Temporal Aspects as Security Automata

Peter Hui

CTI, DePaul University

Chicago, IL, USA

James Riely*

CTI, DePaul University

Chicago, IL, USA

Abstract

Aspect-oriented programming (AOP) has been touted as a promising paradigm for managing complex software-security concerns. Roughly, AOP allows the security-sensitive events in a system to be specified separately from core functionality. The events of interest are specified in a pointcut*. When a pointcut triggers, control is redirected to* advice*, which intercepts the event, potentially redirecting it to an error handler.*

Many interesting security properties are history-dependent; however, currently deployed pointcut languages cannot express history-sensitivity (mechanisms like cflow *in AspectJ capture only the current call stack.) We present a language of pointcuts with past-time temporal operators and discuss their implementation using a variant of security automata. The main result is a proof that the implementation is correct.*

Refining our earlier work ([6]), we define a minimal language of events and aspects in which "everything is an aspect". The minimalist approach serves to clarify the issues and may be of independent interest.

1. Introduction

Aspect-oriented programming (AOP) ([12]) is a relatively new programming paradigm designed to address concerns that cut across encapsulation boundaries of traditional approaches. In this model, the programmer defines *aspects*, each consisting of an *advice body* – a block of code – and a *pointcut*, which states when the code is to be executed. Current implementations allow for the user to define pointcuts which trigger off of a specified atomic event, but facilities for triggering of a program's history is typically limited to the current call stack (as in AspectJ's cflow).

AOP has some potential for specifying and enforcing security policies. However, many such policies are both *history-sensitive* and *dynamic* (likely to change at runtime).

In this paper, we define a syntax and operational semantics for *temporal aspects*, which allow for pointcuts to be defined temporally— that is, in terms of events which have happened in the past. For instance, we would like to be able to declare advice which triggers when some function *f* is called, but only if a function *g* has been called at some point in the program's history. AspectJ's cflow can only capture the case where *g* lies in the call stack at the time when *f* is invoked. An obvious solution is to record every single event during the course of the program's execution. Such an implementation is clearly impractical for long-lived programs. In this vein, we present an equivalent, automaton-based semantics, to be used as a model for implementation, which records only relevant events. The automaton state provides an abstraction of the history, and our main result demonstrates that this abstract view faithfully implements the original semantics.

We use a variant of Schneider's security automata [16]. A *security automaton* enforces a security policy by monitoring the execution of a target system, and intercepting instructions which would otherwise violate the specified policy. For instance, a user may specify that subsequent to a *FileRead* operation, the user is forbidden from executing a *Send* operation. The corresponding automaton would monitor the target system, watching for instances of *FileRead*. If one was seen, the automaton would then monitor the system for an attempted *Send*, and if such an attempt were made, it would intercept the call and presumably execute some error handling code instead.

Security automata have been widely investigated as a means of implementing security policies. In [21], Walker uses security automata to encode security policies to be enforced in automatically generated code. In [20], Erlingsson and Schneider use security automata to implement software fault isolation security policies, which prevent memory accesses outside of the allowable address space. In that work, they discuss techniques used to merge security automata directly into binary code at the x86 assembler and Java Virtual Machine Language (JVML) level. In [4], Barker and Stuckey investigate role based and temporal role based access control policies, implemented using constraint logic specifications. In [18], Thiemann incorporates security au-

tomata into an interpreter for a simply typed call-by-value lambda calculus, which he then translates to an equivalent two-level lambda calculus, upon which type specialization removes all run-time operations involving security state. The limitations of stack-based security policies are explored in [11]; history-based solutions are presented in [1]. Our work can be used as an alternative implementation technique for the ideas in the later paper.

Several recent projects have studied history sensitivity in aspect languages. Douence, et. al. [10, 9] describe *event-based AOP*, in which advice is defined in-line with the event sequences that trigger it; the semantics is given in terms of weaving. Walker and Viggers [23] use a context-free grammar of *tracecuts*. Allan et. al. [2] extend this approach to *tracematches*, providing a novel technique to accommodate variable bindings, while restricting attention to regular properties. Stolz and Bodden [17] describe a technique for instrumenting Java bytecodes with LTL formula, using aspects to implement transitions in the underlying alternating automata. Bockish, et. al. [5] present a method for recording program history using a prolog database and using this to fire advice. De Fraine, et. al. [8] study dynamic weaving as a method for implementing stateful advice.

We contribute to this body of work by providing a foundational language for expressing dynamically loaded advice in a temporal framework, allowing us to define a full source-language semantics and to prove the correctness of its implementation. The situation is complicated by two facts: First, a pointcut may cause an event to be intercepted *before* it occurs; this is typical of security policies that specify sequences which must be aborted, rather than those which are allowed. Second, new advice may arrive at runtime, dynamically modifying existing policies. In both cases a key difficulty is getting the semantics of the source language "right". Refining our previous work [6], we adopt a minimalist approach which lays bare the essence of the problem without having to deal with the overhead of object-oriented details. Other work on the semantic foundations of AOP includes [22, 3, 14, 24, 19, 15, 7, 13].

We proceed as follows: in Section 2, we provide a motivating example. In Section 3, we define Polyadic μABC, a minimal aspect-based calculus defining roles, advice, and non-temporal advised messages. In Section 4, we augment Polyadic μABC to include temporal pointcuts, specified using a subset of the regular expressions, namely those of the form $\phi\alpha$, where ϕ is a regular expression abstracting the program's history, and α is the atomic event (i.e., `call`) which triggers the advice. In Section 5, we define an equivalent, automaton-based implementation semantics. In Section 6, we prove equivalence of the two semantics by providing a translation of a configuration in the history-based semantics to an equivalent configuration in the automaton-based semantics, and showing that the translation is preserved across

evaluation. Future work is discussed in Section 7.

2. Motivation

The following automaton implements a security policy which prohibits *Send* operations after a *FileRead* has been executed [16].
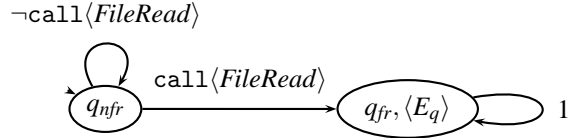
Our presentation differs slightly from that of [16] in that we attach an error handling aspect E_q to state q_{fr}. Its task is to watch for and intercept an attempted `call`$\langle Send \rangle$. We attach an aspect to the state instead of transitioning into a new state because transitions represent *committed* function calls— our intent is to block the `call`$\langle Send \rangle$, whereas transitioning into a new state would indicate that we have indeed committed it.

