to appear in *Automatic Program Development: a Tribute to Robert Paige*,
O. Danvy, F. Henglein, H. Mairson, and A. Pettorosi (Eds.), Springer-Verlag, 2006.

Aspects as Invariants

Douglas R. Smith
Kestrel Institute
Palo Alto, California
smith@kestrel.edu
April 7, 2006

Abstract

Aspect-Oriented Programming (AOP) offers new insights and tools for the modular development of systems with cross-cutting features. Current tool support for AOP is provided mainly in the form of code-level constructs. This paper presents a way to express cross-cutting features as logical invariants and then to generate the kind of code that is usually produced from manually written aspects. 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 invariant approach is illustrated by a variety of examples.

1 Introduction

Aspect-Oriented Programming (AOP) 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.

In this paper we focus on aspects as expressed in AspectJ [3] and recent extensions of it. AspectJ aspects can be thought of 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, perform a logging action. The runtime events are called *join points* and descriptions of join points are called *pointcuts*. One can think of pointcuts as defining a type whose elements are joinpoints. The method-like actions to apply at joinpoints are called *advice* and the process of inserting advice at code locations that satisfy a pointcut is called *weaving*. An *aspect* is a modular treatment of a crosscutting concern that is composed of pointcuts, advice, and other Java code. See [4] for an introduction and many practical examples.

Our goal is to explore how aspects can be specified more abstractly than in current languages. We focus primarily on AspectJ, the most widely used implementation of AOP, for which there is flurry of activity to extend its expressiveness and range of applicability. This paper explores the proposition that aspects can be specified as invariants, and that weaving is invariant maintenance.

AspectJ has attracted a a wide user community partly because it is well integrated with Java: aspects are written in a class-like syntax and have a semantics that closely adheres to Java semantics. Yet despite its attractiveness to programmers, some issues arise due to the operational nature of aspects:

1. *Intent* – What is the intent of an aspect? The program-like nature of an aspect often obscures it's intention. It would be desirable to have a more semantic characterization of aspects, at least as an alternate description. That is, what is the specification for which the aspect is an implementation?
2. *Pointcut Completeness* – Does a pointcut exactly characterize the intended runtime events? AspectJ pointcuts often depend on uniform conformance to syntactic/naming conventions, rather than more semantic considerations, so it may sometimes be difficult to catch all relevant joinpoints in the pointcut. For security aspects in particular, it is important not to overlook a potential joinpoint.
3. *Advice Correctness* – Does an aspect's advice correctly realize its intent?

In this paper we present a generative approach to AspectJ that addresses these issues. The key idea is that an invariant captures the intent of an aspect. Aspect weaving is then the process of maintaining the invariant by generating and inserting code fragments at appropriate locations in the base code. In this approach, a pointcut specification is derived from the invariant and it characterizes the set of code points that might disrupt the invariant. For each such disruption point in the code, a specification for maintenance code is derived from the invariant. Code that is generated from the maintenance specification corresponds to statically woven advice, and could be expressed either directly in AspectJ, or by direct generation and insertion of code into the system. By expressing an aspect as an invariant we more clearly separate its intent from its implementation/realization.

In addition to addressing the three issues listed above, the invariant maintenance approach also provides novel insights and approaches to the problems of (1) context-specialization of advice, (2) aspect interference, and (3) evolution in response to base code changes.

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 in loops 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. We conclude by revisiting the issues listed above and examining how the invariant approach provides answers and fresh insight to them.

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, written $[]$, and $append(S, a)$, written $S :: a$ for sequence S and element a, construct sequences, including behaviors. The selectors on behaviors are

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

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, $state_1.x$.

Two higher-order operators will be useful:

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

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

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

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

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 AOP 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 and modeling 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.

$$\begin{tikzcd} s_0 \arrow[r, "act_0", "hist := hist{::}\langle s_0, act_0, s_1\rangle"'] & s_1 \arrow[r, "act_1", "hist := hist{::}\langle s_1, act_1, s_2\rangle"'] & s_2 \arrow[r, "act_2", "hist := hist{::}\langle s_2, act_2, s_3\rangle"'] & s_3 \cdots \end{tikzcd}$$

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:

$$\text{Invariant:} \quad errlog \ = \ error? \triangleright action \star hist$$

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.

Establishing the Invariant

In order to correctly realize the invariant in the target code, we proceed by induction. The first step is to generate code to establish the invariant initially, by satisfying the following specification:

assume: $hist = []$
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*}

$$errlog \;=\; error? \triangleright action \; \star []$$

$\equiv$ {simplifying }

$$errlog \;=\; []$$

which is satisfied by the initialization code

$$errlog \;:=\; [].$$

More generally, when the invariant contains reified variables, the following scheme specifies code for establishing the invariant $I(x)$:

$$\begin{array}{rl} \textbf{assume}: & hist = [] \\ & \wedge \; \ldots \text{ initial values of other reified variables } \ldots \\ & \wedge \; \ldots \text{ base code preconditions } \ldots \\ \textbf{achieve}: & I(x) \end{array}$$

In Section 4.3, we give an example that does not mention reified variables. It uses a slightly different scheme for specifying the establishment of the invariant.

