Requirement Enforcement by Transformation Automata

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304 USA
smith@kestrel.edu

ABSTRACT

The goal of this work is to treat safety and security policies as requirements to be composed in an aspectual style with a developing application. Policies can be expressed either logically or by means of automata. We introduce the concept of *transformation automaton*, which is an automaton whose transitions are labeled with program transformations. A transformation automaton is applied to a target program by a sound static analysis procedure. The effect is to perform a global transformation that enforces the specified policy. The semantic effect of this global transformation is explored.

In previous work we discussed how the intent of an AspectJ-style aspect can be expressed precisely and abstractly as a state invariant. Here, this result is generalized to handle invariants that are conditional and stated over both events and state properties. A policy stated in such a logical format can be translated to a transformation automaton that enforces it in a target program. The translation process is defined by a collection of inference schemes that can be mechanically instantiated and then solved, at least partially automatically, by deductive calculations.

1. INTRODUCTION

This paper takes steps toward a deep integration of two worlds - the burgeoning field of Aspect-Oriented Software Development (AOSD) and the field of formal software development by mechanized refinement. These two fields have much to offer each other. Viewing each from the point of view of the other provides insights leading to cross-fertilization and new generalizations of both.

Formal software development starts with real-world requirements that are formalized into specifications. Specifications are then subjected to a series of refinements that preserve properties while introducing implementation details. Most work on development-by-refinement takes a posit-and-prove approach: a refinement is manually written that adds implementation detail to the current design specification, and the refinement is proved correct on the side. In contrast, our work has focused on generating refinements by applying representations of abstract design knowledge and using automated reasoning [20, 21]. To achieve wider acceptance and lower lifecycle costs, it is necessary to develop highly automated means for generating refinements.

A crucial fact of complex system design is that no matter how one designs the hierarchical structure of a system, there are always concerns that cross-cut the component structure and introduce dependencies that are not exposed at the component interfaces. These dependencies complicate the understanding and evolution of the system. We view cross-cutting concerns (such as aspects, safety and security policies, nonfunctional requirements) as behavioral requirements on a system.

The main contribution of AOSD is the development and popularization of means for expressing these cross-cutting requirements, or at least implementation prescriptions for them, in modular syntax, and providing automatic methods for weaving or composing them into one's design. What has been lacking is means for specifying the intent of aspects. In previous work [22] we showed how many AspectJ-style aspects can be specified by means of state invariants, and how aspect weaving can be performed as invariant maintenance. By starting with a logical specification of the intent of a cross-cutting concern, we showed how to derive what are called the pointcuts and advice of AspectJ aspects [10]. The derivation process provides assurance that the joinpoints are complete and that the advice correctly implements the specification.

This paper continues our focus on abstract, yet precise means for specifying the intent of cross-cutting requirements. Our previous results are generalized to handle invariants that are conditional and stated over both events and state properties. In particular, the specification of safety and security policies typically requires taking behavioral context into account when deciding whether current actions are acceptable. Policies can be expressed either logically or by means of automata, as convenient.

To implement cross-cutting requirements, we introduce the concept of *transformation automata*, which are automata whose transitions are labeled with program transformations. A transformation automaton is applied to a target program by a sound static analysis procedure. The effect is to per-

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.
Copyright 2007 ACM ISBN 1-59593-671-4/07/03 ...$5.00.

form a global transformation that enforces the specified policy by applying a collection of local transformations. We show how to calculate transformation automata from specifications of cross-cutting requirements.

Formal development poses several questions. What is the semantic effect of mechanically composing a cross-cutting requirement into a program? Is the requirement correctly and completely realized? Do previously satisfied requirements remain satisfied? We examine these issues in the context of a variety of examples.

The goal of this work is to treat cross-cutting concerns as requirement specifications, and to introduce the broadest possible range of mechanisms for composing/weaving those cross-cutting concerns in the context of a refinement process that generates correct-by-construction code. After introduction of notations, we work through a series of examples.

2. PRELIMINARIES

A behavior of a program can be represented graphically as a trace of alternating states and actions

$$\mathit{state}_0 \xrightarrow{act_0} \mathit{state}_1 \xrightarrow{act_1} \mathit{state}_2 \xrightarrow{act_2} \mathit{state}_3 \cdots$$

or more formally as a sequence of *transition triples* of the form

$$\langle \mathit{state}_i, \mathit{act}_i, \mathit{state}_{i+1} \rangle,$$

where states are a mapping from variables to values, and actions are state-changing operations (i.e. program statements). 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 transition triple $\langle \mathit{state}_0, \mathit{act}, \mathit{state}_1 \rangle$, x will refer to the value of x in the preState, $\mathit{state}_0.x$, and x' refers to the value in the postState, $\mathit{state}_1.x$.

For concreteness, an action is represented by abstract syntax so that we can perform pattern-matching and other syntactical operations and tests. The following operators construct sequences, including traces: *nil*, written [], and $\mathit{append}(S, a)$, written $S :: a$ for sequence S and element a.