Now, say that at some point during the program's execution, the user executes a `call`$\langle FileRead \rangle$, and as a result, the automaton is in state q_{fr}. Furthermore, suppose that at this point, a new quarantine policy is added to the system, which prohibits a user from logging into some system *A* after a *FileRead* is called. One possible automaton for this policy is shown below:
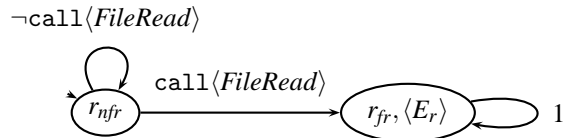
Here, E_r is the error handling advice which monitors for a `call`$\langle Login, A \rangle$ after seeing a `call`$\langle FileRead \rangle$. It may seem that the automaton resulting from adding the quarantine policy to the "read-send" policy is simply the product of the above two automata. However, in general this is unimplementable without storing the entire history. New policies may reference arbitrary events in the system history, whereas a given security automaton is committed to a particular abstract view of that history. Our solution is simple: we interpret policies as holding only from the point at which they are implemented.

Consider, in our example, what happens if the next operation is a `call`$\langle Login, A \rangle$. If we "play back" the program history $(\text{call}\langle FileRead \rangle, \text{call}\langle Login, A \rangle)$ on the product automaton, advice $\langle E_r \rangle$ will fire, which is incorrect according to our interpretation — the quarantine policy was implemented *after* the `call`$\langle FileRead \rangle$. Thus, when constructing the combined automaton, we must be careful to take into account the history of the execution.

As this example shows, while the implementation of such security policies using finite automata is straightforward, a subtlety arises when new policies may be added at runtime; one must be careful in defining which program traces are in fact captured by a new policy added to a running system. To clarify the issues, we define the semantics of dynamic temporal aspects over complete execution histories. We subsequently provide an equivalent, automaton-based implementation semantics which records only an abstraction of the execution history. Finally, we define a translation between states in the two semantics and prove that this translation commutes with evaluation.

3. Polyadic μABC

NOTATION. For any metavariable X, we write $\bar{X}$ for an ordered sequence of X's.

We define a polyadic variant of μABC, introduced in [6]. The earlier paper followed the style of object-oriented languages; each message "$p \to q : \ell$" had a source p, a destination q, and a name ℓ. Such a message is *triadic* in that its meaning depends on a triple of names, or *roles*. Here we generalize triadic messages to polyadic *events*, $\langle p_1, \ldots, p_n \rangle$ (equivalently $\langle \bar{p} \rangle$), with triadic messages as a special case "$\langle p, q, \ell \rangle$".

For simplicity, in this paper, we look at a single-threaded variant. At each moment in runtime, there is a single event $\langle \bar{p} \rangle$ under consideration. Execution is determined by *advice* that triggers on the event. At any given moment, the current event is decorated with a vector of advice $\bar{a}$, which is waiting to process the event. Thus $a_1, \ldots, a_n \langle \bar{p} \rangle$ indicates that advices named a_i are waiting to process event $\bar{p}$. We say that a_i *advises* $\bar{p}$, and that $\bar{a} \langle \bar{p} \rangle$ is an *advised* event. For consistency with the precedence of declarations, we read the advice list from right to left; thus a_n is the first advice to process the event.

The special advice call initiates advice lookup. When call$\langle \bar{p} \rangle$ executes, all the advice triggering on $\langle \bar{p} \rangle$ is listed, resulting in a new execution state: $\bar{a} \langle \bar{p} \rangle$. To determine whether an advice is triggered, we use the pointcut α. Pointcuts may be defined to trigger on an exact role, or a set of roles. We facilitate the specification of such sets using a role preorder, with maximal element top.

An advice body adv $a[\alpha] = u(\bar{x}) N$ is parameterized both on the event $\bar{x}$ and the remaining advice u. Following the terminology of *around* advice in AspectJ, we refer to u as the *proceed* variable.

3.1. Syntax and Evaluation

We give the syntax and evaluation semantics of the language parametrically with respect to pointcuts α and pointcut satisfaction $\bar{D} \vdash \langle \bar{p} \rangle$ sat α, described in the next subsec-

tion. Note that terms have the form $\bar{D}; \bar{a} \langle \bar{p} \rangle$; ie, a term is a list of declarations followed by a single advised event. We refer to $\bar{p}$ as the *current event*, $\bar{a}$ as the *current advice list*, and a_n as the *current advice* ($\bar{a} = a_1, \ldots, a_n$).

TERM SYNTAX

a–e, u–w	Advice Names; call, commit reserved
f–t, x–z	Role Names; top reserved
$D, E ::=$	Declarations
role $p < q$	Role; $dn(\text{role } p) = p$
adv $a[\alpha] = u(\bar{x}) N$	Advice; $dn(\text{adv } a) = a$;
	u and $\bar{x}$ bound in N
$L, M, N ::=$	Terms
$D; M$	Declaration; $dn(D)$ bound in M
$\bar{a} \langle \bar{p} \rangle$	Message

NOTATION. We write $dn(D)$ for the declared name of D. Reserved names may not be declared. We identify syntax up to renaming of bound names. For any syntactic category with typical element $\mathcal{E}$, we write $fn(\mathcal{E})$ for the set of free names occurring in $\mathcal{E}$. We write $\mathcal{E}\{a/x\}$ for the capture avoiding substitution of a for x in $\mathcal{E}$. We write $\mathcal{E}\{\bar{a}/\bar{x}\}$ for $\mathcal{E}\{a_1/x_1, \ldots, a_n/x_n\}$; note that $\mathcal{E}\{\bar{a}/\bar{x}\}$ is defined only if $\bar{x}$ and $\bar{a}$ have the same length.

CONVENTION. To improve readability, we use the following discipline for names:
- a–e are advice names (including the reserved names call and commit);
- u–w are advice names that are bound in the body of an advice declaration;
- f–t are role names (including the reserved name top);
- x–z are role names that are bound in the body of an advice declaration;
- _ is a reserved name used to bind a name that is not of interest — that is, does not occur free in any subterm.

We drop syntactic elements that are not of interest. Consider the declaration "adv $a[\alpha] = u(\bar{x}) N$"; we may elide the name "adv$[\alpha] = u(\bar{x}) N$", or the pointcut "adv $a = u(\bar{x}) N$", or the body "adv $a[\alpha]$", or both the pointcut and the body "adv a".

Evaluation is defined using configurations which consist of a vector of declarations and a term. By EVAL-DEC, declarations are recorded in the configuration whenever they are encountered in a term. By EVAL-CALL, if an event $\langle \bar{p} \rangle$ is being processed with first advice call, then the advice list $\bar{a}$ is calculated, consisting of the advice names a_i such that the pointcut declared with a_i is satisfied by $\langle \bar{p} \rangle$. By EVAL-ADV, if an event $\langle \bar{p} \rangle$ is being processed with first advice a, then the body of a is executed; the advice body is parameterized by both the event $\langle \bar{p} \rangle$ and remaining advice $\bar{b}$. (Note that the

syntax requires that "$\bar{b}, a\langle\bar{p}\rangle$" be parsed as "$(\bar{b}, a)\langle\bar{p}\rangle$". Further note that the substitution $\bar{b}/u$ results in a well-formed term because free advice names can only appear in the context of a sequence.)