Specifying Disruptive Code and Deriving the Pointcut

To proceed with the inductive argument, we must maintain the invariant for all actions of the target code. Since most actions of the target code have no effect on the invariant, it is useful to focus on those actions that might disrupt the invariant. We will then generate code for maintaining the invariant in parallel with the disruptive action.

The set of all code points that might 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.

An exact characterization of the disruption points is given by

$$I(x) \neq I(x') \qquad (1)$$

That is, any action that satisfies (1) as a postcondition is a disruption point. More generally, any action that satisfies a necessary condition on (1) is a potential disruption point.

In our example, we set up the following inference task:

assume: $errlog \;=\; error? \triangleright action \star hist$
$\quad\wedge\; hist' = hist :: \langle \_, act, \_\rangle$
$\quad\wedge\; errlog' = errlog$
simplify: $(errlog \;=\; error? \triangleright action \star hist) \;\neq\; (errlog' \;=\; error? \triangleright action \star hist')$

In words, we assume that the invariant holds before an arbitrary action *act*, and that the *hist* variable is updated in parallel with *act*. Moreover, we add in a frame axiom that asserts that *act* does not change *errlog* since it is a fresh variable introduced by the invariant.

We calculate a pointcut specification as follows:

$$(errlog \;=\; error? \triangleright action \star hist) \;\neq\; (errlog' \;=\; error? \triangleright action \star hist')$$

$\equiv$ { using the frame axiom and simplifying}

$$error? \triangleright action \star hist \;\neq\; error? \triangleright action \star hist'$$

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

$$error? \triangleright action \star hist \;\neq\; error? \triangleright action \star (hist :: \langle \_, act, \_\rangle)$$

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

$$error? \triangleright action \star hist \;\neq\; error? \triangleright ((action \;\star hist) :: act)$$

$\equiv$ { distributing $error? \triangleright$ over :: }

$$\begin{aligned} error? \triangleright action \star hist \;\neq\; (&if\ \neg error?(act) \\ &then\ error? \triangleright action \star hist \\ &else\ (error? \triangleright action \;\star hist) :: act \end{aligned}$$

$\equiv$ { distributing the conditional outward }

$$\begin{aligned} &if\ \neg error?(act) \\ &\;\; then\ error? \triangleright action \star hist \;\neq\; error? \triangleright action \star hist \\ &\;\; else\ error? \triangleright action \star hist \;\neq\; (error? \triangleright action \;\star hist) :: act \end{aligned}$$

$\equiv$ { simplifying }

$$\begin{aligned} &if\ \neg error?(act) \\ &\;\; then\ false \\ &\;\; else\ true \end{aligned}$$

$\equiv$ { simplifying }

$$error?(act).$$

A static analyzer would scan the code (i.e. the abstract syntax representation of the code) looking for all actions that satisfy this derived pointcut.

More generally, the task to infer a pointcut specification is given by an instance of the following scheme:

assume: $I(x)$
$\wedge\ hist' = hist :: \langle \_, act, \_ \rangle$
$\wedge$ ... updates of other reified variables ...
$\wedge$ ... relevant frame conditions ...
simplify: $I(x) \neq I(x')$

The simplified result will typically contain a mixture of terms, some of which can be evaluated statically (i.e. on the abstract syntax of the source code) and some of which must be evaluated dynamically (i.e. on the runtime data). Since we only need a necessary condition on (1), we can weaken the derived pointcut specification by discarding those subformulas that can only be evaluated dynamically. This weakening process means that the pointcut specification may allow false positives, but, as will be seen in later examples, the generated maintenance code incorporates the relevant semantics of the discarded dynamic tests.

Specification and Derivation of Maintenance Code

To complete the induction, we must find each potentially disruptive action (using the derived pointcut specification) and then generate maintenance code to reestablish the invariant in parallel with it. 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: $errlog = error? \triangleright action \star hist$
$\wedge\ error?(act)$
$\wedge\ hist' = hist :: \langle \_, act, \_ \rangle$
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 satisfied by the simple update

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

This maintenance action is to be performed in parallel with *act*.

