to appear in *Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE'04),* Springer-Verlag LNCS, 2004.

A Generative Approach to Aspect-Oriented Programming

Douglas R. Smith
Kestrel Institute
Palo Alto, CA 94304 USA
smith@kestrel.edu

Dedicated to the memory of Robert Paige

Abstract

Aspect-Oriented Software Development (AOSD) offers new insights and tools for the modular development of systems with cross-cutting features. Current tool support for AOSD is provided mainly in the form of code-level constructs. This paper presents a way to express cross-cutting features as logical invariants and to use generative techniques to produce the kind of code that is usually written manually in AOSD. In order to state invariants that express cross-cutting features, we often need to reify certain extra-computational values such as history or the runtime call stack. The generative approach is illustrated by a variety of examples.

1 Introduction

Aspect-Oriented Software Development (AOSD) contributes to the broad goal of modular programming, with a particular focus on cross-cutting concerns [1, 2]. A concern is cross-cutting if its manifestation cuts across the dominant hierarchical structure of a program. A simple example is an error logging policy – the requirement to log all errors in a system in a standard format. Error logging necessitates the addition of code that is distributed throughout the system code, even though the concept is easy to state in itself. Cross-cutting concerns explain a significant fraction of the code volume and interdependencies of a system. The interdependencies complicate the understanding, development, and evolution of the system.

AOSD, as exemplified by AspectJ [3], is based on a key insight: many cross-cutting concerns correspond to classes of runtime events. One can think of aspects as providing a kind of "whenever" construct: whenever an event of type e occurs during execution, perform action a. For example, whenever an exception is thrown, log it. The runtime events are called *join points*, descriptions of join points are called *pointcuts*, and the method-like actions to apply at joinpoints are called *advice*. An *aspect* is a modular treatment of a crosscutting concern that is composed of pointcuts, advice, and other Java code. The process of detecting when events of type e occur and folding action a into the code may occur statically (i.e. the actions are folded into the source

code), or dynamically (e.g. in Java, the actions are triggered by tests for the e events in the JVM).

AOSD typically has a programming language character in order to attract a wide user community. AspectJ for example [3] targets Java programmers: aspects are written in a Class-like syntax and have a semantics that closely adheres to Java semantics. Despite its attractiveness to programmers, this code-level approach has its disadvantages. First, the intent of an aspect may not be clear, since it is stated operationally. Second, the correctness of an aspect may be difficult to ascertain or to prove. Third, the completeness of the pointcut (i.e. will the pointcut apply to all occurrences of the intended runtime events?) may not be obvious. Fourth, the advice method may need to embody many case distinctions that cater for the various contexts that arise at the joinpoints, giving rise to inefficiency and complexity. Fifth, aspects may interfere with one another – since the order in which they are applied makes a semantic difference, the burden is on the programmer to order them and resolve interferences. Sixth, evolution of the program may require extending the pointcut and advice description to cover any extra cases (which requires an understanding of the modified set of runtime events being targeted, and what code to execute in each specific context of occurrence). In addition, the AspectJ pointcut description language has limited expressiveness; for example, security constraints typically disallow certain behavior patterns which do not correspond to a natural class of run-time events (cf. [4]).

In this paper we present a generative approach to AOSD that aims to overcome these disadvantages. Generative programming has the broad goals of reducing the effort in producing and evolving programs by generating them from high-level models or specifications. The result is higher assurance of correctness, greater productivity through automation, and reduced cost of evolution by allowing change at the model level versus the code level. In a sense, AOSD is a special case of generative programming, with a valuable special emphasis on treatment of crosscutting concerns.

Our approach is (1) to express an aspect by a logical invariant, and (2) to generate code to maintain the invariant throughout the system. The generated maintenance code corresponds to statically woven advice in AspectJ, and could be expressed either directly as AspectJ aspects, or by direct generation and insertion of maintenance code into the system. To state invariants that express cross-cutting features often entails the need to reify certain extra-computational values such as history or the runtime call stack.

The generative techniques in this paper derive from transformational work on incremental computation, in particular Bob Paige's pioneering work on Finite Differencing [5]. Finite Differencing is intended to optimize programs by replacing expensive expressions by new data structures and incremental computation. It achieves this by maintaining invariants of the form $c = f(x)$ where c is a fresh variable, x is a vector of program variables, and $f(x)$ is an expensive expression (usually in a loop). Code to maintain the invariant is automatically generated and inserted at points where the dependent variables change.

After introducing some notation, we work through a variety of examples and conclude with general discussion and directions for future work.