EVALUATION $(\bar{D} \rhd M \to \bar{E} \rhd N)$

$$
\begin{array}{ll}
\text{(EVAL-CALL)} & \text{(EVAL-DEC)} \\
 & \dfrac{\bar{D} \rhd E; M \to \bar{D}, E \rhd M}{} \\[2pt]
[\bar{a}] = \left[a \,\middle|\, \begin{array}{l} \bar{D} \ni \mathsf{adv}\ a[\alpha] \\ \bar{D} \vdash \langle\bar{p}\rangle \mathsf{\ sat\ } \alpha \end{array}\right] & \text{(EVAL-ADV)} \\
 & \bar{D} \ni \mathsf{adv}\ a = u(\bar{x})\ N \\[2pt]
\hline
\bar{D} \rhd \bar{b}, \mathsf{call}\langle\bar{p}\rangle \to \bar{D} \rhd \bar{b}, \bar{a}\langle\bar{p}\rangle & \bar{D} \rhd \bar{b}, a\langle\bar{p}\rangle \to \bar{D} \rhd N\{\bar{b}/u, \bar{p}/x\}
\end{array}
$$

3.2. Atomic Event Pointcuts

We now consider a simple boolean logic over events. We allow event sets to be specified using role patterns which include subroles and "varargs", ie, optional roles.[1]

POINTCUT SYNTAX

$P, Q ::=$	Role Pattern
p	Exact Role
$+p$	Sub Role
$\alpha, \beta ::=$	Atomic Event Pointcut
$\langle\bar{P}\rangle$	Call Event
$\langle\bar{P}, *\rangle$	Call Event, varargs
$\alpha \vee \beta$	Disjunction
$\neg\alpha$	Negation
$\sigma, \rho ::= \langle\bar{p}\rangle$	Atomic Event

Define 1 as $\langle*\rangle$; define 0 as $\neg 1$; and define $\alpha \wedge \beta$ as $\neg(\neg\alpha \vee \neg\beta)$. We write $\bar{D} \vdash r \leqslant p$ for the obvious preorder generated from the role declaration order. From this, we derive the following definition of pointcut satisfaction; the obvious rules for conjunction and disjunction are elided.

ATOMIC POINTCUT SATISFACTION $(\bar{D} \vdash \sigma \mathsf{\ sat\ } \alpha)$

$$
\begin{array}{ll}
\text{(SAT-CALL-ANY)} & \text{(SAT-CALL-EMPTY)} \\
\bar{D} \vdash \langle\bar{p}\rangle \mathsf{\ sat\ } \langle*\rangle & \bar{D} \vdash \langle\rangle \mathsf{\ sat\ } \langle\rangle \\[4pt]
\text{(SAT-CALL-EXACT)} & \text{(SAT-CALL-SUB)} \\
\dfrac{\bar{D} \vdash \langle\bar{q}\rangle \mathsf{\ sat\ } \langle\bar{Q}\rangle}{\bar{D} \vdash \langle r, \bar{q}\rangle \mathsf{\ sat\ } \langle r, \bar{Q}\rangle} & \dfrac{\bar{D} \vdash \langle\bar{q}\rangle \mathsf{\ sat\ } \langle\bar{Q}\rangle \quad \bar{D} \vdash r \leqslant p}{\bar{D} \vdash \langle r, \bar{q}\rangle \mathsf{\ sat\ } \langle +p, \bar{Q}\rangle}
\end{array}
$$

4. Temporal Pointcuts

We extend μABC with temporal pointcuts. To do this, we modify the language of advice to include a temporal formula ϕ in addition to the atomic formula α. Intuitively, the

---

1   In the full version we also allow vararg parameters in advice declarations, ie $\mathsf{adv}\ a[\alpha] = u(\bar{x}, *)\ N$.

pointcut fires when ϕ matches the past and α matches the current event.

In an aspect language, the ontology of events is complicated by the fact that events can be diverted; that is, an event can trigger advice that intercepts the event *before* it occurs, potentially causing the event to abort. This is particularly common in applications to security, where pointcuts often specify dangerous event sequences that interrupt normal processing. To indicate that an event is to be recorded in the history, we include the special advice commit.

Thus when the past is considered in firing a pointcut, we require that advice specify both the past ϕ and the potential future α. The past is specified as a regular expression over atomic event pointcuts; the potential future is specified as an atomic event pointcut.

SYNTAX

$D, E ::=$ $\cdots$	Declarations
$\mathsf{adv}\ a[\phi\,\alpha] = u(\bar{x})\ N$	Declare Advice
$\phi, \psi, \chi ::=$	Temporal Pointcuts
α	Atomic Event Pointcut
ε	Empty Sequence
$\phi\,\psi$	Sequence
ϕ^*	Kleene Star
$\phi + \psi$	Disjunction
$\sigma, \rho ::= \langle\bar{p}\rangle$	Atomic Events

The semantics of temporal formulas $\bar{D} \Vdash \bar{\sigma} \mathsf{\ sat\ } \phi$ is defined in the standard way (recalled in Appendix A) over strings of events, building on the semantics of atomic events ($\bar{D} \vdash \sigma \mathsf{\ sat\ } \alpha$). Note that the regular expression $\emptyset$ is represented here as the atomic event pointcut 0. We define the language of the formula as follows: $\mathcal{L}_H(\bar{D}, \phi) = \{\bar{\sigma} \mid \bar{D} \Vdash \bar{\sigma} \mathsf{\ sat\ } \phi\}$.

We now give the evaluation semantics for the language with temporal advice. We augment the semantics to record an execution history. We write $|\bar{\sigma}|$ for the length of string $\bar{\sigma}$. We define $\alpha^n \triangleq \alpha\alpha^{n-1}$, where $\alpha^0 \triangleq \varepsilon$. We write "$\mathsf{adv}\ a[\alpha] = u(\bar{x})\ N$" as shorthand for "$\mathsf{adv}\ a[1^*\,\alpha] = u(\bar{x})\ N$".