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 specified as

assume : $P(x) \;\wedge\; I(x)$
$\wedge\; hist' = hist :: \langle s_0, act, s_1\rangle$
$\wedge$ ...updates to other reified vars...
achieve : $Q(x, x') \;\wedge\; I(x')$

In this schematic specification we compose the aspect with the base code by means of a conjunction. Note that this specification preserves the effect of *act* while additionally reestablishing the invariant I. If it is inconsistent to achieve both, then the specification is unrealizable.

The generated code for *maint* may take the form of a parallel composition

$$act || update$$

of the actions *act* and *update*, or it may take a sequential form. Besides conceptual clarity, an advantage to treating the maintenance action as parallel to the disruptive action is that the invariant is always observed to hold in all states. Most work on programming with invariants (e.g. [7, 8]), as well as AspectJ, sequentializes the maintenance action. If the maintenance action is sequentialized, say for purposes of optimization, the generator needs to 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 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 *InitStack*, *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 predicate *pcall*?(*act*, f) that is true exactly when action *act* is a procedure call to f.

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

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]:

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

and introduce a second invariant:

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

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

Establishing the invariant: Code to establish the *sortcnt* invariant is specified as

assume: $hist = [] \ \wedge\ cs = initStack()$
achieve: $sortcnt \ = \ size(\{call \mid call \in cs \ \wedge\ pcall?(call, Sort)\})$

The postcondition can be simplified as follows:

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

$\equiv$ $\{$using the assumption about $cs\}$

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

$\equiv$ $\{$simplifying $\}$

$$sortcnt \ = \ 0$$

which is satisfied by the initialization code

$$sortcnt \ := \ 0.$$

Disruptive Actions: The following task to infer a pointcut specification assumes a frame axiom that characterizes the effect of an arbitrary base code action on the call stack variable – it either effects a *push*, a *pop*, or has no effect.

assume: $sortcnt \ = \ size(\{call \mid call \in cs \ \wedge\ pcall?(call, Sort)\})$
$\quad \wedge\ hist' = hist :: \langle \_, act, \_ \rangle$
$\quad \wedge\ (cs' = push(cs, \langle P, argvals \rangle)$
$\qquad \vee\ (cs' = pop(cs) \ \wedge\ top(cs) = \langle P, argvals \rangle)$
$\qquad \vee\ cs' = cs)$
$\quad \wedge\ sortcnt' = sortcnt$
simplify: $(sortcnt \ = \ size(\{call \mid call \in cs \ \wedge\ pcall?(call, Sort)\}))$
$\qquad \neq\ (sortcnt' \ = \ size(\{call \mid call \in cs' \ \wedge\ pcall?(call, Sort)\}))$

We calculate

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

$\equiv$ $\{$using the frame axiom for $sortcnt$ and simplifying$\}$

$$size(\{call \mid call \in cs \ \wedge\ pcall?(call, Sort)\}) \ \neq\ size(\{call \mid call \in cs' \ \wedge\ pcall?(call, Sort)\})$$

Using the disjunctive frame axiom on the call stack variable, we can proceed by cases:

$\equiv$ $\{$ Case 1: assume $cs' = push(cs, \langle P, argvals \rangle)$ $\}$

$$size(\{call \mid call \in cs \ \wedge\ pcall?(call, Sort)\}) \\ \neq\ size(\{call \mid call \in push(cs, \langle P, argvals \rangle) \ \wedge\ pcall?(call, Sort)\})$$

$\equiv$ { eliding the left-hand side and distributing call stack membership over *push* }

$$\ldots \neq size(\{call \mid (call \in cs \ \vee \ call = \langle P, argvals\rangle) \ \wedge \ pcall?(call, Sort)\})$$

$\equiv$ { distributing }

$$\ldots \neq size(\{call \mid call \in cs \ \wedge \ pcall?(call, Sort)\}) \\ + size(\{call \mid call = \langle P, argvals\rangle \ \wedge \ pcall?(call, Sort)\})$$

$\equiv$ { simplifying }

$$\ldots \neq size(\{call \mid call \in cs \ \wedge \ pcall?(call, Sort)\}) \\ + \mathit{if}\ pcall?(\langle P, argvals\rangle, Sort)\ \mathit{then}\ 1\ \mathit{else}\ 0$$

$\equiv$ { distributing the conditional outwards }

$$\mathit{if}\ pcall?(\langle P, argvals\rangle, Sort)\ \mathit{then}\ \mathit{true}\ \ \mathit{else}\ \mathit{false}$$

$\equiv$ { simplifying }

$$pcall?(\langle P, argvals\rangle, Sort).$$

The derived pointcut specification in this case is

$$cs' = push(cs, \langle P, argvals\rangle) \ \wedge \ pcall?(\langle P, argvals\rangle, Sort).$$