The semantics of a system S is given by a set of traces $\mathit{Traces}(S)$. To specify a system, we determine the observations that a stakeholder could make, and then write constraints on the observable state properties and event orderings. Here we assume that the observables of the system are exactly the states and actions of a trace; e.g. we cannot observe the state while a (primitive) action is taking place.

Actions are specified 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 $x := \sqrt{x}$.

A *refinement* is a morphism in a suitable category of specifications. Intuitively, a refinement morphism preserves structure and properties. For algebraic specifications, a refinement morphism maps vocabulary such that typing is preserved, and formulas/sentences remain provable under translation (i.e. theorems are preserved). This means that properties are preserved. For behavioral specifications, a refinement morphism maps vocabulary such that typing is preserved, theorems are preserved, and domain behavior is simulated by codomain behavior [15]. The last condition implies that if system S refines to system T then $\mathit{Traces}(T) \subseteq \mathit{Traces}(S)$, or more generally that there is a simulation map from traces of T to traces of S.

Reification

In order to specify requirements that express cross-cutting features, we often need to reify certain extra-computational values such as history, the runtime call stack, the runtime heap, or external agents.

Suppose for example that we need some way to discuss the history of the program at any point in time. The execution history of the program can be reified into the state by means of a *specification* variable (sometimes 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; so each transition has the form

$$\langle st_i,\ (\mathit{act}_i \,||\, \mathit{hist} := \mathit{hist} :: \langle st_i, \mathit{act}_i, st_{i+1} \rangle),\ st_{i+1} \rangle$$

where $\alpha || \beta$ denotes parallel composition of actions α and β. Obviously this would be an expensive variable, but it is only needed for specification purposes, and typically at most a residue of it will appear in the executable code.

Other common examples of values to reify include the call stack (to constrain dynamic control context), heap (to constrain dynamic data context), time (to state performance constraints), and agency (to express the principals who are responsible for system actions).

3. EXAMPLE: AUTOSAVE REQUIREMENT

Suppose that we are developing a data editing application, and we desire to impose an autosave requirement (adapted from [1]): every 6 changes to the data from a file, save the data back out. With the aid of the reified variable *hist*, a specification of this requirement is easily stated:

$\square\, \mathit{cnt} = (\mathit{length} \cdot \mathit{dataop}? \triangleright \mathit{action} \star \mathit{hist})\ \mathit{mod}\ 6$
$\square\, \mathit{cnt} = 5 \implies \mathit{data} = \mathit{file}$

where
(1) the *action* function selects the action from a transition $\langle \mathit{state}_i, \mathit{act}_i, \mathit{state}_{i+1} \rangle$
(2) $\star$ is the image operator, so $\mathit{action} \star \mathit{hist}$ is the list of actions performed up to the present
(3) *dataop*? holds for the representation of an action that changes the data of concern
(4) $\triangleright$ is the filter operator, so $\mathit{dataop}? \triangleright \mathit{action} \star \mathit{hist}$ is the list of dataops performed up to the present
(5) $\square$ the always modality of temporal logic [12]; $\square\phi$ asserts that the state (or transition) formula ϕ holds invariantly at every state of a trace.
In words, the two formulas assert that in every observable state, the variable *cnt* records the number of dataops modulo 6 that have occurred to that point in the current behavior (which is recorded in *hist*), and furthermore, in each state in which *cnt* has value 5, the data and the file have the same contents.

3.1 Establishing the Invariant

We have two invariants to establish, and we proceed along the lines presented in [22], by simultaneously deriving the essential parts of a inductive proof and the transformations that carry them out.

The first step is to generate code to establish the invariant initially, by satisfying the following two specifications:

assume: $\mathit{hist} = []$
achieve: $\mathit{cnt}' = \mathit{lengthDataops}\ \mathit{mod}\ 6$

where we abbreviate

$$\mathit{length} \cdot \mathit{dataop}? \triangleright \mathit{action} \star \mathit{hist}$$

by *lengthDataops*. The postcondition can be simplified as follows:

$$\mathit{cnt}' = (\mathit{length} \cdot \mathit{dataop}? \triangleright \mathit{action} \star \mathit{hist})\ \mathit{mod}\ 6$$

$$\iff \quad \{\text{ using the definition of } \mathit{hist} \text{ and simplifying }\}$$

$$\mathit{cnt}' = 0$$

which is satisfied by the initialization code

$$\mathit{cnt} \ := \ 0.$$

Generating initialization code for the other invariant is similar:

assume: $\mathit{hist} = [] \ \wedge \ \mathit{cnt} = 0$
achieve: $\mathit{cnt} = 5 \implies \mathit{data} = \mathit{file}$

The postcondition can be simplified as follows:

$$\mathit{cnt} = 5 \implies \mathit{data} = \mathit{file}$$

$$\iff \quad \{\text{ using the assumption and simplifying}\}$$

$$\mathit{true}$$

which is vacuously satisfied (i.e. by the empty code, or *skip*).

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

assume: $\mathit{hist} = []$
$\wedge$ ... initialization constraints on other reified variables
achieve: $I(x)$

3.2 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, for efficiency 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 events that satisfy a pointcut.

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. We can simplify (1) a little by assuming that $I(x)$ holds before the action, so all we need is to find a necessary condition on $\neg I(x')$.

