externally regulative. Thus it does not influence any of the internal safety or liveness properties, but does mean that there will not be any computations where the age of the user is declared to be under 7.

7 INVASIVE ASPECTS AND PARTIAL ANALYSIS

7.1 Weakly invasive aspects

Invasive aspects do change the values of variables in the underlying system. Thus in principle, they could completely invalidate any property that held previously in the underlying system. Nevertheless, in many situations invasive aspects change the underlying system in restricted ways suitable for detection and exploitation. The basic question about an invasive aspect is how it reconnects to the computation from the underlying system. This will influence the extent to which the aspect influences the continuation of the computation. Even for a *before* aspect, although the code of the underlying system continues executing after the advice, the advice may have created a new state that did not exist in the underlying system. Thus in the augmented system even the code from the underlying system can continue to create an entirely new subgraph whose properties are unknown. In this situation it seems difficult to avoid having to consider the entire augmented system at once. An interesting special case can help alleviate the difficulty.

Definition: An aspect is **weakly invasive** if in the state graph of the augmented system, transitions that correspond to operations from the underlying system begin in states that already existed in the state graph of the underlying system (perhaps for different inputs from those in the augmented system).

It follows that advice transitions end in states that were in the original state graph, at least whenever an underlying operation can then be executed. Although it can be difficult to identify when an aspect is weakly invasive only using static analysis, there are special cases where this is possible. Otherwise, the user can simply declare that the aspect is weakly invasive, for cases where this is obvious. For example, an aspect may identify conditions (joinpoints) at which it is necessary to *reset* the underlying variables to some fixed values that also occur in the underlying system without the aspect. In a transaction control system, an aspect can be used to record values and later *rollback* the computation to the earlier state with the recorded values when a transaction cannot be completed. In an

```
public aspect ReductionAspect {
        // Pointcut
        pointcut rationalChanging(Rational r) :
                (call (void Rational.set*(int)) || …) && target(r);

                // Advice
        after(Rational r) returning : rationalChanging(r) { reduce(r); }
```

Figure 6. An aspect to reduce fractions

aspect to impose a discount policy for prices, the prices are changed, but the continuation is equivalent to one in which the price was originally set to the value now obtained through the discount policy. All of these can be identified as weakly invasive aspects.

In Figure 6, an aspect that changes the values chosen for the arithmetic operations on fractions to reduced fractions is outlined. The partially specified pointcut identifies where new values are given to a fraction, and a method to reduce them is then activated before continuing. (Details of the method *reduce* which performs the reduction, and thus changes the value of the fraction, are not shown.) This aspect is weakly invasive because the original choice of values might already be reduced, so the state reached is a possible state of the original system.

Lemma 9. If an aspect is weakly invasive, then an invariant true for the underlying system can be proven to hold for the augmented system by assuming it true at the beginning of each aspect advice, and showing it true for each state generated during the execution of the advice.

Proof: By induction on the joinpoints reached in any computation of the augmented system. To base the induction, the invariant is true in all states until the first joinpoint is reached, since only states of the underlying computation are reached. For the general case, assume the invariant is true until a joinpoint is reached. If the invariant then also holds for all states generated by the advice code until the computation reconnects to a state that was in the underlying one, then it will continue to hold until the next joinpoint because it is true in all states of the underlying state graph. □

The implications of this lemma are that to extend an invariant to an augmented system for a weakly invasive aspect, the original system does not have to be rechecked, and the entire augmented system does not have to be considered at once. It is sufficient to reason only about the much smaller state graphs that correspond to the advice of the aspect. Since the lemma holds for global invariants, it clearly also holds for class invariants, invariants at a location, and partial correctness assertions, since all these can be expressed as a global invariant.

In the weakly invasive ReductionAspect, above, if an invariant G(denominator>0) was true of the original system, then to extend it to the augmented system it is sufficient to show that it is maintained by the *reduce* method.