Continuing with the case analysis, we calculate

$\equiv$ { Case 2: assume $(cs' = pop(cs) \ \wedge \ top(cs) = \langle P, argvals\rangle)$
which implies $(cs = push(cs', \langle P, argvals\rangle)$ }

$$size(\{call \mid call \in push(cs', \langle P, argvals\rangle) \ \wedge \ pcall?(call, Sort)\}) \\ \neq size(\{call \mid call \in cs' \ \wedge \ pcall?(call, Sort)\})$$

$\equiv$ { using similar reasoning to previous case}

$$pcall?(\langle P, argvals\rangle, Sort).$$

The derived semantic pointcut in this case is

$$cs' = pop(cs) \ \wedge \ top(cs) = \langle P, argvals\rangle \ \wedge \ pcall?(\langle P, argvals\rangle, Sort).$$

And the final step in the case analysis is

$\equiv$ { Case 3: assume $cs' = cs$ }

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

$\neq \ size(\{call \mid call \in cs' \ \wedge \ pcall?(call, Sort)\})$

$\equiv$ { using the assumption, and simplifying}

false.

Combining the case assumptions with their derived pointcut specifications, we obtain

$$\begin{array}{l} \quad cs' = push(cs, \langle P, argvals\rangle) \ \wedge \ pcall?(\langle P, argvals\rangle, Sort) \\ \vee \\ \quad cs' = pop(cs) \ \wedge \ top(cs) = \langle P, argvals\rangle \ \wedge \ pcall?(\langle P, argvals\rangle, Sort) \\ \vee \\ \quad cs' = cs \ \wedge \ false. \end{array}$$

or simply

$$\begin{array}{l} \quad cs' = push(cs, \langle P, argvals\rangle) \ \wedge \ pcall?(\langle P, argvals\rangle, Sort) \\ \vee \\ \quad cs' = pop(cs) \ \wedge \ top(cs) = \langle P, argvals\rangle \ \wedge \ pcall?(\langle P, argvals\rangle, Sort). \end{array}$$

which specifies entrances and exits of calls to *Sort* respectively.

Specification and derivation of maintenance code: The pointcut specification gives rise to two cases: *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: $cs' = push(cs, \langle Sort, argvals\rangle)$
$\qquad \wedge \ sortcnt' \ = \ size(\{call \mid call \in cs' \ \wedge \ pcall?(call, Sort)\})$

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

$$cs \ := \ Push(cs, \langle Sort, argvals\rangle) \ || \ sortcnt \ := \ sortcnt + 1$$

on entrance to procedure *Sort*.

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

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

which is satisfied by the concurrent assignment

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

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.

4.2 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 *InitStack*, *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.

$$\begin{aligned} swpcnt = length([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in hist \ \wedge\ pcall?(act_0, swap) \\ \wedge\ \exists(pc)(pc \in st_0.cs\ \wedge\ pcall?(pc, Sort))]) \end{aligned}$$

or, more simply, using the invariant from the previous example

$$swpcnt = length([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in hist \ \wedge\ pcall?(act_0, swap)\ \wedge\ st_0.sortcnt > 0\ ])$$

Establishing the invariant: Code to establish the *sortcnt* invariant is specified as

assume: $hist = [] \ \wedge\ cs = initStack()$
achieve: $swpcnt = length([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in hist \ \wedge\ pcall?(act_0, swap)\ \wedge\ st_0.sortcnt > 0\ ])$.

The postcondition can be simplified as follows:

$$\begin{aligned} & swpcnt \ = \ length([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in hist \ \wedge\ \ldots]) \\ \equiv\quad & \{ \text{ using the assumption about } hist \ \} \\ & swpcnt \ = \ length([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in [] \ \wedge\ \ldots]) \\ \equiv\quad & \{ \text{ simplifying } \} \\ & swpcnt \ = \ 0 \end{aligned}$$

which is satisfied by the initialization code

$$swpcnt \ := \ 0.$$

Disruptive Actions: The inference task to infer a pointcut is specified as follows

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

simplify: $(\mathit{swpcnt} = \mathit{length}([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in \mathit{hist} \ \wedge\ \mathit{pcall?}(act_0, \mathit{swap})$
$\qquad\qquad\qquad\qquad \wedge\ st_0.\mathit{sortcnt} > 0]))$
$\neq\ (\mathit{swpcnt}' = \mathit{length}([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in \mathit{hist}' \ \wedge\ \mathit{pcall?}(act_0, \mathit{swap})$
$\qquad\qquad\qquad\qquad \wedge\ st_0.\mathit{sortcnt} > 0]))$

We calculate a pointcut specification as follows:

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

$\equiv$ { using the frame axiom, and simplifying }

$$\mathit{length}([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in \mathit{hist} \ \wedge\ \mathit{pcall?}(act_0, \mathit{swap}) \ \wedge\ st_0.\mathit{sortcnt} > 0])$$
$$\neq\ \mathit{length}([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in \mathit{hist}' \ \wedge\ \mathit{pcall?}(act_0, \mathit{swap}) \ \wedge\ st_0.\mathit{sortcnt} > 0])$$

$\equiv$ { eliding the left hand side and using the assumption about *hist'* }

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

$\equiv$ { distributing }

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

$\equiv$ { distributing the disjunction outwards }

$$\ldots\ \neq\ \mathit{length}([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in \mathit{hist} \ \wedge\ \mathit{pcall?}(act_0, \mathit{swap}) \wedge\ st_0.\mathit{sortcnt} > 0])$$
$$+\ \mathit{length}([\ act_0 \mid \langle st_0, act_0, st_1\rangle = \langle s_0, act, s_1\rangle \wedge \mathit{pcall?}(act_0, \mathit{swap}) \ \wedge\ st_0.\mathit{sortcnt} > 0])$$

$\equiv$ { distributing the equality in the second addend }

$$\ldots\ \neq\ \mathit{length}([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in \mathit{hist} \ \wedge\ \mathit{pcall?}(act_0, \mathit{swap}) \ \wedge\ st_0.\mathit{sortcnt} > 0])$$
$$+\ \mathit{length}([\ act \mid \mathit{pcall?}(act, \mathit{swap}) \ \wedge\ s_0.\mathit{sortcnt} > 0])$$

$\equiv$ { simplifying }

$$\ldots\ \neq\ \mathit{length}([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in \mathit{hist} \ \wedge\ \mathit{pcall?}(act_0, \mathit{swap}) \ \wedge\ st_0.\mathit{sortcnt} > 0])$$
$$+\ \mathit{if}\ \mathit{pcall?}(act, \mathit{swap}) \ \wedge\ s_0.\mathit{sortcnt} > 0\ \mathit{then}\ 1\ \mathit{else}\ 0$$

$\equiv$ { distributing the conditional outwards and simplifying }

$$\textit{if } pcall?(act, swap) \ \wedge\ s_0.sortcnt > 0 \textit{ then true else false}$$

$\equiv$ { simplifying }

$$pcall?(act, swap) \ \wedge\ s_0.sortcnt > 0.$$

The derived pointcut specification is

$$pcall?(act, swap) \ \wedge\ sortcnt > 0.$$

Note that the second conjunct is not statically determinable in general, so we weaken the pointcut to $pcall?(act, swap)$ (recall that we only need a necessary condition on the disruption of the invariant). When we derive the maintenance code below, the extra condition $sortcnt > 0$ will show up as a runtime test.

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

The second conjunct of the postcondition can be simplified as follows:

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

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

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

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

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

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

$$swpcnt' \ =\ length([\ act_0 \mid \langle st_0, act_0, st_1\rangle \in hist$$

$$\wedge\ pcall?(act_0, swap)\ \wedge\ ts_0.sortcnt > 0\ ])$$
$$+\ length([\ act_0 \mid \langle st_0, act_0, st_1\rangle\ =\ \langle s_0, act, s_1\rangle$$
$$\wedge\ pcall?(act_0, swap)\ \wedge\ st_0.sortcnt > 0\ ])$$

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

$$swpcnt'\ =\ swpcnt\ +\ length([\ act \mid pcall?(act, swap)\ \wedge\ s_0.sortcnt > 0\ ])$$

$\equiv$ { using assumption about *act*, and simplifying}

$$swpcnt'\ =\ swpcnt\ +\ length([\ act \mid s_0.sortcnt > 0\ ])$$

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

$$swpcnt'\ =\ swpcnt\ +\ (if\ s_0.sortcnt > 0\ then\ length([\ act \mid true])\ \ else\ length([\ act \mid false])$$

$\equiv$ { simplifying }

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

$\equiv$ { pulling the conditional outward and simplifying }

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

Consequently, the maintenance specification is satisfied by the parallel statement

$$act \mid\mid if\ sortcnt > 0\ \ then\ swpcnt :=\ swpcnt\ +\ 1.$$

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.

4.3 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 at the method-call 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: $llength = length(\ell)$

Disruptive Actions: The derivation of the pointcut specification results in $\ell \neq \ell'$; i.e. any action that changes ℓ may disrupt the invariant. Static analysis looks for any action that changes ℓ, such as assignments to ℓ.

Establishing the invariant: The invariants that were treated in previous examples all refered to reified variables, and consequently, the code to establish them belonged to the outermost program initialization phase. This example refers only to program variables (ℓ in particular), so its initialization code belongs to the scope of those variables. For this example, with one dependent variable, it is easy for static analysis to locate its initialization action, say *linit*. Then the code to establishment of the invariant initially is specified as

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

That is, we wish to establish the invariant concurrently with the initialization of its dependent variable. To be concrete, if the static analyzer finds the initialization code $\ell \ := \ nil$ then the appropriate instance of the above scheme results in the concurrent code

$$\ell \ := \ nil \ || \ llength \ := \ 0$$

which establishes the invariant.