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

assume: $\mathit{cnt} = \mathit{lengthDataops}\ \mathit{mod}\ 6$
$\wedge\ \mathit{hist}' = \mathit{hist} :: \langle \_, \mathit{act}, \_ \rangle$
$\wedge\ \mathit{cnt}' = \mathit{cnt}$
simplify: $\neg(\mathit{cnt}' = \mathit{lengthDataops}\ \mathit{mod}\ 6)$

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 *cnt* since it is a fresh variable introduced by the invariant.

Intuitively, one would expect to derive *dataop?* as the characterization of actions that could disrupt the invariant, and that is indeed the case. Since the details of the calculation are similar to examples in [22], we omit them here, in favor of later examples that exhibit new features.

For the other invariant, we set up the following inference task:

assume: $\mathit{cnt} = 5 \implies \mathit{data} = \mathit{file}$
$\wedge\ \mathit{cnt} = \mathit{lengthDataops}\ \mathit{mod}\ 6$
$\wedge\ \mathit{hist}' = \mathit{hist} :: \langle \_, \mathit{act}, \_ \rangle$
$\wedge\ \mathit{dataop}?(\mathit{act})$
$\wedge\ \mathit{cnt}' = (\mathit{cnt} + 1)\ \mathit{mod}\ 6$
simplify: $\neg(\mathit{cnt}' = 5 \implies \mathit{data}' = \mathit{file}')$

We calculate a pointcut specification as follows:

$$\neg(\mathit{cnt}' = 5 \implies \mathit{data}' = \mathit{file}')$$

$$\iff \quad \{\text{ simplifying }\}$$

$$\mathit{cnt}' = 5 \ \wedge \ \mathit{data}' \neq \mathit{file}'$$

$$\iff \quad \{\text{ using postcondition of dataop }\}$$

$$\mathit{cnt}' = 5$$

$\iff$ { using assumption on cnt' }

$(cnt + 1)\ mod\ 6 = 5$

$\iff$ { simplifying }

$cnt = 4.$

That is, it is only the occurrence of a *dataop* action when $cnt = 4$ that could possibly disrupt the invariant.

Generally, the task to infer a pointcut is given by the inference scheme in Figure 1.

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

Figure 1: Inference Scheme for Joinpoint Specification

The simplified result will typically contain a mixture of constraints, some of which constrain the action code (which actions might violate the invariant), and some of which constrain the state in which the action is taken.

3.3 Specification and Derivation of Maintenance Code

To complete the induction, for each potentially disruptive action (using the derived pointcut specification), we generate maintenance code to reestablish the invariant in parallel with it. Consider the second derived pointcut specification. Suppose that *act* is an action such that $dataop?(act)$ and suppose that $cnt = 4$. In order to preserve the invariant, we need to perform a maintenance action that satisfies

assume: $(cnt = 5 \implies data = file)$
$\wedge\ cnt = lengthDataops\ mod\ 6$
$\wedge\ dataop?(act)$
$\wedge\ cnt = 4$
$\wedge\ hist' = hist :: \langle \_, act, \_ \rangle$
$\wedge\ cnt' = (cnt + 1)\ mod\ 6$
achieve: $cnt' = 5 \implies data' = file'$

The postcondition simplifies straightforwardly to the postcondition $data' = file'$ which is satisfied by an operator, say *saveData*, that saves *data* into the *file*. Similarly, we calculate the straightforward maintenance postcondition

$$cnt' = (cnt + 1)\ mod\ 6$$

for the first derived pointcut from Section 3.2.

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 can be specified as in Figure 2. 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.

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

Figure 2: Inference Scheme for Maintenance Specification

We are not finished with this example yet. It remains to explain the mechanism whereby the parts of the induction argument, derived above, are carried out on the target system design. The next section introduces the required mechanism, and then completes the example.

4. TRANSFORMATION AUTOMATA

Program transformations have long been used to effect change on program designs, for example as in the optimizing transformations in compilers. Traditionally, a transformation has the form

$$sourcePat \rightarrow targetPat \quad \text{if } C$$

which applies to an expression *expr* in a program context if (1) *expr* matches *sourcePat* with certain bindings; i.e. $\theta = match(expr, sourcePat)$ where θ is a substitution, and (2) the condition C holds in context; i.e. $C\theta$ can be proved in context. The effect of the transformation is to replace *expr* with $targetPat\theta$.

Clearly, a transformation produces a local change in a program text. In general there is little that can be said about the semantic effect of a transformation, since *expr* can be replaced with arbitrary code. Typically however, most transformations are used to replace an expression with an equal expression (modulo context), so the effect is to preserve the semantics of the whole despite a syntactic change to a local part. We are concerned with the more general problem of whether a collection of local changes enforces a global policy and effects a global refinement.

To enforce an invariant global policy/requirement on a system, it is necessary to ensure that the invariant holds in all transitions in all system traces. We introduce the notion of a transformation automaton as the means for carrying out a systematic collection of local transformations that achieve a desired global effect.