EVALUATION $(\bar{\sigma}; \bar{D} \rhd M \to \bar{\rho}; \bar{E} \rhd N)$

$$
\begin{array}{ll}
\text{(EVAL-DEC-ROLE)} & \text{(EVAL-DEC-ADV)} \\
\dfrac{\bar{\sigma}; \bar{D} \rhd \mathsf{role}\ p < q; M}{\to \bar{\sigma}; \bar{D}, \mathsf{role}\ p < q \rhd M} & \dfrac{\bar{\sigma}; \bar{D} \rhd \mathsf{adv}\ a[\phi\,\alpha] = u(\bar{x})\ N; M}{\to \bar{\sigma}; \bar{D}, \mathsf{adv}\ a[1^{|\bar{\sigma}|}\,\phi\,\alpha] = u(\bar{x})\ N \rhd M}
\end{array}
$$

$$
\text{(EVAL-COMMIT)}
$$
$$
\bar{\sigma}; \bar{D} \rhd \bar{b}, \mathsf{commit}\langle\bar{p}\rangle \to \bar{\sigma}, \langle\bar{p}\rangle; \bar{D} \rhd \bar{b}
$$

$$
\text{(EVAL-CALL)}
$$
$$
[\bar{a}] = \left[a \,\middle|\, \bar{D} \ni \mathsf{adv}\ a[\phi\,\alpha] \ \text{ and }\ \bar{D} \vdash \bar{\sigma}, \langle\bar{p}\rangle \mathsf{\ sat\ } \phi\,\alpha\right]
$$
$$
\bar{\sigma}; \bar{D} \rhd \bar{b}, \mathsf{call}\langle\bar{p}\rangle \to \bar{\sigma}; \bar{D} \rhd \bar{b}, \bar{a}\langle\bar{p}\rangle
$$

$$
\text{(EVAL-ADV)}
$$
$$
\bar{D} \ni \mathsf{adv}\ a = u(\bar{x})\ N
$$
$$
\bar{\sigma}; \bar{D} \rhd \bar{b}, a\langle\bar{p}\rangle \to \bar{\sigma}; \bar{D} \rhd N\{\bar{b}/u, \bar{p}/x\}
$$

EVAL-COMMIT causes an event to be recorded in the history. The original EVAL-DEC is split into different cases for roles and advice. EVAL-DEC-ROLE, EVAL-CALL, and EVAL-ADV are largely unchanged from the non-temporal semantics. Note only that in EVAL-CALL the history is used, along with the current event, to determine whether an advice fires.

Of particular note is the rule EVAL-DEC-ADV, which takes a newly declared advice, and prepends a string of 1s to the temporal pointcut prior to adding it to the list of declarations. The purpose of doing so is to ensure that the advice only triggers on the event α *from the point of declaration onwards*, as opposed to some event that has already occurred in the past.

5. Automaton

In this section, we define an equivalent automaton-based semantics.

Our automata are constructed from regular expressions of the form $\phi\alpha$, corresponding to an advice declaration adv $a[\phi\alpha]$, where ϕ is a regular expression abstracting the relevant events in the program history, and α is the triggering atomic event. For each advice adv $a[\phi\alpha]$, we construct the automaton for ϕ. From the point of declaration onwards, the automaton monitors program execution. If the automaton ever enters its final state, this indicates that an attempt to execute α should be intercepted, and the advice body executed instead. To implement this, we attach the advice name to each final state for the automaton. For this reason, we refer to final states as advice states:

ADVICE STATES $(\phi\checkmark)$

$$\frac{}{\varepsilon\checkmark} \quad \frac{\phi\checkmark \quad \psi\checkmark}{\phi\psi\checkmark} \quad \frac{\phi\checkmark}{\phi+\psi\checkmark} \quad \frac{\psi\checkmark}{\phi+\psi\checkmark} \quad \frac{}{\phi^*\checkmark}$$

The states are sets of temporal pointcut formulas ϕ, the transition alphabet ranges over the atomic event pointcuts α, and the transitions of the automaton are defined by the standard transition relation:

TRANSITION RELATION $(\phi \xrightarrow{\alpha} \psi)$

$$\frac{}{\alpha \xrightarrow{\alpha} \varepsilon} \quad \frac{\phi \xrightarrow{\alpha} \phi'}{\phi\psi \xrightarrow{\alpha} \phi'\psi} \quad \frac{\phi\checkmark \quad \psi \xrightarrow{\alpha} \psi'}{\phi\psi \xrightarrow{\alpha} \psi'} \quad \frac{\phi \xrightarrow{\alpha} \phi'}{\phi^* \xrightarrow{\alpha} \phi'\phi^*}$$

$$\frac{\phi \xrightarrow{\alpha} \phi'}{\phi+\psi \xrightarrow{\alpha} \phi'} \quad \frac{\psi \xrightarrow{\alpha} \psi'}{\phi+\psi \xrightarrow{\alpha} \psi'}$$

Transitions between states are taken on `commits`.

We write $\xrightarrow{\phi}$ for the reflexive transitive closure of $\xrightarrow{\alpha}$. Next, we formally state how to derive an automaton from an advice adv $a[\phi\alpha]$:

NOTATION. For any advice adv $a[\phi\alpha]$, let the automaton $\iota(\phi,a)$ *induced by a* be the security automaton with states and transitions as defined by the transition relation given above, with start state ϕ, and advice a associated with each advice state.

We represent our automata as (state, advice set) pairs:

AUTOMATON SYNTAX

$\Phi,\Psi ::= \phi \mid \phi,\Phi$		State
$\mathcal{A} ::= \langle\Phi,\bar{a}\rangle \mid \langle\Phi,\bar{a}\rangle,\mathcal{A}$		Automaton

For instance, $\mathcal{A}_R$ and $\mathcal{A}_Q$ from Section 2 are represented as $\langle\phi_0,\emptyset\rangle,\langle\phi_1,\{E_1\}\rangle$ and $\langle\psi_0,\emptyset\rangle,\langle\psi_1,\{E_2\}\rangle$, respectively, with $\phi_0 = \psi_0 \triangleq [\neg\text{call}\langle FileRead\rangle]^*\text{call}\langle FileRead\rangle 1^*$, and $\phi_1 = \psi_1 \triangleq 1^*$. The product automaton $\mathcal{A}_R \times \mathcal{A}_Q$ would be represented as

$$(\langle(\phi_0,\psi_0),\emptyset\rangle,(\langle\phi_0,\psi_1\rangle,\{E_2\}),$$
$$(\langle\phi_1,\psi_0\rangle,\{E_1\}),(\langle\phi_1,\psi_1\rangle,\{E_1,E_2\})$$

There is no need to explicitly encode the transition relation. For instance, in the product automaton just presented, we know from the definition of the transition relation that $\langle\phi_1,\psi_0\rangle \xrightarrow{\text{call}\langle FileRead\rangle} \langle\phi_1,\psi_1\rangle$. To make the presentation more readable, we elide advice when a state has none associated with it. That is, we write the state "$\langle\phi\rangle,\emptyset$" simply as ϕ.