Specification and Derivation of Maintenance Code: For each potentially disruptive action *act*, we generate a specification for an action that jointly achieves the effect of *act* and maintains the invariant. For example, suppose that the pointcut specification matches an assignment

$$\ell \ := \ \ell :: elt$$

an action *act* that appends an element onto ℓ results in the maintenance specification

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

from which one can easily calculate the satisfying concurrent assignment

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

For other actions that change ℓ, we create the corresponding maintenance code specifications, and then generate code. Note that each change to ℓ can result in code that is completely different from other maintenance actions for ℓ.

4.4 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.

Note that this example has no newly introduced variables as in previous examples. The nature of the problem is to enforce a new constraint on existing variables.

Problem: Maintain consistency between a data model $md : Model$ and a graphical view $vw : View$. Generalizing the example to multiple models with multiple views is discussed at the end of this section.

Domain Theory: Assume that the data content of $md : Model$ is given by an attribute $mValue : Model \to Value$ for some type $Value$, and, similarly, the data content of a view is given by $vValue : View \to Value$ for $View$. Although equality is used between these values to express consistency, in practical situations, a more complex predicate is needed.

Invariant: $vw.vValue = md.mValue$

Disruptive Actions: We assume that the only changes that can be made to a model are via a call to its *update* method; similarly for views. Formally, the third assumption below is a frame axiom asserting that a system action either (i) leaves a model unchanged, or (ii) is a call to the model *update* method. The fourth assumption states the analogous constraint on views.

The task to infer a pointcut is specified as follows

assume: $s_0.vw.vValue = s_0.md.mValue$
$\wedge\ hist' = hist :: \langle s_0, act, s_1\rangle$
$\wedge\ (s_0.md = s_1.md\ \vee\ pcall?(act, md.update))$
$\wedge\ (s_0.vw = s_1.vw\ \vee\ pcall?(act, vw.update))$
simplify: $(s_0.vw.vValue = s_0.md.mValue)\ \neq\ (s_1.vw.vValue = s_1.md.mValue)$

We calculate a pointcut specification as follows:

$$(s_0.vw.vValue = s_0.md.mValue)\ \neq\ (s_1.vw.vValue = s_1.md.mValue)$$

$\equiv$ { using the assumption that the LHS holds and simplifying}

$$s_1.vw.vValue \neq s_1.md.mValue.$$

Using the disjunctive frame axiom on models, we can proceed by cases:

$\equiv$ { Case 1: assume $s_0.md = s_1.md$ }

$$s_1.vw.vValue \neq s_0.md.mValue$$

$\equiv$ { using the assumption that the invariant holds }

$s_1.vw.vValue \neq s_0.vw.mValue.$

Now, using the disjunctive frame axioms on views, we proceed by cases:

$\equiv$ { Case 1.1: assume $s_0.vw = s_1.vw$ }

false.

$\equiv$ { Case 1.2: assume $\mathit{pcall?}(act, vw.update)$ }

$s_1.vw.vValue \neq s_0.vw.mValue.$

Finally, popping up a level and proceeding with the case analysis:

$\equiv$ { Case 2: assume $\mathit{pcall?}(act, md.update)$ }

$s_1.vw.vValue \neq s_1.md.mValue.$

Combining the case assumptions with their derived pointcut specifications, we obtain

$$\begin{array}{l} \quad s_1.vw.vValue \neq s_1.md.mValue \;\wedge\; \mathit{pcall?}(act, md.update) \\ \vee \\ \quad s_1.vw.vValue \neq s_0.vw.mValue \;\wedge\; \mathit{pcall?}(act, vw.update) \end{array}$$

which specifies calls to update either the model or the view.

Specification and derivation of maintenance code: If action *act* has the form *md.update*(*newval*), then we generate following the maintenance specification

assume: $s_0.vw.vValue = s_0.md.mValue$
achieve: $s_1.vw.vValue = s_1.md.mValue$
$\wedge\ s_1.md.mValue = newval$

which is satisfied by the concurrent command

$$md.update(newval) \;||\; vw.update(newval)$$

The same code is derived when action *act* has the form *vw.update*(*newval*).

In general, one would like to maintain consistency between a dynamic collection of data models and their corresponding views. To specify this would require quantifying over all currently allocated models and views, which in turn requires reifying the heap. It also requires taking into account the methods for creating, destroying, and associating models and views.