A *transition transformation* has the form

$$[P]\{actPat\}[Q] \rightarrow [A]\{newActPat\}[B] \qquad \text{if } C$$

where P, Q, A, B are state predicates, *actPat* and *newActPat* are patterns (expressed over the specification language and using appropriate pattern notations), and C is a state predicate that expresses the conditions of the transformation.

A transition transformation *matches* an action *act* if (1) *act* matches *actionPat* with bindings θ, and (2) *act* satisfies the pre/postcondition $[P\theta, Q\theta]$; i.e.

$$P\theta \implies wp(act, Q\theta)$$

holds, where *wp* is the weakest precondition operator [6]. The intention is that a transition transformation matches a system action either via pattern matching with the *actionPat*, or by satisfying a pre/post-condition specification, or by a combination of the two. Deciding whether an transition transformation is enabled is undecidable in general. A practical implementation of this approach must restrict the language and logic to allow efficient decision procedures. Of course, by forgoing the use of the pre/postcondition specifications, one has ordinary transformations on each transition, and therefore fast matching.

When a transition transformation matches action *act* with substitution θ, then its effect is to replace *act* by a new action

$$\text{if } C\theta \text{ then } newAct \text{ else } act$$

where *newAct* satisfies the right-hand side (RHS) specification; i.e. such that

$$\theta' = match(newAct, newActPat\theta)$$

and

$$A\theta\theta' \implies wp(newAct, B\theta\theta')$$

holds.

A *Transformation Automaton* (TA) is an automaton whose transitions are labeled with transition transformations. We write transformation automata using a Java-like syntax of the form

```
TA policyName {
    variable-declaration*
    transition-declaration*
}
```

Each variable-declaration introduces a local variable to the policy and is declared using Java-like syntax (* is used to denote zero or more occurrences).

Each transition-declaration specifies a transition transformation as described above. Policy variables can be initialized, referenced, and modified by the transition transformations. In addition, each transition transformation can have local metavariables in its patterns that are bound to system action expressions. For purposes of this paper TAs do not have a mechanism to bind system values/objects to local variables. This simplifies the presentation and process of applying TAs, but prohibits the application of more than one instance of a policy to a system design. The extensions needed to capture source system values and support multiple policy instances can be found in [23, 24].

Since TAs track both state properties and events, they can represent the checking and enforcement of a variety of kinds of requirements, ranging from event ordering to temporal logic properties and combinations of these.

Returning to the AutoSave example, we assemble a TA from the pieces of the inductive argument that were derived above:

```
TA AutoSave {
    Nat  $cnt$
    $\{init\} \rightarrow [\,]\{\}[cnt' = 0]$
    $\{dataop?(act)\}$
            $\rightarrow [act_{pre}]\{\}[act_{post} \wedge cnt' = cnt + 1]$
    $\{dataop?(act)\}$
            $\rightarrow [act_{pre}]\{\}[act_{post} \wedge data' = file']$
                if $cnt = 4$
}
```

where *init* is a no-op action at the beginning of the program. The *init*-enabled transition transformation serves to initialize state and uses the postcondition $cnt' = 0$ derived in Section 3.1. The remaining transition transformations are assembled from the derived pointcut specifications (from Section 3.2) and corresponding derived maintenance code specifications (from Section 3.3). Each derived pointcut specification forms the left-hand side (LHS) and the corresponding specification of maintenance code forms the right-hand side(RHS). The predicates act_{pre} and act_{post} denote the pre- and post-conditions of *act*, respectively. Also, we use a predicate on actions in place of a pattern, here *dataop*?. This is more concise for communication purposes and avoids some of the formal noise which is necessary in particular pattern languages. Also we omit the pre- and post-conditions, action patterns, and conditions when they provide no constraints.

The AutoSave TA can be made more concise by using some abbreviations for transformation patterns that are both commonly occurring and have pleasant semantic properties. Each of these abbreviations effects a refinement - it preserves properties of the system action as well as establishing a new property.

Abbreviation	*RHS pattern*
achieve R	$[act_{pre}]\{\}[act_{post} \wedge R]$
maintain I	$[act_{pre} \wedge I]\{\}[act_{post} \wedge I]$
ensure $[P, Q]$	$[act_{pre} \wedge P]\{\}[act_{post} \wedge Q]$
ok	$\{act\}$

Figure 3: TA Abbreviations

With these abbreviations, the AutoSave TA can be expressed more compactly as

```
TA AutoSave {
  Nat  $cnt$
  $\{init\} \rightarrow$ achieve $[cnt' = 0]$
  $\{dataop?(act)\} \rightarrow$ achieve $[cnt' = cnt + 1 \; mod \; 6]$
  $\{dataop?(act)\} \rightarrow$ achieve $[data' = file']$
```

if $cnt = 4$
}

or, after carrying out the straightforward syntheses,

```
TA AutoSave {
  Nat  $cnt$
  $\{init\} \rightarrow \{cnt := 0\}$
  $\{dataop?(act)\} \rightarrow \{act;\ cnt := cnt + 1\ mod\ 6\}$
  $\{dataop?(act)\} \rightarrow \{act; saveData\}$ if  $cnt = 4$
}
```

The deductive calculations that translate a logically stated policy into a TA are similar to those performed in the Finite Differencing transformation [14, 20]. They can be automated over some domains, but in general may require some user interaction.