Note that general safety properties do *not* have to be extended to augmented systems with weakly invasive aspects, even when the property holds within the aspect state graph, because the history of how a state is reached can be different in the underlying and in the augmented system. For example, a safety property "a state satisfying P is always preceded by a state in the history satisfying Q" could be true for the underlying system, and could hold for the state graph of the aspect itself, and yet not be true in the augmented system. This could be because the reconnection is to a predecessor of a state satisfying P, and yet Q is not true anywhere in the new history including the aspect. Moreover, liveness and existence properties are also not similarly extended to the augmented system with weakly invasive aspects, and there does not seem to be such a clean separation of reasoning, because states from the underlying system can be skipped or made unreachable in the augmented system.

7.2 Identifying independence for invasive aspects

Another direction for simplifying reasoning for invasive aspects considers the degree of invasiveness. In other words, the aspect usually will only affect some of the variables in the underlying system, and identifying independent parts can allow simplifications similar to those for spectative and regulative aspects. A variable x of the underlying system is *independent* of an aspect if no variable that is assigned values in the aspect code is bound to x and all variables used in the computation of x or that influence conditionals that control basic blocks with assignments to variables that influence x, are also independent of the aspect. As previously, standard dataflow techniques from optimizations in compilers can be used to detect independence in many cases.

Lemma 10: An invariant property of the underlying system that does not include a next-state property and only includes variables *independent* of an aspect, is maintained in an augmented program with that aspect woven into the underlying program.
Proof: Due to the definition of `independent', even an invasive aspect does not influence any of the computation that affects the values of the independent variables in the invariant property, and does not add any new computation affecting those variables. This includes not influencing the control flow of assignments to the variables in the invariant. An invariant only relates those values within each state. Thus the invariant property is also true in the augmented system. □

Note that general safety properties could be affected because of changes in the history. Stronger notions of independence could be defined to treat such cases, but may be difficult to establish through syntactic analysis.

7.3 Verifying inductive invariants for invasive aspects

When an invariant of the underlying system involves variables that are changed by the aspect, or the invariant includes variables from both the aspect and the underlying system, a simplified verification is nevertheless possible for a restricted class of invariant properties. Consider extending an invariant that was true of an underlying system when the aspect applied is (strongly) invasive, i.e., changes the values of variables in the underlying system, redirects control, and need not resume executing the code of the underlying system where it was suspended. If the invariant is what is known as "inductive", it can be shown to also be an invariant of the augmented system by only checking the advice, even without analyzing in what situations the aspect code will be applied and without rechecking the code of the original system.

An invariant I is *inductive* if {I} s {I} can be shown directly for each individual step s, without any knowledge of the state before the step s except that it satisfies I. In some sense, it means that the assertion is "self-contained" and by assuming just itself, a proof can be constructed to show that it is reestablished after each step. Note that even states that do not occur in the system but for which I is true are sufficient preconditions to guarantee I after s is executed.

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}.
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.

Assume that the invariant G(denominator>0) was inductive for the fractions system, i.e., it was provable for each step of the original system only assuming itself. Then even if the *reduce* method in the above example could introduce negative fractions that were not treated in the original system, only the aspect advice must be newly checked. Note that the aspect now is strongly invasive, since states not in the original system are generated at the end of the advice, and the exact effect of applying code of the basic system to these new states is unknown. Still, the new resultant states will satisfy the invariant. On the other hand, if the invariant was not inductive, in the original proof the fact that the fractions were nonnegative may have been true before some step and used to help establish the invariant after the step. In that case a new proof is needed for the entire augmented system now that the aspect introduces new states.