5 Remarks

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 inference tasks in the examples are comparable in difficulty to those that were performed routinely and automatically in KIDS [11]. However, automated deduction requires the presence of an adequate inference-oriented theory of the language, data types, 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 [5], 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 static analysis could be used to check both (i) potential disruption points of the invariant, and (ii) the largest enclosing scope of dependent variables 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.

- *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 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-node automata with an axiom at the node. 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 [12]). It is natural to consider generalizing the techniques of this paper to classes of policy automata. In recent work we developed a behavioral notion of pointcut, using automata to specify behavioral context for the application of advice [13]. Independently, several other research groups have been developing similar concepts [14, 15, 16, 17].

- *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 may have overlapping dependent variables and consequently 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 [18]. 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, 19] 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.

  In a similar spirit, JMangler [20] implements a capability to iterate class transformations at class load-time when they have mutual dependencies. Under conditions on the transformations that satisfy the conditions above, the iteration converges to a fixpoint and the result guarantees that all transformations are fully applied.

- *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 use of the around advice in AspectJ allows the replacement of a method call by arbitrary code, changing its semantics (for example, consider the aspect that replaces every call to method m by advice that throws an exception). 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.

  In the long run, it may be that only semantics-preserving aspects will be embraced in practice. If aspects are allowed to modify the behavior of code, then locality of program semantics is destroyed. This undercuts the potential for improved understandability of code due to the increased modularity that is the hallmark of aspects.

6 Recapitulation and Comparison with Related Work

We now summarize how the invariant maintenance approach treats the issues raised in the Introduction.

1. *What is the intention of an aspect?* – The intention of an aspect is expressed by an invariant property of state. The invariant can be thought of as a formal specification of an aspect.

2. *Is the pointcut complete?* – Does a pointcut express exactly the set of the intended runtime events? The invariant maintenance approach characterizes the joinpoints semantically, as those actions that could possibly disrupt the invariant. It thus does not require adherence to naming conventions. The pointcut is derived as a necessary condition on the violation of the invariant by a system action. Static analysis is then guaranteed to identify a superset of code locations that, at runtime, give rise to a violation of the invariant.

   The derived pointcut is a specification of an AspectJ-style pointcut. To show the implementation relation between them, one would need to link the method calls and other joinpoints of the AspectJ pointcut to their semantic description (via pre/post-conditions) to show that they satisfy the derived pointcut specification.

3. *Is the advice correct?* – Does an aspect correctly implement its intention? For each potentially disruptive action, our approach generates an action-specific specification, which, if realized by synthesis, maintains the invariant while still accomplishing the action. The overall argument that the invariant is enforced in the target system is essentially by induction. We establish the invariant initially, and then use static analysis to ensure that the invariant is maintained inductively. The invariant maintenance approach achieves correctness by construction. Other approaches to correctness of aspects and to woven code include verification through model checking [21], and runtime checking of contracts [22].

The invariant maintenance approach also provides novel contributions to the following issues.

1. *Context-Specialized Advice* – The advice of an AspectJ aspect may need to embody many case distinctions that cater for the various contexts that arise at the joinpoints, giving rise to inefficiency and complexity. It would be desirable if the aspect weaver could tailor the advice body to the specific context in which it will execute.

   In AspectJ, and most extensions of it, the advice body is a code template that is instantiated with expressions from the context of a pointcut, and is parametric on runtime values. One can increase the range of context and parametricity, but the advice is still a code template. There have been a variety of extensions to AspectJ that aim to increase the range of context that can be captured at joinpoints. For example, LogicAJ [23] and Sally [24] use logical metavariables to supplement the pattern constructs of AspectJ. Both also allow patterns in pointcut and advice definitions where AspectJ only allows constants (e.g. metavariables that match types). This mechanism supports the binding of more pieces of context (e.g. types) than AspectJ allows and thus advice bodies can be more context-sensitive. The use of metavariables also adds consistency constraints in the matching process.

The invariant maintenance approach is fully context-sensitive in the sense that maintenance code/advice is unique to each code location satisfying the pointcut specification. It can be completely different in different contexts, not just different instances of a fixed template. The list example in Section 4.4 illustrates this. To obtain the same effect in AspectJ or LogicAJ would require either (1) a single disjunction pointcut together with a big context switch in the advice, or (2) multiple aspects, one for each context together with the appropriate advice template for that context. The invariant approach neatly specifies what to do in an unbounded number of contexts. Of course it only *specifies* what to do in each context – the prescription of what to do must be achieved by synthesis. The difference is generation of arbitrary code per disruptive action versus multiple instances of a single advice template.

The use of reification is a potentially unbounded technique for bringing context into play. In this paper we have mentioned reification of history, the call stack, and the heap. There are many other possibilities. For example, reifying instruction timing information would allow one to maintain real-time properties of the base code.