We can modulate the transition relation from atomic event pointcuts to atomic events: define $\bar{D} \vdash \phi \xrightarrow{\sigma} \phi'$ if $\phi \xrightarrow{\alpha} \phi'$ and $\bar{D} \vdash \sigma$ sat α. Further we can lift the definition to automaton states: $\bar{D} \vdash \phi_1,\ldots,\phi_n \xRightarrow{\sigma} \psi_1,\ldots,\psi_n$ if $\bar{D} \vdash \phi_i \xrightarrow{\sigma} \psi_i$ for all i between 1 and n. Finally we lift the resulting relation $(\bar{D} \vdash \Phi \xRightarrow{\sigma} \Psi)$ to event sequences: $D \vdash \Phi_0 \xRightarrow{\sigma_1,\ldots,\sigma_n} \Phi_n$ if $\bar{D} \vdash \Phi_{i-1} \xRightarrow{\sigma_i} \Phi_i$ for all i between 1 and n.

We define the product of two automata using the standard product construction, taking the set union of each component state's associated advice names:

DEFINITION 1. For any two automata A,B,

$$A \times B = \{\langle\Phi_A,\Phi_B;\bar{a},\bar{b}\rangle | \langle\Phi_A,\bar{a}\rangle \in A, \langle\Phi_B,\bar{b}\rangle \in B\}$$

Next, we show how to merge an advice adv $a[\phi\alpha]$ with an existing automaton $\mathcal{A}$. Namely, we construct the automaton for the advice, and create the product automaton:

$$\nu(\mathcal{A},\phi,a) \triangleq \mathcal{A} \times \iota(\phi,a)$$

We now give the equivalent, automaton-based evaluation semantics to our language. Whereas previously we recorded the entire program history, we now instead maintain an automaton and state, which records only events of interest.

EVALUATION $(\mathcal{A}; \Phi; \bar{D} \rhd M \to \mathcal{A}'; \Psi; \bar{D}' \rhd M')$

(EVAL-DEC-ROLE and EVAL-ADV as before)

(EVAL-COMMIT)

$$\dfrac{\bar{D} \vdash \Phi \stackrel{\bar{p}}{\Longrightarrow} \Psi}{\mathcal{A}; \Phi; \bar{D} \rhd \bar{b}, \texttt{commit}\langle \bar{p}\rangle \to \mathcal{A}; \Psi; \bar{D} \rhd \bar{b}}$$

(EVAL-DEC-ADV)

$$\dfrac{\mathcal{A}; \Phi; \bar{D} \rhd (\mathsf{adv}\ a[\phi\alpha] = u\,(\bar{x})\ N\,;M)}{\to \mathsf{v}(\mathcal{A}, \phi, a); \langle \Phi, \phi\rangle; \bar{D}, (\mathsf{adv}\ a[\alpha] = u\,(\bar{x})\ N) \rhd M}$$

(EVAL-CALL)

$$\dfrac{[\bar{a}] = \left[\ a \ \middle|\ \begin{array}{l} \langle \Phi, (\bar{b}, a, \bar{b}')\rangle \in \mathcal{A} \\ \bar{D} \ni \mathsf{adv}\ a[\alpha] \\ \bar{D} \vdash \langle \bar{p}\rangle\ \mathsf{sat}\ \alpha \end{array}\ \right]}{\mathcal{A}; \Phi; \bar{D} \rhd \bar{b}, \texttt{call}\langle \bar{p}\rangle \to \mathcal{A}; \Phi; \bar{D} \rhd \bar{b}, \bar{a}\langle \bar{p}\rangle}$$

Operationally, EVAL-DEC-ROLE and EVAL-ADV act the same as in the history-based semantics. EVAL-DEC-ADV takes a new advice, merges it into the automaton, updates the current state, and adds the advice to the list of declarations. EVAL-CALL looks through the list of advices attached to the current state for one whose atomic pointcut matches the role vector $\bar{p}$ being called. If a matching advice is found, then the $\texttt{call}\langle \bar{p}\rangle$ is replaced with the advice body. EVAL-COMMIT simply updates the state of the automaton.

6. Equivalence

In this section, we demonstrate equivalence of the history-based semantics provided in Section 4 with the automaton-based semantics provided in Section 5 by providing a translation from a configuration in the former to an equivalent one in the latter. We conclude by showing that evaluation preserves the translation.

Intuitively, we translate a history-based configuration $\langle \bar{\sigma}, \bar{D}\rangle$ to an automaton-based configuration $\langle \mathcal{A}, \Phi, \bar{E}\rangle$ as follows: given a history $\bar{\sigma}$ and a set of declarations $\bar{D}$, we first construct an intermediate automaton $\mathcal{A}'$ using the aspect declarations in $\bar{D}$. We compute the state Φ by simulating the history $\bar{\sigma}$ on $\mathcal{A}'$. Finally, we convert the intermediate automaton $\mathcal{A}'$ to the final automaton $\mathcal{A}$ by removing intermediate states.

Recall the manner in which EVAL-DEC-ADV is defined in the history-based semantics: whenever an advice is declared, the current "timestamp" is explicitly noted in the form of a string of '1's prepended to the temporal pointcut. Thus if an advice $\mathsf{adv}\ a[\phi\alpha]$ is declared at time n, then in the history-based semantics, the pointcut is noted as $\mathsf{adv}\ a[1^n\phi\alpha]$, and the corresponding automaton in the automaton-based semantics will have a string of n "placeholder" states $\pi_1, ..., \pi_n$, where $\pi_i \stackrel{1}{\longrightarrow} \pi_{i+1}$ for i between 1

and $n-1$, and $\pi_n \stackrel{1}{\longrightarrow} \phi$, as shown below:
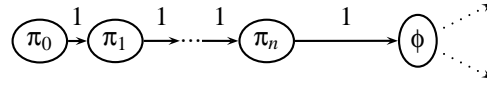
CONVENTION. In constructing an automaton for an advice $\mathsf{adv}\ a[\phi\alpha]$ declared at time n, we label the states used as placeholders for time 1 through n as $\pi_0, ..., \pi_n$, and we refer to these as π-states.

Strictly speaking, we must account for the fact that for an advice $\mathsf{adv}\ a[\phi\alpha]$, ϕ may in fact begin with a string of leading 1s. We can easily get around this by syntactically differentiating between those 1s implicitly inserted by EVAL-DEC-ADV as a timestamp, and those explicitly specified by the user. In the interest of simplifying the presentation, we choose not to do so here.