2 Preliminaries

For purposes of this paper, a behavior of a program can be represented graphically as alternating states and actions

$$state_0 \xrightarrow{act_0} state_1 \xrightarrow{act_1} state_2 \xrightarrow{act_2} state_3 \cdots$$

or more formally as a sequence of triples of the form $\langle state_i, act_i, state_{i+1} \rangle$, where states are a mapping from variables to values, and actions are state-changing operations (i.e. program statements). The details of representing an action are not important here, although some form of concrete or abstract syntax suffices. The representation is a system-, language- and application-specific decision. The operators *nil* and *append*, written $S :: a$, construct sequences, including behaviors. The selectors on behaviors are

$$\begin{array}{rcl} preState(\langle state_0, act, state_1 \rangle) & = & state_0 \\ action(\langle state_0, act, state_1 \rangle) & = & act \\ postState(\langle state_0, act, state_1 \rangle) & = & state_1 \end{array}$$

If x is a state variable and s a state, then $s.x$ denotes the value of x in s. Further, in the context of the action triple $\langle state_0, act, state_1 \rangle$, x will refer to the value of x in the preState, $state_0.x$, and x' refers to the value in the postState, $state_1.x$.

Several higher-order operators will be useful:

image: Written $f \star S$, computes the image of f over a sequence S:

$$\begin{array}{rcl} f \star nil & = & nil \\ f \star (S :: a) & = & (f \star S) :: f(a) \end{array}$$

filter: Written $p \triangleright S$, computes the subsequence of S comprised of elements that satisfy p:

$$\begin{array}{rcl} p \triangleright nil & = & nil \\ p \triangleright (S :: a) & = & \mathit{if}\ p(a)\ \mathit{then}\ (p \triangleright S) :: a\ \mathit{else}\ p \triangleright S \end{array}$$

always: The temporal logic operator $\Box I$ holds for a behavior when I is true for each state in the sequence.

We specify actions in a pre- and post-condition style. For example, the specification

assume: $x \geq 0$
achieve: $x' * x' = x \ \wedge \ x' \geq 0$

is satisfied by the action $x := \sqrt{x}$.

This paper presents its results in a generic imperative language framework, even though most AOSD approaches target object-oriented languages and even though some of the details of static analysis and code generation are necessarily language-specific. The specifications that we work with are sufficiently abstract that we believe it will not be difficult to generate code in most current programming languages.

3 An Example

A simple example serves to introduce the technique: maintaining an error log for a system. More precisely, whenever an exception handler is invoked, we require that an entry be made in an error log.

The overall approach is to specify an invariant that gives a declarative semantical definition of our requirement, and then to generate aspectual code from it. First, what does the error log mean as a data structure? Informally, the idea is that at any point in time t, the error log records a list of all exceptions that have been raised by the program up to time t. In order to formalize this we need some way to discuss the history of the program at any point in time.

Maintaining a history variable

The execution history of the program can be reified into the state by means of a *virtual* variable (also called a shadow or ghost variable). That is, imagine that with each action taken by the program there is a concurrent action to update a variable called *hist* that records the history up until the current state.

$$s_0 \xrightarrow[hist := hist::\langle s_0, act_0, s_1\rangle]{act_0} s_1 \xrightarrow[hist := hist::\langle s_1, act_1, s_2\rangle]{act_1} s_2 \xrightarrow[hist := hist::\langle s_2, act_2, s_3\rangle]{act_2} s_3 \cdots$$

Obviously this would be an expensive variable, but it is only needed for specification purposes, and usually only a residue of it will appear in the executable code.

Invariant

Given the history variable, $action \star hist$ represents the sequence of actions so far in the execution history. To express the invariant, we need a test for whether an action represents an error; i.e. whether it represents the invocation of an exception handler. Let $error?(act)$ be true when act is an exception, so $error? \triangleright action \star hist$ is the sequence of error actions so far in the execution history.

We can now represent the semantics of the error log:

Invariant: $\Box\ errlog\ =\ error? \triangleright action \star hist$

i.e. in any state, the value of the variable $errlog$ is the sequence of error actions that have occurred previously.

The idea is that the programmer asserts this formula as a requirement on the code. It is a cross-cutting requirement since exceptions can be raised anywhere in the code, regardless of its structure.