To summarize this discussion:
Lemma 11: If an invariant has been shown to be inductive, and proven for an underlying system by only using the invariant itself as an assumption (i.e., with assertions {I}s{I} in Hoare logic notation, for each statement s of the original system), then the invariant can be extended to hold for an augmented system with an invasive aspect, by proving {I}t{I} for each statement t only of the aspect code, without reconsidering the original system.
Proof: Since I is already known to be an invariant of the original system, it 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 from the code of the aspect, and it is proven that if I holds, then it will again hold after t, for any state. Thus, I is an invariant of the augmented system, even without rechecking the original code. □

8 INTERFERENCE AMONG ASPECTS

So far the relation has been considered between an underlying system and an augmented system with a single aspect added to the underlying one. However, most augmented systems have multiple aspects applied to them, and the issue of whether these aspects interfere with each other is of obvious concern. Several compositions of aspects are possible, as described in [27]. In some cases, there is a clear ordering among the aspects to be applied. In effect, a later aspect is actually applied to the augmented system obtained after applying an earlier aspect. In this case the joinpoints of the later aspect may include events generated by the earlier one. The treatment in earlier sections, of a single aspect woven to an underlying system, may be used iteratively. That is, once the properties of an augmented system with one aspect have been established, possibly using the lemmas above, it can be considered as an underlying system relative to another aspect. However, the syntactic analysis to determine possible interactions becomes more complicated. Program slicing for aspects, as seen in the following section, can be used to determine the category of aspect relative to an underlying system that itself has aspects.

In other situations, several aspects are to be applied independently only to the underlying system. In this case we clearly do not want any direct binding of variables, objects, methods, or parameters local to one aspect to those elements of another aspect. This also can be enforced syntactically, and checked. Still, when pointcuts of several aspects define the same joinpoints, and parameters of more than one aspect are bound to the same underlying variable, indirect (usually unintended) connections can occur. From the property preservation claims seen earlier, we have:

Lemma 12: If there are no direct bindings among aspects, spectative aspects can be applied in any order without influencing safety, liveness, or existence properties of the result. Spectative aspects should be applied before restrictive or invasive ones, and will not influence their safety, liveness, or existence properties.
Proof: An augmented system with a spectative aspect resumes from the state of the joinpoint after executing advice, since only new variables of the aspect are changed. Moreover, the assumptions of an aspect specification about the underlying program and joinpoints to which that aspect may be applied only relate to variables of the underlying program and local variables of that aspect. Therefore, the assumptions of any aspect B will not be influenced by applying a spectative aspect A first, even at joinpoints where both are activated. Since the aspect is presumed correct when woven over any underlying system satisfying its assumptions, it remains correct relative to its specification when a spectative aspect is applied previously. □

For multiple regulative or invasive aspects, their order of application changes the semantics (i.e., the resultant computation graphs of the augmented systems may be different for different orderings). As previously, a finer analysis of partial independence can be made to determine allowable orderings.

9 RELATED ASPECT CATEGORIES AND STATIC ANALYSIS TOOLS

Additional categories of aspects have been defined in several works. Although in [6] a tool was not developed, that work defines an *observer* aspect (roughly similar to a spectative one). It also points out difficulties of aliasing that can complicate syntactic static analysis of code to determine whether an aspect is an observer and suggests a development methodology using observers. In [17], an aspect is defined to be *harmful* if it invalidates any desired properties of the system to which it is applied. As seen in the lemmas, one way to exploit the categories in this paper is to help determine whether desirable properties true of the underlying system are still true in the augmented one.

Clearly, the categories here can help in automatically detecting whether there is any danger of harmful aspects, once the specification properties are identified as belonging to the appropriate classes of temporal logic properties. Then a system can be unaware of, or *oblivious* [11][12] to, the particular aspects to be applied to it, but its specification can nevertheless restrict new aspects to those that do not violate key properties, so that the system specification can influence which aspects are woven. The obliviousness of the system is thus slightly restricted.

Several works have concentrated on automatic identification of aspect categories either based on static code analysis for aspects using dataflow, or with transformation rules that allow deriving only aspects of a desired type. The code analysis system for a simplified aspect language seen in [26] is intended primarily for code optimization, but the information gathered can also be (and has been) used by the authors to identify spectative aspects. Similarly, in [29] an extensive interference analysis is made for real Java and AspectJ-like programs, emphasizing the complications introduced by inheritance and multiple instances. Again, the result is the effective syntactic identification of spectators/observers and non-overlap among aspects (which is there called *interference-freedom*).