Applying Transformation Automata

The application of a transformation automaton to a system design is accomplished by a form of sound static analysis. Requiring the analysis to be sound means that the transition transformations are applied locally to all program actions to which the transformations apply. This means that in all traces and all transitions in each trace, if a TA transition transformation applies, then it has been applied. There are no false negatives. This key property enables us to assert strong semantic claims about the design after transformation by a TA.

Since we are consumers and not developers of static analysis technology, we only informally specify the necessary techniques here. The analysis algorithms are well-known, e.g. [5, 18, 2], although practical implementations must pay careful attention to efficiency.

The strategy for applying a transformation automaton proceeds in stages, as presented below.

The first stage is a flow-sensitive interprocedural dataflow analysis that simulates the transformation automata over the Control Flow Graph (CFG) of the system design. The result of TA simulation includes (1) a map from each source control point to a representation of possible policy variable values, (2) a map from each source code action to a set of policy transitions, and (3) a summary of the state changes effected by method calls.

In the second stage, the transition transformations that label each system action are applied. Schematically, let *act* be a system action that is labeled with policy transition

$$act \rightarrow\ newAct \text{ if } C$$

and suppose that the control point just before *act* has formula V as the representation of possible policy variable values. Soundness of static analysis means that V characterizes a superset of values that the policy variables can take on over all possible system traces. As discussed above, the effect of applying the transition transformation is to replace *act* with

$$\text{if } C\theta \text{ then } newAct \text{ else } act.$$

Simplifying $C\theta$ with respect to V can simplify the whole conditional, especially if $C\theta$ reduces to *true* or *false*.

For example, suppose that $Transpose$ is a system action that satisfies *dataop*?. Then the AutoSave transition

$$\{dataop?(act)\} \rightarrow \{act; saveData\} \text{ if } \ cnt = 4$$

matches and it results in the replacement of $Transpose$ by

```
if $cnt = 4$
    then ($Transpose$; $saveData$)
    else $Transpose$.
```

If the contextual property representation V is $cnt \in \{0..5\}$, then no simplification can be performed. If V is $cnt \in \{4\}$, then the conditional simplifies to just the then-branch.

Finally, any necessary synthesis is performed on pre/post-condition specifications that have been inserted into the design. Aside from the synthesis subtasks, a TA can be applied automatically.

Returning to the AutoSave example again, the net effect of applying TA `AutoSave` to a design $D0$ resulting in design $D1$ is to enforce the invariants, allowing us to assert

$$D1 \vdash \Box\ cnt = (length \cdot dataop? \triangleright action \star hist)\ mod\ 6$$

and

$$D1 \vdash \Box\ cnt = 5 \implies data = file.$$

In this case it is also clear that D1 is a refinement of $D0$ because we have only used refinement-inducing transition transformations.

5. MORE EXAMPLES

5.1 Access Control

Essentially, access control policies prescribe which agents are allowed to access which resources. More elaborate policies may also take into account the type of access, the time of access, roles, and other features. Lampson's permission tables [11] are the basic extensional way to represent a policy – as a relation between agents/subjects/principals, and resources/objects (and possibly action-type, time, etc.). Role-Based Access Control [8] is a leading current approach to represent the permissions tables in a rule-based way that (1) is natural and compact and (2) allows easier maintenance/evolution than a tabular/relational format.

Our overarching concern is to formally specify and enforce cross-cutting requirements on a system. Many requirements can be specified as state invariants that are given by a state predicate that is required to hold before and after each system action. Other requirements place constraints on the order of system actions. Access control policies are requirements on both events (an action to access a protected resource) and state properties (the current permission table).

Intuitively, access control (or authorization) is a requirement that whenever a system action *act* whose principal or agent a accesses resource r, then a has current permission to access r. Although the policy is easy to state in a positive way (i.e. what behaviors are allowed), it is applied with the understanding that the target design may not satisfy the requirement, and there may be a need to deal with exceptions to it. If we think of the policy as expressing normal behavior, then ultimately the specification of it must deal with possible departures from normal behavior.

In order to formalize access control we need some way to discuss the principal of an action, and current permissions. Both of these are extra-computational entities, so we must reify them in order to mention them in a formal requirement.

Reifying Agency and Permissions

To formalize access control, we must reify the agents who are the initiators of system actions. To do so, we introduce a finite type *Agent*, and label each action in a trace with an *Agent*. Not all labelings make sense - there are constraints on consistent labelings. The main constraint is that if system action α_i is labeled with agent a (meaning that a is the principal behind action α_i), and the control of the system naturally flows from α_i to α_{i+1}, then α_{i+1} is also labeled with agent a. Since each action in a trace has a unique label, we write $prin(act)$ to denote that label.

The semantics of the system is now all possible system traces with all possible consistent labelings of system actions with agents.

The other reification we need is the permissions. We add a finite type *Resource* and a finite map $ACP : Agent \times Resource \rightarrow Boolean$ (Access Control Permissions). We'll assume that ACP is a variable and that it can change from system state to system state. It is not obvious what kinds of constraints to put on these changes, so we won't assume any.