Disruptive Code and Static Analysis

In order to enforce the invariance of the asserted formula, we must find all actions in the code that

could possibly disrupt the invariant, and then generate new code for maintaining the invariant in parallel with the disruptive action. The set of all code points that could disrupt the invariant corresponds to the AspectJ concept of code points that satisfy a pointcut. The maintenance code that we generate for each such disruptive code point corresponds to a point-specific instance of the advice of an aspect.

Generally, the characterization of disruptive code is based on the Liebniz or substitutivity rule:

$$x = x' \implies I(x) = I(x')$$

where x is the vector of state variables and $I(x)$ is the invariant. The disruptive actions are necessarily those actions in the code that might change the dependent variables of the invariant. A static analyzer would be used to (1) find all actions in the source code that could possibly change the dependent variables of the invariant, and (2) when possible, run inexpensive tests to determine if the invariant is actually violated by the action. For each potentially disruptive action that the static analyzer finds, action-specific maintenance code needs to be generated.

In our example, the dependent variable of the invariant is *hist*, which is changed by every program action. The *error*? predicate serves as an inexpensive test that an action might violate the invariant. A static analyzer would scan the code (i.e. the abstract syntax representation of the code) looking for all actions that satisfy *error*?.

Specification and Derivation of Maintenance Code

Suppose that *act* is an action such that $error?(act)$. In order to preserve the invariant, we need to perform a maintenance action that satisfies

assume: $errlog = error? \triangleright action \star hist$
achieve: $errlog' = error? \triangleright action \star hist'$

The postcondition can be simplified as follows:

$errlog' = error? \triangleright action \star hist'$

$\equiv$ {using the definition of *hist*}

$errlog' = error? \triangleright action \star (hist :: \langle \_, act, \_ \rangle)$

$\equiv$ {distributing $action\star$ over :: }

$errlog' = error? \triangleright ((action \star hist) :: act)$

$\equiv$ {distributing $error? \triangleright$ over ::, using assumption that $error?(act)$ }

$errlog' = (error? \triangleright action \star hist) :: act$

$\equiv$ {using the precondition/invariant inductively }

$errlog' = errlog :: act$

which is easily satisfied by the simple update

$$errlog \ := \ errlog :: act.$$

This maintenance action is to be performed in parallel with *act*. Again, note that this generated maintenance code corresponds to an instance of an aspect's advice that is applicable where *act* occurs in the source code.

More generally, suppose that static analysis has identified an action *act* as potentially disruptive of invariant $I(x)$. If *act* satisfies the specification

assume : $P(x)$
achieve : $Q(x, x')$

then the maintenance code *maint* can be formally specified as

assume : $P(x) \ \wedge \ I(x)$
achieve : $Q(x, x') \ \wedge \ I(x')$

Code for *maint* often takes the form of a parallel composition

$$act||update$$

of the actions *act* and *update*. Implicit in this specification is the need to preserve the effect of *act* while additionally reestablishing the invariant I. If it is inconsistent to achieve both, then the specification is unrealizable.

By specifying a maintenance action that includes the existing disruptive action, we generalize several options. In AspectJ, programmers must specify whether the maintenance code goes before, after, or replaces the disruptive action. These alternatives all satisfy our specification. A further possibility is that the satisfaction of one constraint may cause the violation of another constraint, triggering an action to maintain it, causing the violation of another constraint, and so on. Our specification may need to be realized by maintenance code that iterates to a solution (as in constraint propagation systems, cf. [6]). Another possibility is that multiple invariants may need to be maintained, and it may be possible (or necessary) to implement the disruptive action and various maintenance actions differently to find a mutually satisfactory solution.

Unfortunately, most programming languages do not have a parallel composition control structure, despite its naturality. This fact has prompted most related work on programming with invariants [7, 8], as well as AspectJ, to prematurely sequentialize the maintenance action – either as a before-method, after-method, or around-method. Conceptually, the maintenance action is parallel to the disruptive action so that the invariant is always observed to hold in all states. The sequentialization of the parallel composition should be treated as an opportunity for machine optimization, not as additional information that must be prescribed by a programmer. However, the sequentialization should take care that no (external) process that depends on the invariant could observe the state between the two actions and notice that the invariant is (temporarily) violated. One technique for assuring that no observation of the intermittent violation can be made is to lock the relevant variables while the maintenance is being performed.

4 More Examples

4.1 Model-View Consistency Maintenance

The classic model-view problem is to maintain consistency between a data model and various graphical views when the program and/or user can change any of them. That is, whenever the program changes the data model, the graphical views should be updated to maintain consistency, and conversely, if the user changes one graphical view interactively, then the data model and the other views must be updated to reflect the change.