If a state $\Phi = \langle \phi_i, \psi_i, ...\chi_i\rangle$ is such that none of $\phi_i, \psi_i, ...\chi_i$ are π-states, we say that Φ is π-free. We will need to project the π-free states of an automaton, so we formally define this operation:
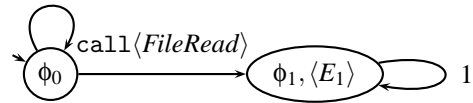
$$\mathcal{P}_{\bar{\pi}}(A) = \{\langle \Phi, \bar{a}\rangle \in \mathcal{A} | \Phi \text{ contains no } \pi \text{ states}\}$$

LEMMA 2. *For two automata $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{P}_{\bar{\pi}}(\mathcal{A} \times \mathcal{B}) = \mathcal{P}_{\bar{\pi}}(\mathcal{A}) \times \mathcal{P}_{\bar{\pi}}(\mathcal{B})$*

Proof. Immediate.   □

To construct $\mathcal{A}'$, we take the product of the automata induced by each advice in $\bar{D}$. To construct Φ, we simulate the program history $\bar{\sigma}$ on $\mathcal{A}'$. For instance, in the example in Section 2, $\mathcal{A}'$ is the product of the following automata, with ϕ_0, ϕ_1, ψ_0, and ψ_1 defined as in Section 5:

Simulating the program history $(\texttt{call}\langle \textit{FileRead}\rangle, \texttt{call}\langle \textit{Login}, A\rangle)$ on the product automaton places us in state $\langle \phi_1, \psi_0\rangle$, as expected.

We compute $\mathcal{A}$ by removing from $\mathcal{A}'$ any states containing a π state. In our example, this amounts to removing from the product automaton states $\langle (\phi_0, \pi_0)\rangle$ and $\langle (\phi_1, \pi_0), \{E_1\}\rangle$. The result is equivalent to the product automaton $\mathcal{A}_Q \times \mathcal{A}_R$, where $\mathcal{A}_Q$ and $\mathcal{A}_R$ are as in Section 2.

We now formalize the translation just discussed. That is, given a history, declaration pair $\langle\bar{\sigma},\bar{D}\rangle$, we formally show how to construct the corresponding automaton, state, declaration triple $\langle\mathcal{A},\Phi,\bar{E}\rangle$.

Our translation makes use of the following functions:

$$\mathcal{T}_{dec}(\bar{D}) = \{\text{adv } a[\alpha]\,|\,\text{adv } a[\phi\alpha] \in \bar{D}\}$$
$$\mathcal{T}_{state}(\bar{\sigma},\bar{D}) = \Psi, \text{ where}$$
$$\bar{D} = \langle\text{adv}\,\_[1^{i_1}\phi_1\,\alpha_1],\dots,\text{adv}\,\_[1^{i_n}\phi_n\,\alpha_n]\rangle \text{ and}$$
$$\langle 1^{i_1}\phi_1, ..., 1^{i_n}\phi_n\rangle \stackrel{\bar{\sigma}}{\Longrightarrow} \Psi$$

We are now in a position to define the function $\mathcal{T}$ which translates a history-based configuration $\langle\bar{\sigma};\bar{D}\rangle$ to an automaton-based configuration $\langle\mathcal{A};\Phi;\bar{E}\rangle$:

DEFINITION 3. $\mathcal{T}(\bar{\sigma};\bar{D}) = \mathcal{A};\Phi;\bar{E}$, where

$$\mathcal{A} = \mathcal{P}_{\vec{\pi}}\left[\prod_{\text{adv } a[\phi\alpha]\in\bar{D}} \iota(\phi,a)\right] \qquad \Phi = \mathcal{T}_{state}(\bar{\sigma},\bar{D})$$

$$\bar{E} = \mathcal{T}_{dec}(\bar{D})$$

We define the language of the formula as follows:

$$\mathcal{L}_A(\bar{D},\phi) = \{\bar{\sigma}\,|\,\bar{D}\vdash\phi\stackrel{\bar{\sigma}}{\Longrightarrow}\phi',\ \phi'\checkmark,\ \text{and } \phi'\in\mathcal{T}_{state}(\bar{\sigma},\bar{D})\}$$

LEMMA 4. *For all $\bar{D}$ and ϕ, $\mathcal{L}_H(\bar{D},\phi) = \mathcal{L}_A(\bar{D},\phi)$.*

Proof. By induction on the structure of $\bar{\sigma}$. $\qquad\square$

We conclude by showing that the translation is preserved by evaluation. That is, if

- $\bar{\sigma};\bar{D}\rhd M \to \bar{\sigma}';\bar{D}'\rhd M'$
- $\mathcal{T}(\bar{\sigma},\bar{D}) = \mathcal{A};\Phi;\bar{E}$,
- $\mathcal{A};\Phi;\bar{E}\rhd M \to \mathcal{A}';\Phi';\bar{E}'\rhd M'$, and
- $\mathcal{T}(\bar{\sigma}',\bar{D}') = \mathcal{A}'';\Phi'';\bar{E}''$

then $\mathcal{A}'=\mathcal{A}'',\Phi'=\Phi''$, and $\bar{E}'=\bar{E}''$, as shown below:

$$
\begin{array}{ccc}
\bar{\sigma};\bar{D} & \longrightarrow & \bar{\sigma}';\bar{D}' \\
\mathcal{T}\downarrow & & \downarrow\mathcal{T} \\
\mathcal{A};\Phi;\bar{E} & \longrightarrow & \mathcal{A}';\Phi';\bar{E}'
\end{array}
$$

PROPOSITION 5. *If $\bar{\sigma};\bar{D}\rhd M \to \bar{\sigma}';\bar{D}'\rhd M'$ and $\mathcal{T}(\bar{\sigma},\bar{D}) = \mathcal{A},\Phi,\bar{E}$, then $\mathcal{A};\Phi;\bar{E}\rhd M \to \mathcal{A}';\Phi';\bar{E}'\rhd M'$, where $\mathcal{T}(\bar{\sigma}',\bar{D}') = \mathcal{A}',\Phi',\bar{E}'$.*

Proof. In each case, we first translate the left hand side into the automaton-based semantics. We then apply the evaluation rule (e.g., EVAL-DEC-ADV) to the automaton to obtain the next configuration $\langle\mathcal{A}',\Phi',\bar{E}'\rangle$. We then translate

the right hand side into the automaton based semantics and show that the result equals $\langle\mathcal{A}',\Phi',\bar{E}'\rangle$.