The semantics of the system is now all possible system traces with all possible consistent labelings of system actions with *Agent* s and with all possible values of ACP at system states.

Comments on this semantics:

1. *Messages* – A service (method) that passively waits for control and data, acquires the agency of the invoker. On the other hand, a process B that receives a message from process A naturally continues on its course with its agency unchanged.

2. *Delegation* – The situation in which agent A temporarily endows agent B with some of A's permissions is handled in traces by B temporarily gaining additional permissions. What is not modeled is the situation in which A passes her credentials to B so that B can act as A – credentials are an implementation concept used to satisfy requirements on authentication and access control. The semantic model here is more abstract and admits both credential-based implementations as well as others.

3. *Permission table modifications* – Agent actions that modify the ACP (permission table) are not modeled. Again, the idea is to abstract away implementation detail. A more elaborate semantical model would include (i) the principal behind the actions that change the access control policy (ACP) and (ii) permissions to effect such changes.

This is a fairly simple semantics for reifying identity in a system. Doubtless a more elaborate model could be constructed. This one is accurate enough for present purposes.

Access Control Requirement

We can now specify the access control requirement on system S; for each trace $tr : Traces(S)$, transition $\langle s, act, s' \rangle \in tr$, and resource $r : Resource$:

$$access?(act, r) \implies s.ACP(principal(act), r)$$

where $access?(act, r)$ holds if action act directly accesses resource r. The requirement states that if the current action directly accesses resource r, and the principal behind the action is a, then $ACP(a, r)$ holds in the prestate (i.e. agent a has permission to access resource r). Naturally, there are many variants and elaborations of this requirement, but this form lets us treat the essential ideas.

The reader should not confuse a simple clear specification with the ease of implementing it - accurate tracking of identity in a system is a notoriously difficult problem.

Enforcing the Requirement

In order to correctly realize the requirement in the target code, we proceed by direct synthesis.

First, we can derive a joinpoint specification as a necessary condition that a system action violates the requirement. We instantiate the inference scheme in Figure 1 as follows.

assume: $hist' = hist :: \langle s, act, s' \rangle$
simplify: $\neg(access?(act, r) \Rightarrow s.ACP(prin(act), r))$

and calculate a pointcut specification as follows:

$$\neg(access?(act, r) \implies s.ACP(prin(act), r))$$

$$\iff \quad \{ \text{ simplifying } \}$$

$$access?(act, r) \land \neg s.ACP(prin(act), r)$$

as one expects. The constraint on the system action, $access?(act, r)$ will serve as a joinpoint specification, and the state predicate $\neg s.ACP(prin(act), r)$ will serve as the condition on a policy transition in a TA.

Next, suppose that the current action act satisfies the joinpoint specification and has the particular specification

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

then the code to enforce the requirement can be specified by instantiating the inference scheme from Figure 2.

assume: $P(x) \wedge access?(act, r)$
$\wedge\, hist' = hist :: \langle s, act, s' \rangle$
achieve: $Q(x, x')$
$\wedge\, (access?(act, r) \Rightarrow s.ACP(prin(act), r))$

We refine the postcondition as follows:

$$Q(x, x') \wedge (access?(act, r) \Rightarrow s.ACP(prin(act), r))$$

$\Longleftrightarrow$ { simplifying }

$$Q(x, x') \wedge s.ACP(prin(act), r)$$

$\Longleftrightarrow$ { ordering the evaluation }

```
if $s.ACP(prin(act), r)$
    then $Q(x, x')$
    else false.
```

We have derived a postcondition specification for an action that would jointly realize both the current action and satisfy the access control requirement. The fact that the postcondition is *false* in one case is the essence of the semantic problem - we have deduced inconsistency between the current system design and the access control policy. Although we have calculated a correct refinement, it specifies a step in the design that is unimplementable! Just as any specification refines to the inconsistent specification, it is mathematically sound to have an action specification refine to an inconsistent action specification.

A transformation automaton to realize the policy as calculated above is

```
TA AccessControl {
    Resource $r$
    $\{access?(act, r)\} \rightarrow$ achieve $[false]$
                                if $\neg ACP(prin(act), r)$
}
```

What we can say is that if the source design $D0$ satisfies invariant R; i.e. $D0 \vdash \Box R$, and $D1$ results from the application of TA AccessControl, then

$$D1 \vdash (R \wedge AC)\ \mathcal{W}\ false$$

where AC is the access control invariant and $\mathcal{W}$ is the *unless* modality of temporal logic [12]. In words, $D1$ traces satisfy the state/transition property $R \wedge AC$ up to the point (if any) that the transition has a *false* postcondition. Another way of putting it is that the composition of the policy and $D0$ has preserved its safety properties, but has possibly decreased its liveness properties.