Problem: Maintain consistency between a data model and its various views. For simplicity we focus on eager evaluation – maintaining consistency with every change to data models or views.

Domain Theory: At any particular time there are a set of data models of type $Model$. Each model has one or more graphical views of type $View$. Views have an attribute $ModelOf : Model$ that gives the unique model that they display (written $vw.ModelOf$ for view vw). For simplicity, we assume that the data content of $mod : Model$ is given by an attribute $MValue : Model \rightarrow Value$, and, similarly, the data content of a view is given by $Value : View \rightarrow Value$ for $View$. Although we use equality between these values to express consistency, in practical situations, a more complex predicate is needed.

Invariant: $\Box\, \forall(md, vw)\ md = vw.ModelOf \implies vw.Value = md.MValue$

Disruptive Actions: There are two classes of disruptive actions: changes to a view or changes to a model.

Specification and derivation of maintenance code: for each disruptive action act, generate a specification for an action that jointly achieves the effect of act and maintains the invariant:

$$\begin{array}{lrcl} \textbf{assume}: & vw.Value & = & md.MValue \wedge precondition(act) \\ \textbf{achieve}: & vw'.Value & = & md'.MValue \wedge postcondition(act) \end{array}$$

For example, an action act that updates a view

$$vw.Value := expr$$

results in the maintenance specification

$$\begin{array}{lrcl} \textbf{assume}: & vw.Value & = & md.MValue \\ \textbf{achieve}: & vw'.Value & = & md'.MValue \wedge vw'.Value = expr \end{array}$$

which is satisfied by the concurrent assignment

$$vw.Value := expr \,||\, md.MValue := expr$$

Similar code is derived for other cases.

4.2 Procedure Calls and Dynamic Context

This exercise treats procedure calls and the reification of dynamic procedure call context.

Problem: Maintain a global that flags when a Sort procedure is executing.

Reification: This problem requires that we reify and maintain the call stack, analogously to the way that history is maintained in *hist*. To reify the call stack, it is necessary to elaborate the model of behavior presented in Section 2. A call to procedure P, $s_0 \xrightarrow{x:=P(x)} s_1$, can be elaborated to a sub-behavior

$$s_0 \xrightarrow{eval\ args} s_{00} \xrightarrow[parms:=argvals]{enter\ P} s_{01} \xrightarrow{execute\ P} s_{02} \xrightarrow[x:=result]{exit\ P} s_1$$

With this elaboration, it is straightforward to maintain a call stack variable *cs* with operators $Initialize$, $Push$, and Pop:

$$s_0 \xrightarrow{eval\ args} s_{00} \xrightarrow[cs:=Push(cs,\langle P,argvals\rangle)]{enter\ P} s_{01} \xrightarrow{execute\ P} s_{02} \xrightarrow[cs:=Pop(cs)]{exit\ P} s_1$$

Procedural languages abstract away these details so a static analyzer must take this finer-grain model into account when appropriate.

Domain Theory: The boolean variable *sorting*? is to be true exactly when a call to *Sort* is on the call stack *cs*. In the invariant, we use a (meta)predicate $pcall?(act, f)$ that is true exactly when action *act* is a procedure call to f.

Invariant: $\Box\ sorting? = \exists(call)(call \in cs\ \wedge\ pcall?(call, Sort))$

Incrementally maintaining a boolean value is difficult, and a standard technique is to transform a quantified expression into an equivalent set-theoretic form that is easier to maintain [5]:

$\Box\ sorting? = size(\{call \mid call \in cs\ \wedge\ pcall?(call, Sort)\}) > 0$

and introduce a second invariant:

$\Box\ sortcnt = size(\{call \mid call \in cs\ \wedge\ pcall?(call, Sort)\})$

By maintaining *sortcnt*, we can replace *sorting*? by $sortcnt > 0$ everywhere it occurs.

Disruptive Actions: The static analyzer seeks actions that change the dependent variable *cs*; i.e. pushes and pops of that call stack *cs* that satisfy $pcall?(call, Sort)$.