In [25] finer distinctions are defined, but the basic categories resemble those already described. That work concentrates on determining the relations between an aspect and a method of the underlying system. If they are *orthogonal*, the two access disjoint fields, if they are *independent* neither writes to a field that the other may read or write (but both may read the same field), in an *observation* relation the advice may read fields that the methods may write, *actuation* means that the advice may write to a field that the method may read, but they are otherwise independent, and *interference* means that both may write to the same field. If the aspect code segments can also be shown to always terminate and not redirect control, the first three correspond to spectative aspects. Otherwise, if basic code is not skipped, they are regulative. The later two categories are invasive, but could be further analyzed to identify special cases of weakly invasive aspects. The tool described in [25] uses standard dataflow techniques for AspectJ over Java, and numerous sample programs have been analyzed.

There is also considerable work ([1], [2], [33]) on extending well-known programming slicing techniques based on dataflow to aspects. These are used to identify the extent of influence of an aspect on the underlying system, and to identify potential conflicts among aspects. In effect, any potential interactions or conflicts are identified. Such techniques can also be used to reduce the size of the model that must be analyzed when model-checking techniques are to be applied. In [2] an implemented slicing system for AspectJ is presented to identify the influence of each aspect. Similar ideas to detect interference has been undertaken for Composition Filters with the Compose* analysis system [23].

Although the systems above do not explicitly deal with families of properties, they are usually demonstrated on spectative aspects as tests. However, many of them either ignore or simply assume termination of the aspect code (some assume straight-line code). Generally, the code of the basic system is resumed from the joinpoint after the aspect advice, and thus the aspects are shown regulative. Lemma 6 is therefore applicable, which means that safety properties are necessarily extended to the augmented system if they held in the original, but liveness might not be preserved.

Correctness-preserving transformation systems guaranteeing that only aspects of a particular type are generated are another approach to establishing aspect categories. An abstract transformational theory is presented in [10] to prevent interference among aspects (discussed in Section 8), emphasizing pointcut conflicts for event-based aspects. In [8] a type system and transformation rules are presented to guarantee *harmless advice*. This is defined as advice that does not change the final values produced by the original system, if the augmented system still terminates normally. In the terms here, partial correctness properties are preserved. Since the rules are informally described in their papers as guaranteeing regulative aspects, it is likely that their results can be extended to general safety properties, again using Lemma 6.

In [17] *aspect-aware interfaces* are suggested as a weakening of obliviousness, on the code level. The aspects activated in a module are explicitly identified in these interfaces so that the effects of method activations can be more easily understood for modularity and analysis purposes. Although the construct is orthogonal to the semantically defined categories in this paper, it can help in the syntactic identification of aspect categories by making the connections among aspects and objects more explicit and isolating the effect of advice to parts of the underlying system. In [28] and [13] *crosscut programming interfaces* are introduced to help modularize and decouple aspects from the implementation details of the underlying system when defining pointcuts. These also provide a convenient mechanism for specifying aspects through preconditions and postconditions, supporting the assume-guarantee specifications suggested in Section 2.

In [20] aspect advice represented directly as a state transition graph is model checked to extend invariants proven true for the original system to the augmented one. They assume that the advice reconnects to the state transition graph of the original system at the joinpoint. However, they do not identify parts of the statespace local to the aspect. If the changes made by the advice are to parts of the state not used in the original system, the aspect would be spectative, and the model check is extraneous, since all safety properties are maintained. If the original state variables *are* changed, they must be changed back to the original values before ending the advice, by assumption. In that case, their aspect is weakly invasive, and Lemma 9 applies, justifying a model check of only the aspect state graph, as they also show.

10 SUMMARY

The categories of aspects, the procedures for their identification, and the semantic influence of aspects from each category on classes of properties provide the basis for a code-analysis module for aspects.