Of course, we cannot have a program with an unimplementable action in it. The solution is to weaken the postcondition to something implementable. The following TA specifies the throwing of an exception

```
TA AccessControl1 {
    Resource $r$
    $\{access?(act, r)\} \rightarrow$ {throw new error("...")}
                                if $\neg ACP(prin(act), r)$
}
```

Current work on self-healing systems attempts to deal with situations like this, albeit dynamically. Ideally, there is a way to lift out of the black hole of an inconsistency and take an action that allows the system to continue toward its goals.

5.2 A Simple Information Flow Policy

Consider the following simple security policy which is adapted from [19]. If a process ever reads from a particular file f, it is henceforth not allowed to send any messages. The policy states an information flow requirement. One might want to automatically enforce an instance of this policy on an applet downloaded onto a personal computer.

This example is distinguished from previous examples in that it is a pure event ordering constraint, whereas AutoSave is a state invariant and AccessControl constrains an action and the state in which it executes.

The reification of history allows us to represent the event ordering as a state invariant. If $Send$ matches any $send(..)$ action and $Readf$ matches any action that reads file f, then the policy can be expressed as an invariant: for all traces $tr : Traces(S)$ and transitions $\langle s, act, s' \rangle \in tr$

$$Send(act) \Rightarrow \neg\exists(a)(a \in action \star hist\ \wedge\ Readf(a))$$

however, this seems less than straightforward. Constraints on the order of events are often more naturally expressed using the tools of language theory: regular expressions, recognition automata, grammars. Using regular expressions for example allows the straightforward formulation

$$Send^{*} Readf^{*}$$

and a corresponding automaton is similarly clear. Note that all of these formulations specify normal or allowed behaviors but do not prescribe what to do with violations.

Our approach is to generate a TA that effects the specified policy, allowing developers to fill in how to handle violations.

```
TA InfoFlow {
    Boolean $rf$
    $\{init\} \rightarrow$ achieve $[rf' = false]$
    $\{Send\} \rightarrow$ ok      if $\neg rf$
    $\{Readf\} \rightarrow$ achieve $[rf' = true]$
}
```

where rf flags whether a $Readf$ action has occurred. Using a derivation similar to that for AutoSave and AccessControl, we can derive the point of inconsistency (sending when condition rf holds). Here we manually weaken the inconsistent specification to *abort* resulting in

```
TA InfoFlow {
    Boolean $rf$
    $\{init\} \rightarrow$ achieve $[rf' = false]$
    $\{Send\} \rightarrow$  abort     if $rf$
    $\{Readf\} \rightarrow$ achieve $[rf' = true]$
}
```

6. RELATED WORK

This work ties together research in a wide range of topic areas. The refinement view offers the opportunity to abstract aspects to the level of requirement specification and to treat aspect weaving as a powerful new tool for generating specification refinements. There is an opportunity to unify aspect weaving with other related techniques, including intrusion detection [26], Software Fault Isolation [7], security policy enforcement [19], and others, in addition to software development by refinement.

Runtime verification is a recent field that foregoes full program verification in favor of runtime monitoring of code with respect to a specified property of interest [3, 9]. Transformation automata can be seen as a generalization of runtime verification. Although we haven't emphasized it, when static analysis cannot decide whether a policy transition applies, then the decision must be pushed to runtime when more information is available. As such, runtime monitoring of a property then becomes a special case of applying a TA in which we defer all decisions to runtime. The static analysis performed in our approach has the effect of optimizing the runtime monitors - if we can prove statically that a certain property holds at a code location for all behaviors, then there is no need to monitor it. Also, static analysis may be able to simplify the monitoring code without eliminating it entirely, resulting in lower overhead.

The next step is to both monitor the code and take action when the policy is about to be violated. Schneider [19] defines a class of enforceable security policies as a subclass of safety properties, and uses a form of finite state machine (labeled with an event vocabulary) to express them. The effect of applying a security policy is to abort the system whenever it is about to violate the policy. In [7] the authors inject the policy automaton at each code location and then use partial evaluation to optimize away all or most of the inlined code. In [4] Colcombet and Fradet propose a similar approach except that static analysis (vs partial evaluation) is used to optimize away unnecessary runtime code. Static analysis can exploit more context and can in general optimize away more of the runtime monitoring code.

TA's generalize previous work in transformations in the following ways. The automaton provides behavioral context for the transition transformations, thereby providing more flexible and coordinated control over when they are applied. By packaging a collection of related transformations and using static analysis to explore the behaviors of the target system code, a new range of global effects are enabled. Also, the use of optional pre/post-condition specifications for both matching and target code generation is unique to our knowledge, although our Refine [17] system allowed a limited postcondition capability in specifying target code.

Recent work in AOSD has extended AspectJ concepts in various dimensions. Several authors have proposed generalizing pointcuts to take behavioral context into account, e.g. tracecuts [1], Jasco [25], PQL [13], [27], and our own policy automata [23]. Other works have increased the amount of static and reflective context that can be picked up at program points. TA's generalize previous work on AOP, including work on behavioral pointcut specifications. One can include stacks and other data structures internally to gain full computability power. Also, one could write TA's that have arbitrarily complex pre/post-conditions on their RHS which would entail arbitrarily hard synthesis problems to effect them. Effective implementations of TA's would likely place restrictions on the expressiveness to gain full automation of the enforcement process. As a special case, if no synthesis tasks appear on the RHS of transitions, then application of a TA can be fully automatic.