In the cases of EVAL-DEC-ROLE and EVAL-ADV, this is trivial. In the case of EVAL-DEC-ADV, recall its evaluation rule in the history-based semantics:

$$\bar{\sigma};\bar{D}\rhd\text{adv } b[\psi\beta],M \to \bar{\sigma};\bar{D},\text{adv } b[1^{|\bar{\sigma}|}\psi\beta]\rhd M$$

The declarations $\bar{D}$ (equivalently $\bar{E}$) are trivially preserved by EVAL-DEC-ADV, which leaves us to show that the automaton $\mathcal{A}$ and the state Φ are preserved. Translating the left hand side yields $\mathcal{T}(\bar{\sigma};\bar{D}) = \mathcal{A};\Phi;\bar{E}$, where

$$\mathcal{A}\stackrel{\triangle}{=}\mathcal{P}_{\vec{\pi}}\left[\prod_{\text{adv } a[\phi\alpha]\in\bar{D}} \iota(\phi,a)\right] \quad \Phi\stackrel{\triangle}{=}\mathcal{T}_{state}(\bar{D}) \quad \bar{E}\stackrel{\triangle}{=}\mathcal{T}_{dec}(\bar{D})$$

By EVAL-DEC-ADV in the automaton semantics, $\mathcal{A};\Phi;\bar{E}\rhd\text{adv } b[\psi\beta],M \to \mathcal{A}';\Phi';\bar{E}'\rhd M$, where

$$\mathcal{A}'\stackrel{\triangle}{=}\mathcal{P}_{\vec{\pi}}\left[\prod_{\text{adv } a[\phi\alpha]\in\bar{D}} \iota(\phi,a)\right]\times\iota(\psi,b) \quad \Phi'\stackrel{\triangle}{=}\langle\Phi,\psi\rangle$$
$$\bar{E}'\stackrel{\triangle}{=}\mathcal{T}_{dec}(\bar{D}),b$$

Finally, we must show that $\mathcal{T}(\bar{\sigma}';\bar{D},\text{adv } b[1^{|\bar{\sigma}|}\psi\beta]) = \mathcal{A}';\Phi';\bar{E}'$. By definition, $\mathcal{T}(\bar{\sigma}';\bar{D},\text{adv } b[1^{|\bar{\sigma}|}\psi\beta]) = \mathcal{A}'';\Phi'';\bar{E}''$, where

$$\mathcal{A}'' = \mathcal{P}_{\vec{\pi}}\left[(\prod_{\text{adv } a[\phi\alpha]\in\bar{D}} \iota(\phi,a))\times\iota(1^{|\bar{\sigma}|}\psi,b)\right]$$
$$= \mathcal{P}_{\vec{\pi}}\left[(\prod_{\text{adv } a[\phi\alpha]\in\bar{D}} \iota(\phi,a))\times\iota(\psi,b)\right]$$

Finally, Lemma 2 gives us that $\mathcal{A}'=\mathcal{A}''$:

$$\mathcal{P}_{\vec{\pi}}\left[(\prod_{\text{adv } a[\phi\alpha]\in\bar{D}} \iota(\phi,a))\times\iota(\psi,b)\right]$$
$$= \mathcal{P}_{\vec{\pi}}\left[\prod_{\text{adv } a[\phi\alpha]\in\bar{D}} \iota(\phi,a)\right]\times\iota(\psi,b)$$

and hence that the automaton is preserved by EVAL-DEC-ADV.

To show that the state Φ is preserved by EVAL-DEC-ADV, we simulate $\bar{\sigma}$ on the intermediate automaton $\left[\prod_{\text{adv } a[\phi\alpha]\in\bar{D}} \iota(\phi,a)\right]\times\iota(1^{|\bar{\sigma}|}\psi,b)$. It immediately follows that the resulting state $\Phi'' = \langle\Phi,\psi\rangle = \Phi'$. The declarations $\bar{E}$ are trivially preserved by EVAL-DEC-ADV.

We now consider the case of EVAL-CALL. We must show that in a history-based configuration $\langle\bar{\sigma};\bar{D}\rangle$, for any declared advice adv $a[\phi\alpha]$, if $\bar{D}\Vdash\bar{\sigma}$ sat ϕ and $\bar{D}\vdash\bar{p}$ sat α where $\bar{p}$ is the role vector being called, then adv $a[\alpha]$ is associated with the state Φ in $\mathcal{T}(\bar{\sigma},\bar{D})$. This follows directly from Lemma 4: if $\bar{D}\Vdash\bar{\sigma}$ sat ϕ in the history-based semantics, then in the automaton based semantics, $\phi\stackrel{\bar{\sigma}}{\Longrightarrow}\phi'$, where $\phi'\checkmark$, so $\langle\phi',a\rangle\in\Phi$.

Finally, the case of EVAL-COMMIT is trivial. Recall the evaluation rule in the history based semantics: $\bar{\sigma}; \bar{D} \triangleright M, \mathtt{commit}\langle \bar{p}\rangle \to \bar{\sigma}, \bar{p}; \bar{D} \triangleright M$, and in the automaton-based semantics:

$$
\begin{array}{c}
\text{(EVAL-COMMIT)} \\
\bar{D} \vdash \Phi \overset{\bar{p}}{\Longrightarrow} \Psi \\
\hline
\mathcal{A}; \Phi; \bar{E} \triangleright \bar{b}, \mathtt{commit}\langle \bar{p}\rangle \\
\to \mathcal{A}; \Psi; \bar{E} \triangleright \bar{b}
\end{array}
$$

In this case, $\mathcal{T}(\bar{\sigma}; \bar{D}) = \mathcal{A}; \Phi; \bar{E}$, and $\mathcal{T}(\bar{\sigma}, \bar{p}; \bar{D}) = \mathcal{A}; \Phi'; \bar{E}$ where

$$
\mathcal{A} = \mathcal{P}_{\bar{\pi}} \left[\prod_{\mathsf{adv}\, a[\phi\alpha]\in\bar{D}} \iota(\phi, a) \right] \qquad \bar{E} = \mathcal{T}_{dec}(\bar{D})
$$

What remains is to show that $\Psi = \Phi'$. In doing so, we will have succeeded in showing that $\mathcal{A}, \Phi$, and $\bar{E}$ are all preserved by EVAL-COMMIT. By definition of $\mathcal{T}$, simulating $\bar{\sigma}$ on the intermediate automaton $\prod_{\mathsf{adv}\, a[\phi\alpha]\in\bar{D}} \iota(\phi, a)$ places $\mathcal{A}$ in state Φ. To derive Φ' from $\bar{\sigma}, \bar{p}; \bar{D}$, we simply carry the simulation one step further, taking transition $\bar{p}$. By EVAL-COMMIT in the automaton-based semantics, we know that $\Phi \overset{\bar{p}}{\Longrightarrow} \Phi'$, and hence that $\Psi = \Phi'$, which is what we needed to show. $\qquad \square$

PROPOSITION 6. *If* $\mathcal{A}; \Phi; \bar{E} \triangleright M \to \mathcal{A}'; \Phi'; \bar{E}' \triangleright M'$, *and* $\mathcal{T}(\bar{\sigma}, \bar{D}) = \mathcal{A}, \Phi, \bar{E}$, *then* $\bar{\sigma}; \bar{D} \triangleright M \to \bar{\sigma}'; \bar{D}' \triangleright M'$, *where* $\mathcal{T}(\bar{\sigma}', \bar{D}') = \mathcal{A}', \Phi', \bar{E}'$.