The distinction between the semantic definitions of the categories and restrictions sometimes needed in order to syntactically determine to which category an aspect belongs can be seen by considering whether the aspects can contain “call-backs” to methods of classes already declared in the original system, beyond a *proceed* statement. The absence of such method calls can be easily determined by static analysis, and may help in determining that the aspect is, for example, spectative. However, there is no real semantic problem with such calls. As seen in Section 6, a class declared in the original system may have objects effectively local to the aspect. Moreover, even for objects defined and used in the original system, methods that only return values can be activated by spectative aspects, to obtain information not explicitly in the pointcut definition. Only methods that change values in the original object, or change the conditions under which method calls in the original program may be activated are forbidden. Similar considerations hold for regulative and weakly invasive aspects.

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, identifying categories of aspects through syntactic analysis is a valuable exercise. A significant improvement in code reliability and quality can be obtained at a relatively low cost, especially when specifications of the underlying system and the aspects are available, or at least the classes of properties needed to express desired features are understood. Proper language design for aspects, with local variables and parameterization, can help extend the static analysis of only the aspect code, either for classes of properties and for every possible weaving, or by reanalyzing only the aspect for each weaving. Even for invasive aspects, partial syntactic analysis can be useful and can ease the task of establishing the properties of the augmented system.

As the level of interference of the aspect increases, from spectative, to regulative, to weakly invasive, to strongly invasive, the classes of properties which are automatically extended or have modular proofs become successively smaller. Thus, for spectative aspects all safety, liveness, and existence properties without next-state properties are maintained, while for regulative ones, just safety properties are automatically maintained. For weakly invasive aspects, safety properties are not maintained, but any invariants of the underlying system can be extended to the augmented one by only checking the advice. Finally, for strongly invasive aspects, only the restricted class of inductive invariants can be modularly extended to the augmented system.

As already noted, syntactic checks to determine the category of aspect use dataflow techniques also seen in code optimizations, and therefore should be incorporated into the compilation and optimization of aspect languages. In addition to the analysis systems already developed, detection of the new categories of externally regulatory and weakly invasive aspects seems valuable, since they both are practically widespread, and significantly easier reasoning about properties is possible for augmented systems with aspects in these categories.

A tool for static analysis to determine aspect categories could connect to further analysis, testing, or verification tools (e.g., [16] or [20]) by determining the easiest verification techniques to extend properties from the underlying system, establish new properties, or further analyze the system when a conservative analysis indicates possible interferences that may not occur in practice.

ACKNOWLEDGMENTS
I would like to thank the referees of this paper for their valuable and constructive comments. I also acknowledge support from AOSD-Europe, a Network of Excellence in the FP6 IST of the European Union.