To our knowledge, the work on retrenchment by Poppleton and Banach [16] is the only other work that confronts the issue of transformations that impose limitations on a design from a refinement point-of-view. Their solution is to define a generalization of refinement that allows preconditions to strengthen and postconditions to weaken in some situations. Their approach is broadly consistent with the discussion above: enforcing a policy typically strengthens the guards on actions, and in the case of a derived inconsistency, we are forced to weaken an inconsistent postcondition to make progress.

7. CONCLUDING REMARKS

This paper takes a step in the direction of integrating and cross-fertilizing the two fields of AOSD and software development by refinement. The unification requires generalizations of concepts from both fields.

This paper advocates the following process for enforcing global system requirements. First, a natural specification of a requirement is translated, via some deductive calculation, into a transformation automaton. Static analysis simulates the TA over the target system design, and then applies the component transformations of the TA. The resulting transformed design satisfies the given requirement, and under certain conditions, is a refinement of the starting design. The composition process preserves the invariants of the starting design, but may reduce its liveness. That is, the enforcement of safety and security policies on a design may result in the curtailment of some behaviors that violate the policies.

This overall process enriches previous approachs to refinement by offering an automated technique for folding requirements into a design. The refinement process starts by focusing on a subset of requirements, say, to meet key functional and performance needs. Then, one can add in other requirements incrementally. Feedback from the enforcement process informs the revision of earlier design decisions, hopefully leading to designs that satisfy all requirements under a broader range of conditions.

Acknowledgements: This work benefited from discussions with Klaus Havelund, Dusko Pavlovic, Peter Pepper, and from comments from the reviewers. This work was partially supported by the Office of Naval Research under Grant N00014-04-1-0727 and by the US Department of Defense.

8. REFERENCES

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

[2] Brat, G., and Venet, A. Precise and scalable static program analysis of NASA flight software. In *Proceedings of the 2005 IEEE Aerospace Conference* (2005).

[3] Cohen, D., Feather, M. S., Narayanaswamy, K., and Fickas, S. S. Automatic monitoring of software requirements. In *ICSE '97: Proceedings of the 19th international conference on Software engineering* (1997), ACM Press, pp. 602–603.

[4] Colcombet, T., and Fradet, P. Enforcing trace properties by program transformation. In *Proc. 27th ACM Symp. on Principles of Programming Languages* (Jan. 2000), pp. 54–66.

[5] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1977), ACM, pp. 238–252.

[6] Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

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

[8] Ferraiolo, D., and Kuhn, D. Role based access control. In *15th National Computer Security Conference* (1992).

[9] Havelund, K., and Rosu, G. Monitoring Java programs with Java PathExplorer. In *Electronic Notes in Theoretical Computer Science* (2001), K. Havelund and G. Rosu, Eds., vol. 55, Elsevier.

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

[11] Lampson, B. W. Protection and access control in operating systems. *Operating Systems, Infotech State of the Art Report 14* (1972), 309–326.

[12] Manna, Z., and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, 1992.

[13] Martinand, M., Livshits, B., and Lam, M. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications* (2005), ACM Press.

[14] Paige, R., and Koenig, S. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 402–454.

[15] Pavlovic, D., and Smith, D. R. Composition and refinement of behavioral specifications. In *Proceedings of Sixteenth International Conference on Automated Software Engineering* (2001), IEEE Computer Society Press, pp. 157–165.

[16] Poppleton, M., and Banach, R. Retrenchment: Extending the reach of refinement. In *Proceedings of the Fourteenth Automated Software Engineering Conference* (1999), IEEE Computer Society Press, pp. 158–165.

[17] Reasoning Systems, Palo Alto, CA. *The $REFINE^{TM}$ User's Guide*, 1985.

[18] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (1995), ACM, pp. 49–61.

[19] Schneider, F. Enforceable security policies. *ACM Transactions on Information and System Security 3*, 1 (February 2000), 30–50.

[20] Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (1990), 1024–1043.

[21] Smith, D. R. Mechanizing the development of software. In *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, M. Broy and R. Steinbrueggen, Eds. IOS Press, Amsterdam, 1999, pp. 251–292.

[22] Smith, D. R. Aspects as invariants. In *Automatic Program Development: a Tribute to Robert Paige* (2006), O. Danvy, F. Henglein, H. Mairson, and A. Pettorosi, Eds., Springer-Verlag LNCS. (earlier version in Proceedings of GPCE-04, LNCS 3286, 39-54).

[23] Smith, D. R., and Havelund, K. Automatic enforcement of error-handling policies. Tech. rep., Kestrel Technology, September 2004.

[24] Smith, D. R., and Havelund, K. Enforcing safety and security policies. Tech. rep., Kestrel Technology, December 2005.

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

[26] Vigna, G., and Kemmerer, R. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security 7*, 1 (1999), 37–71.

[27] Walker, R., and Viggers, K. Implementing protocols via declarative event patterns. In *SIGSOFT Foundations of Software Engineering (FSE04)* (2004), ACM Press, pp. 159–169.