2. *Aspect Interference* – Aspects may interfere with one another – since the order in which they are applied makes a semantic difference, the burden is on the AspectJ programmer to order them and resolve interferences. Interference is a fundamental problem of aspect composition.

   The invariant maintenance approach brings two extra degrees of freedom in treating interference relative to programmatic aspects: semantic abstraction and inherent concurrency. Since the maintenance specification is expressed in terms of pre/post-conditions and the invariants to be maintained, the synthesizer has maximal freedom to design a mutually satisfactory behavior. The resulting generated code does not need to be the same as the default implementation of the aspects individually. Also, the maintenance code and the system code are conceptually concurrent. To render them into conventional programming languages, it is necessary to sequentialize them, which introduces the possibility for interference. Thus the interference detection/resolution tools that have been explored for dependency analysis and instruction ordering are needed.

   On the other hand, in some situations the system code and the aspects simply conflict. In the invariant maintenance approach this is signaled by the inability to synthesize maintenance code at a pointcut location. Depending on the techniques used, a failed synthesis process may be able to return a counterexample that can be used by the developers to pinpoint the semantic discrepancy between their invariants and the system code.

   Several projects have developed analysis tools for detecting aspect interference and inferring safe orderings; e.g. LogicAJ [23]. Reflex [25] provides programmatic mechanisms for detecting potential interference and prescribing how they should compose.

3. *Evolution* – Evolution of the base program may require extending the AspectJ pointcut and advice description to reference any new class members (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 our approach, if the invariant remains unchanged, then the derived pointcut specification doesn't change. Consequently, if the base code changes, then the static analyzer can simply run the pointcut specification over the new base code. Ideally, this can be done incrementally if the structure of the changes have been recorded. In other words, an invariant will tend to be more stable under base code changes than an

aspect that implements it.

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 program variables. This paper introduces the notion of reifying extra-computational information, enabling the expression of system-level cross-cutting features as invariants.

A common use for AspectJ is to enforce preconditions, postconditions, and invariants (e.g. [4]). The intention is to insert code that dynamically *checks* those conditions and flags violations. In contrast, this paper focuses on adding code to *enforce* invariants when they would otherwise be violated. The resulting code is guaranteed to satisfy the invariant even though the base code may not.

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 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 [26], 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 base 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 lower the cost of development and evolution and provide increased assurance.

Acknowledgments: Thanks to Cordell Green, Gregor Kiczales, Peter Pepper, and Kevin Sullivan for discussions of this work. Thanks also to Lambert Meertens, Stephen Westfold, and the reviewers for useful comments on the text. This work was partially supported by the US Department of Defense and by the Office of Naval Research under Grant N00014-04-1-0727.

References

[1] Aspect-Oriented Software Development website (2003), http://www.aosd.net/.

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

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

[4] Laddad, R., AspectJ in Action. Manning Publishing Co. (2003)

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

[13] Smith, D.R., Havelund, K.: Automatic enforcement of error-handling policies. Technical report, Kestrel Technology (2004), http://www.kestreltechnology.com//

[14] Douence, R., Fradet, P., Suedholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: Aspect-Oriented Software Development (AOSD04), ACM Press (2004) 141–150

[15] Vanderperren, W., Suvee, D., Cibran, M., de Fraine, B.: Stateful aspects in JAsCo. In: Proceedings of SC 2005, Springer-Verlag LNCS (2005)

[16] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhotk, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: Proceedings of OOPSLA. (2005) 345–3640

[17] Walker, R., Viggers, K.: Implementing protocols via declarative event patterns. In: SIGSOFT Foundations of Sofware Engineering (FSE04), ACM Press (2004) 159–169

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

[19] 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

[20] Kniesel, G., Costanza, P., Austermann, M.: Jmangler – a powerful back-end for aspect-oriented programming. In Filman, R., Elrad, T., Clarke, S., Aksit, M., eds.: Aspect-Oriented Software Development, Prentice Hall (2004)

[21] Li, H., Krishnamurthi, S., Fisler, K.: Modular verification of open features through three-valued model checking. Automated Software Engineering Journal **12** (2005) 349–382

[22] Lorenz, D.H., Skotiniotis, T.: Extending design by contract for aspect-oriented programming (2005)

[23] Rho, T., Kniesel, G.: Uniform genericity for aspect languages. Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn (2004)

[24] Hanenberg, S., Unland, R.: Parametric introductions. In: Aspect-Oriented Software Development. (2003)

[25] Tanter, É., Noyé, J.: A versatile kernel for multi-language AOP. In: Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005). Lecture Notes in Computer Science, Tallin, Estonia, Springer-Verlag (2005)

[26] 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)