Specification and derivation of maintenance code: There are three basic cases that can arise: *Initialize*, $Push$, and Pop operations. For a push operation of the form

$$cs := Push(cs, \langle Sort, \_ \rangle)$$

the maintenance specification is

assume: $sortcnt = size(\{call \mid call \in cs \wedge pcall?(call, Sort)\})$
achieve: $sortcnt' = size(\{call \mid call \in cs' \wedge pcall?(call, Sort)\})$
$\wedge\ cs' = Push(cs, \langle Sort, \_ \rangle)$

which an easy calculation shows to be satisfied by the concurrent assignment

$$cs := Push(cs, \langle Sort, \_ \rangle) \mid\mid sortcnt := sortcnt + 1$$

For a pop operation of the form $cs := Pop(cs)$ where $top(cs) = \langle Sort, \_ \rangle$, the maintenance specification is

assume: $top(cs) = \langle Sort, \_ \rangle$
$\wedge\ sortcnt = size(\{call \mid call \in cs \wedge pcall?(call, Sort)\})$
achieve: $sortcnt' = size(\{call \mid call \in cs' \wedge pcall?(call, Sort)\})$
$\wedge\ cs' = Pop(cs)$

which is satisfied by the concurrent assignment

$$cs := Pop(cs) \mid\mid sortcnt := sortcnt - 1$$

The concurrent formulation of the maintenance code can be implemented by sequentializing the *sortcnt* updates into the body of the procedure, just after entry and just before return.

An initialization operation on *cs* will cause *sortcnt* to be set to zero.

4.3 Counting Swaps in a Sort Routine

This problem builds on the previous problem and illustrates the execution of advice within dynamic contexts, a key feature of AspectJ.

Problem: Count the number of calls to a *swap* procedure that are invoked during the execution of a sort procedure *Sort*.

Domain Theory: As in the previous problem, let *cs* be the reified call stack, with operators *Initialize*, *Push*, and *Pop*.

Invariant: The invariant uses a sequence comprehension notation, so that *swpcnt* is the length of a sequence of actions satisfying various properties. Also, recall that the notation $s_0.cs$ refers to the value of variable *cs* in state s_0.

$$\Box\ swpcnt = length([\ act \mid \langle s_0, act, s_1\rangle \in hist\ \wedge\ pcall?(act, swap) \\ \wedge\ \exists(pc)(pc \in s_0.cs\ \wedge\ pcall?(pc, Sort))])$$

Disruptive Actions: The dependent variable is *hist*. It is easy to statically analyze for $pcall?(act, swap)$. Let's assume that it is not statically determinable whether a particular call to *swap* occurs within the dynamic context of a call to *Sort*. To proceed, we extract the subexpression that cannot be checked statically and formulate an invariant for it:

$$\Box\ sorting? = \exists(call)(call \in cs\ \wedge\ pcall?(call, Sort))$$

Using the result in the previous example allows a simpler formulation for the *swpcnt* invariant:

$$\Box\ swpcnt = length([\ act \mid \langle s_0, act, s_1\rangle \in hist\ \wedge\ pcall?(act, swap) \\ \wedge\ s_0.sortcnt > 0])$$

Specification and derivation of maintenance code: Any call to *swap* is a potentially disruptive action. The following specification jointly achieves the effect of *act* and maintains the invariant:

assume: $hist' = hist :: \langle st_0, act_0, st_1\rangle$
$\wedge\ pcall?(act_0, swap)$
$\wedge\ swpcnt = length([\ act \mid \langle s_0, act, s_1\rangle \in hist$
$\wedge\ pcall?(act, swap)$
$\wedge\ s_0.sortcnt > 0])$
$\wedge\ precondition(act_0)$

achieve: $swpcnt' = length([\ act \mid \langle s_0, act, s_1\rangle \in hist'$
$\wedge\ pcall?(act, swap)\ \wedge\ s_0.sortcnt > 0])$
$\wedge\ postcondition(act_0)$

The postcondition can be simplified as follows (where changes are underlined):

$$swpcnt' = length([\ act \mid \langle s_0, act, s_1\rangle \in hist' \\ \wedge\ pcall?(act, swap)\ \wedge\ s_0.sortcnt > 0])$$

$\equiv$ {using the assumption about *hist'*}

$$swpcnt' = length([\ act \mid \langle s_0, act, s_1\rangle \in \underline{hist :: \langle st_0, act_0, st_1\rangle} \\ \wedge\ pcall?(act, swap)\ \wedge\ s_0.sortcnt > 0])$$

$\equiv$ {distributing $\in$ over :: }

$$swpcnt' = length([\ act \mid \underline{(\langle s_0, act, s_1\rangle \in hist\ \vee\ \langle s_0, act, s_1\rangle = \langle st_0, act_0, st_1\rangle)} \\ \wedge\ pcall?(act, swap)\ \wedge\ s_0.sortcnt > 0])$$