REFERENCES

[1] D. Balzarotti and M. Monga, Using program slicing to analyze aspect-oriented composition, Foundations of Aspect Languages (FOAL) Workshop, 2004.
[2] D. Balzarotti, A. C. D'Ursi, L. Cavallaro, and M. Monga, Slicing AspectJ woven code, Foundations of Aspect Languages (FOAL) Workshop, 2005.
[3] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely, μabc: a minimal aspect calculus, CONCUR 2004, LNCS 3170, September 2004.
[4] E.M. Clarke and E.A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, Workshop on Logics of Programs, LNCS 131, 1981, pp. 52-71.
[5] E.M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.
[6] C. Clifton and G. Leavens, Observers and assistants: a proposal for modular aspect-oriented reasoning, Foundations of Aspect Languages (FOAL) Workshop 2002. (also, modified as Spectators and assistants: enabling modular aspect-oriented reasoning, Iowa State TR02-10, 2002).
[7] D. Dams, R. Gerth, and O. Grumberg, Abstract interpretation of reactive systems, ACM *Transactions on Programming Languages and Systems*, 19(2), 1997.
[8] D. Dantas and D. Walker, Harmless advice, Foundations of Object Languages (FOOL) Workshop, January, 2005; extended version to appear in Principles of Programming Languages (POPL), 2006.
[9] B. Devereux, Compositional reasoning about aspects using alternating-time logic, Foundations of Aspect Languages (FOAL) Workshop, associated with AOSD 2003.
[10] R. Douence, P. Fradet, and M. Sudholt, Trace-based aspects. In *Aspect-Oriented Software Development* (M. Aksit, S. Clarke, T. Elrad, and R. Filman, eds.), Addison-Wesley, 2004.
[11] R.E. Filman and D.P. Friedman, Aspect-oriented programming is Quantification and Obliviousness, Workshop on Advanced separation of Concerns, OOPSLA 2000.
[12] R.E. Filman, What is AOP, Revisited, Workshop on Advanced Separation of Concerns, 15th ECOOP, June, 2001.
[13] W. Griswald, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, H. Rajan, Modular software design with crosscutting interfaces, IEEE Software, Special issue on Aspect-Oriented Porgramming, to appear, 2006.
[14] J. Hatcliff and M. Dwyer, Using the Bandera tool set to model check properties of concurrent Java software, CONCUR2001, LNCS 2154, 2001, pp. 39-58.
[15] R. Jagadeesan, A. Jeffrey, and J. Riely, A calculus of untyped aspect-oriented programs, ECOOP 2003, LNCS 2743, pp. 54-73, 2003.
[16] S. Katz and M. Sihman, Aspect validation using model checking , Symposium on Verification in honor of Zohar Manna, Springer-Verlag, LNCS 2772, 2003, pp. 389-411.
[17] S. Katz, Diagnosis of harmful aspects using regression verification, Foundations of Aspect Languages (FOAL) Workshop associated with AOSD 2004.

[18] G. Kiczales, et al, An overview of AspectJ, 16th ECOOP, 2001.
[19] G. Kiczales and M. Mezini, Aspect-oriented programming and modular reasoning, Intl. Conference on Software Engineering (ICSE), 2005, pp. 49-58.
[20] S. Krishnamurthi, K. Fisler, and M. Greenberg, Verifying aspect advice modularly, Foundations of Software Engineering (FSE) Conference, 2004.
[21] L. Lamport., What good is temporal logic?, IFIP 9th World Congress, 1983, pp. 657-668.
[22] Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems—Specification, Springer-Verlag, 1991.
[23] I. Nagy, L. Bergmans, and M. Aksit, Declarative aspect composition, SE Properties of Languages and Aspect Technologies (SPLAT)Workshop of AOSD04, 2004.
[24] H. Ossher and P. Tarr, Multi-dimensional separation of concerns and the Hypersapace approach. In M. Aksit, editor, *Software Architectures and Component Technology*, Kluwer Academic, 2001.
[25] M Rinard, A. Salcianu, and S. Bugrara, A classification system and analysis for aspect-oriented programs, Foundations of Software Engineering (FSE) Conference, 2004.
[26] D. Sereni and O. de Moor, Static analysis of aspects, Aspect-Oriented Software Development (AOSD) 2003, pp. 30-39
[27] M. Sihman and S. Katz, Superimposition and aspect-oriented programming, The Computer Journal, 46 (5), 2003, pp. 529-541.
[28] H.B. Sipma, A formal model for cross-cutting modular transition systems, Foundations of Aspect Languages (FOAL) Workshop associated with AOSD 2003.
[29] M. Storzer and J. Krinke, Interference analysis for AspectJ, Foundations of Aspect Languages (FOAL) Workshop, 2003.
[30] K. Sullivan, W.G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, Information hiding interfaces for aspect-oriented design, European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE), 2005, pp. 166-175.
[31] D. Walker, S. Zdancewic, and J. Ligatti, A theory of aspects. ICFP'03, ACM Press, pp. 127-139, 2003.
[32] M. Wand, G. Kiczales, and C. Dutchyn, a semantics for advice and dynamic join points in aspect-oriented programming, Transactions on Programming Languages and Systems (TOPLAS), 26 (5), 2004, pp. 890-910.
[33] J. Zhao, Slicing aspect-oriented software, IEEE International Workshop on Programming Comprehension, 2002, pp. 251-260.