$\equiv$ {driving $\vee$ outward through $\wedge$, sequence-former, and *length*}

$$swpcnt' = \underline{length([\ act \mid \langle s_0, act, s_1\rangle \in hist}$$

$$+ \, length([\, act \mid \langle s_0, act, s_1 \rangle \;=\; \langle st_0, act_0, st_1 \rangle \;\wedge\; pcall?(act, swap) \;\wedge\; s_0.sortcnt > 0 \,])$$

$\equiv$ {using assumption about *swpcnt*, distribute equality in sequence-former}

$$swpcnt' \;=\; swpcnt \;+\; length([\, act_0 \mid pcall?(act_0, swap) \;\wedge\; st_0.sortcnt > 0 \,])$$

$\equiv$ {using assumption about act_0, and simplifying}

$$swpcnt' \;=\; swpcnt \;+\; length([\, act_0 \mid st_0.sortcnt > 0 \,])$$

$\equiv$ {using using independence of act_0 from the sequence-former predicate}

$$swpcnt' \;=\; swpcnt \;+\; (if\ st_0.sortcnt > 0\ then\ length([\, act_0 \mid true])\ else\ length([\, act_0 \mid false]))$$

$\equiv$ {simplifying}

$$swpcnt' \;=\; swpcnt \;+\; (if\ st_0.sortcnt > 0\ then\ 1\ else\ 0).$$

Consequently, the maintenance specification is satisfied by the parallel statement

$$act_0 \;||\; swpcnt := \;\; swpcnt \;+\; (if\ sortcnt > 0\ then\ 1\ else\ 0).$$

Note that a residue of the invariant appears in the maintenance code. The test $sortcnt > 0$ could not be decided statically, so it falls through as a runtime test.

4.4 Maintaining the Length of a List

This example does not require the reification of an extra-computational entity. It is presented as an example of our technique that cannot currently be treated in AspectJ because it is handled at the assignment level, rather than method level.

Problem: Maintain the length of a list ℓ.

Domain Theory: The list data type includes constructors (*nil*, *append*, *concat*), selectors (*first*, *rest*), *deleteElt*, as well as a length function and other operators.

Invariant: $\Box\, llength = length(\ell)$

Disruptive Actions: Any action that changes ℓ may disrupt the invariant.

Specification and Derivation of Maintenance Code: for each disruptive action *act*, generate a specification for an action that jointly achieves the effect of *act* and maintains the invariant. For example, an action *act* that appends an element onto ℓ results in the maintenance specification

assume: $llength = length(\ell) \wedge true$
achieve: $llength' = length(\ell') \wedge \ell' = \ell :: elt$

from which one can easily calculate the satisfying concurrent assignment

$$\ell := \ell :: elt \;||\; llength := llength + 1$$

Other maintenance actions include: when the list ℓ is created, then the variable *llength* is also created; when ℓ is set to *nil*, then *llength* is set to 0; etc.

Notice that in the examples of this section, the generated maintenance code is highly specific to the disruptive action. We start with a single invariant, but in response to its possible violations, we generate a variety of maintenance codes. Programming language approaches to AOSD would have to provide this variety by potentially complex case analysis in the runtime aspect code.

5 Issues

This work may develop in a number of directions, some of which are discussed below.

- *Implementation* – We anticipate implementing the techniques of this paper in our behavioral extension [9] of the Specware system [10]. The calculations for simplifying the maintenance specifications in the examples are comparable in difficulty to those that were performed routinely and automatically in KIDS [11]. However, the simplifier requires the presence of an adequate inference-oriented theory of the language, datatypes, and application domain. As can be seen from the examples, most of the theorems needed are in the form of distributivity laws.

  In general, the problem of synthesizing code from pre/post-conditions is not decidable. However, two factors help to achieve tractability. First, note that the synthesis problem here is highly structured and incremental in nature – the goal is to reestablish an invariant that has just been perturbed by a given action. Second, synthesis can be made tractable by suitable restrictions on the language/logic employed. For example, in Paige's RAPT system, invariants and disruptive actions were restricted to finite-set-theoretic operations from the SETL language, and the corresponding maintenance code could be generated by table lookup.

- *Granularity of Maintenance Code* – It may be convenient to treat a code block or a procedure/method as a single action for purposes of invariant maintenance. The main issue is that no (external) process that depends on the invariant could observe a state in which the invariant is violated. This notion suggests that the possibility of designing a static analyzer

to check both (i) change of dependent variables, and (ii) the largest enclosing scope that is unobservable externally. An advantage of using a larger grain for maintenance is the performance advantage of bundling many changes at once, rather than eagerly updating at every dependent-variable-change. This is particularly advantageous when the update is relatively expensive.

A good example arises in the model-view-consistency problem (Section 4.1). If the model is changing quickly (say, due to rapid real-time data feed), then user may not want screen updates at the true change frequency. Instead, aesthetics and user needs may dictate that a time granularity be imposed, e.g. no more frequently than every 100 sec. Where is this granularity-of-observation specified? In open system design, this is a design decision and it can be made precise by means of a specification of environment assumptions. In our behavioral specification formalism, called especs [9], environment assumptions are modeled as parametric behaviors. Under composition, the parametric behavior of a component must be implemented by the rest of the system. In the model-view-consistency problem, the parametric behavior (environmental assumptions) might be that the user observes the display screen with a frequency of at most 0.01Hz. This requirement is satisfied by code that reestablishes the invariant at least once every 100sec.

- *Constraint Maintenance: Maximization versus Invariance* – Sometimes a cross-cutting feature may not have the form of an invariant for practical reasons. Consider, for example, the quality of service offered by a wireless communications substrate. Ideally, full capacity service is provided invariantly. However, physical devices are inherently more or less unreliable. There are at least two characterizations of the constraint maintenance that make sense in this situation:

  1. *Maximize the uptime of the service* – That is, maximize the amount of time that a prescribed level of service is provided. Design-time maintenance might involve composing a fault-adaptive scheme to improve uptime.
  2. *Maximize the provided bandwidth*– That is, continually make adjustments that provide maximal bandwidth given the circumstances.

- *Enforcing Behavioral Policies* – This paper focuses on cross-cutting concerns that can be specified as invariants. Behavioral invariants can be equivalently expressed as single-mode automata with an axiom at the mode. It is natural to consider cross-cutting concerns that are specified by more complex automata and their corresponding temporal logic formulas. As mentioned earlier, some security policies disallow certain behavior patterns, as opposed to individual run-time events (see for example [4]). One intriguing direction is to consider generalizing the generation techniques of this paper to classes of policy automata.

- *Maintaining Interacting Constraints* – Many application areas, including active databases with consistency constraints and combinatorial optimization problems with constraint propagation, have the characteristic that a single change ($x := e$) can stimulate extensive iteration until quiescence (a fixpoint) is reached. In terms of this paper, several invariants can refer to the same variables and their maintenance can interfere with each other's truth. That is, a change to maintain one constraint may cause the violation of another.

A sufficient condition that maintaining such a set of constraints leads to a fixpoint may be found in [12] – constraints over a finite semilattice that are definite (a generalized Horn-clause form $x \sqsupseteq A(x)$ where x is a variable over the semilattice and A is monotone) can be solved in linear time. Using essentially the same theory, in [6, 13] we describe the process of automatically generating a customized constraint solver for definite constraints. The resulting solving process is an iterative refinement of variable values in the semilattice.

This context leads to a generalization of the formalism of this paper when (1) changes to certain variables can be treated as decreasing in a semilattice, and (2) constraints are definite. Then, a disruptive action ($x := e$) has postcondition ($x' \sqsupseteq e$) rather than the stronger ($x' = e$), and all constraint maintenance is downward in the semilattice, until a fixpoint is reached.

- *Comparison with AspectJ* – We conjecture that many aspects in AspectJ can be expressed as invariants, and that their effect can be achieved by means of the general process of this paper. However, the around advice in AspectJ allows the replacement of a method call by arbitrary code, changing its semantics. Our approach is restricted to maintenance that refines existing actions, so it is not complete with respect to AspectJ. On the other hand several of the examples in this paper cannot be carried out in AspectJ, so the two are expressively incomparable.

6 Related Work

The generative techniques in this paper derive from transformational work on incremental computation, especially Paige's Finite Differencing transformation [5, 7]. Finite Differencing, as implemented in the RAPTS system, automatically maintains invariants of the form $c = f(x)$ where c is a fresh variable, x is a vector of program variables, and f is a composite of set-theoretic programming language operations. Maintenance code is generated by table lookup. In the KIDS system [11], we extended Finite Differencing by (1) allowing the maintenance of both language- and user-defined terms, and (2) using automatic simplifiers to calculate maintenance code at design-time. The functional language setting in KIDS naturally reveals the concurrency of disruptive code and maintenance updates.

As in Finite Differencing, other approaches to programming with invariants (e.g. [8]) work exclusively with state variables. This paper introduces the notion of reifying extra-computational information, enabling the expression of system-level cross-cutting features as invariants.

Besides AspectJ, other approaches to AOSD may also be treatable by a generative approach. The Demeter system [14] emphasizes path expressions to provide reusable/context-independent access to data. A typical example is to enumerate all objects that inherit from a superclass *Employee* that have a *Salary* attribute regardless of the inheritance chain from the superclass. Reification of the class graph is essential and is responsible for the support provided for run-time binding of data accessors. HyperJ [15] emphasizes a symmetric notion of aspect, in contrast to the asymmetric approach of AspectJ. The focus of HyperJ is on a cross-product-like composition of features between classes, obtaining a multiplicative generation of code. A foundation for this symmetric composition may be found in the colimit operation of Specware [9, 10].

7 Concluding Remarks

Aspect-Oriented Software Development aims to support a more modular approach to programming, with a special focus on cross-cutting concerns. This paper explores techniques for specifying cross-cutting concerns as temporal logic invariants, and generating the code necessary to maintain them. The reification of extra-computational entities helps in expressing many cross-cutting concerns.

Our invariants provide an abstract yet precise, semantic characterization of cross-cutting concerns. The abstraction should aid in clarifying the intention of a concern and promote stability under evolution. The precise semantics means that the generation of maintenance code can be performed mechanically, with assurance that the result meets intentions.

The generally accepted semantics of AspectJ is based on call-stack reification [16], suggesting that AspectJ cross-cutting concerns can be characterized as actions to take about method calls in a specified dynamic context. Our approach lifts to a more general perspective: what kinds of cross-cutting concerns can be addressed when arbitrary extra-computational information is reified.

This work advocates a design process that focuses on generating a normal-case program from high-level models or specifications, followed by the generation and insertion of extensions to implement various cross-cutting concerns. Code structure simplifies to a clean natural decomposition of the basic business logic together with system-level invariants that specify cross-cutting concerns. The improved modularity should help to ease the cost and effort of development and evolution.

Acknowledgments: Thanks to Cordell Green, Gregor Kiczales, and Kevin Sullivan for discussions of this work. Thanks to Lambert Meertens, Stephen Westfold, and the GPCE reviewers for useful comments on the text.

References

[1] Aspect-Oriented Software Development, www.aosd.net.

[2] Elrad, T., Filman, R., Bader, A., eds.: Communications of the ACM– Special Issue on Aspect-Oriented Programming. Volume 44(10). (2001)

[3] Kiczales, G., et al.: An Overview of AspectJ. In: Proc. ECOOP, LNCS 2072, Springer-Verlag. (2001) 327–353

[4] Erlingsson, U., Schneider, F.: SASI enforcement of security policies: A retrospective. In: Proceedings of the New Security Paradigms Workshop, Ontario, Canada (1999)

[5] Paige, R., Koenig, S.: Finite differencing of computable expressions. ACM Transactions on Programming Languages and Systems **4** (1982) 402–454

[6] Westfold, S., Smith, D.: Synthesis of efficient constraint satisfaction programs. Knowledge Engineering Review **16** (2001) 69–84 (Special Issue on AI and OR).

[7] Paige, R.: Programming with invariants. IEEE Software **3** (1986) 56–69

[8] Deng, X., Dwyer, M., Hatcliff, J., Mizuno, M.: Invariant-based specification, synthesis and verification of synchronization in concurrent programs. In: Proceedings of the 24th International Conference on Software Engineering. (May 2002)

[9] Pavlovic, D., Smith, D.R.: Evolving specifications. Technical report, Kestrel Institute (2004)

[10] Kestrel Institute: Specware System and documentation. (2003) http://www.specware.org/.

[11] Smith, D.R.: KIDS – a semi-automatic program development system. IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering **16** (1990) 1024–1043

[12] Rehof, J., Mogenson, T.: Tractable constraints in finite semilattices. Science of Computer Programming **35** (1999) 191–221

[13] Smith, D.R., Parra, E.A., Westfold, S.J.: Synthesis of planning and scheduling software. In Tate, A., ed.: Advanced Planning Technology, AAAI Press, Menlo Park (1996) 226–234

[14] Lieberherr, K., Orleans, D., Ovlinger, J.: Aspect-oriented programming with adaptive methods. CACM 44(10) (2001) 39–42

[15] Ossher, H., Tarr, P.: Using multidimensional separation of concerns to (re)shape evolving software. CACM 44(10) (2001) 43–50

[16] Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic join points in aspect-oriented programming. ACM Transactions on Programming Languages and Systems (2003)