Proof. The proof closely parallels that of Proposition 5, and as such, we omit the details here. Details can be found in Appendix B.

This brings us to the main result: that the two semantics are equivalent:

THEOREM 7. $\bar{\sigma}; \bar{D} \triangleright M \to^* \bar{\rho}; \bar{E} \triangleright N$ *if and only if* $\mathcal{T}(\bar{\sigma}; \bar{D} \triangleright M) \to^* \mathcal{T}(\bar{\rho}; \bar{E} \triangleright N)$.

Proof. By Propositions 5 and 6, and induction on the length of $\to^*$. $\qquad \square$

7. Conclusions

We have described a novel minimal language for aspect-oriented programming with temporal pointcuts. We described an implementation of the language using security automata and proved the correctness of the implementation. We have presented examples of applications to software security.

Future work will address type-preserving translations of class-based languages into μABC. We have already developed untyped translations; finding type-preserving translations presupposes a suitable typing systems for μABC.

References

[1] Martín Abadi and Cedric Fournet. Access control based on execution history. In *Proceedings of the Network and Distributed System Security Symposium Conference*, 2003.

[2] Chris Allan, Pavel Avgustinov, Sascha Kuzins, Oege de Moor, Damien Sereni, Ganesh Sittampalam, Julian Tibble Aske Simon Christensen, Laurie Hendren, and Ondřej Lhoták. Adding trace matching with free variables to aspectj. In *OOPSLA 2005*, 2005.

[3] J. Andrews. Process-algebraic foundations of aspectoriented programming. In *In Reflection, LNCS 2192*, 2001.

[4] Steve Barker and Peter Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Transations on Information and System Security*, 6(4):501–546, 2003.

[5] Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Quantifying over dynamic properties of program execution. In *Dynamic Aspects Workshop (DAW05)*, 2005. Available at http://www.aosd.net/2005/workshops/daw/.

[6] Glen Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μABC: A minimal aspect calculus. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004: Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224, London, August 2004. Springer.

[7] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Submitted for publication, at ftp://ftp.ccs.neu.edu/pub/people/wand/papers/clw-03.pdf, oct 2003.

[8] Bruno De Fraine, Wim Vanderperren, Davy Suvée, and Johan Brichau. Jumping aspects revisited. In *Dynamic Aspects Workshop (DAW05)*, 2005. Available at http://www.aosd.net/2005/workshops/daw/.

[9] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *3rd Int. Conf. on Aspect-Oriented Software Development (AOSD'04)*, mar 2004.

[10] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns*, LNCS. Springer Verlag, September 2001. long version is http://www.emn.fr/info/recherche/publications/RR01/01-3-INFO.ps.gz.

[11] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*,

volume 1241 of *Lecture Notes in Computer Science*. Springer, June 1997.

[13] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, April 2002. ACM Press.

[14] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs.

[15] W. De Meuter. Monads as a theoretical foundation for aop. In *International Workshop on Aspect-Oriented Programming at ECOOP*, 1997.

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

[17] V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *RV'05 - Fifth Workshop on Runtime Verification*, 2005. To Appear.

[18] Peter Thiemann. Enforcing safety properties using type specialization. In *Programming Languages and Systems: 10th European Symposium on Programming, ESOP 2001*, volume 2028. Springer, 2001.

[19] David Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *Conference Record of AOSD 03: The 2nd International Conference on Aspect Oriented Software Development*, 2003.

[20] Úlfar Erlingsson and Fred Schneider. SASI enforcement of security policies: a retrospective. In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, pages 87–95, New York, NY, USA, 2000. ACM Press.

[21] David Walker. A type system for expressive security policies. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267, New York, NY, USA, 2000. ACM Press.

[22] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Conference Record of ICFP 03: The ACM SIGPLAN International Conference on Functional Programming*, 2003.

[23] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Twelfth International Symposium on the Foundations of Software Engineering*, 2004.

[24] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. appeared in Informal Workshop Record of FOOL 9, pages 67-88; also presented at FOAL (Workshop on Foundations of Aspect-Oriented Languages), a satellite event of AOSD 2002, 2002.

A. Semantics of Temporal Pointcuts

TEMPORAL POINTCUT SATISFACTION $\quad(\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi)$

(SAT-ATOM)

$$\frac{\bar{D} \vdash \sigma \text{ sat } \alpha}{\bar{D} \Vdash \sigma \text{ sat } \alpha}$$

(SAT-OR-LEFT)

$$\frac{\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi}{\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi + \psi}$$

(SAT-SEQ)

$$\frac{\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi \quad \bar{D} \Vdash \bar{\rho} \text{ sat } \psi}{\bar{D} \Vdash \bar{\sigma}, \bar{\rho} \text{ sat } \phi\psi}$$

(SAT-STAR)

$$\frac{\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi \quad \bar{D} \Vdash \bar{\rho} \text{ sat } \phi^*}{\bar{D} \Vdash \bar{\sigma}, \bar{\rho} \text{ sat } \phi^*}$$

(SAT-OR-RIGHT)

$$\frac{\bar{D} \Vdash \bar{\sigma} \text{ sat } \psi}{\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi + \psi}$$

(SAT-SEQ-EMPTY)

$$\frac{}{\bar{D} \Vdash \varepsilon \text{ sat } \varepsilon}$$

(SAT-STAR-EMPTY)

$$\frac{}{\bar{D} \Vdash \varepsilon \text{ sat } \phi^*}$$

B. Proof of Proposition 6

Again, in the cases of EVAL-DEC-ROLE and EVAL-ADV, this is trivial. In the case of EVAL-DEC-ADV, recall its evaluation rule:

$$\mathcal{A}; \Phi; \bar{E} \rhd \text{ adv } b[\psi\beta], M \to \mathcal{A}'; \Phi'; \bar{E}' \rhd M$$

where

$$\mathcal{A}' = \mathcal{A} \times \iota(\psi, b) \qquad \Phi' = \Phi, \psi \qquad \bar{E}' = \bar{E}, b$$

In the history-based semantics, we have

$$\bar{\sigma}; \bar{D} \rhd \text{ adv } b[\psi\beta], M \to \bar{\sigma}; \bar{D}, \text{adv } b[1^{|\bar{\sigma}|}\psi\beta] \rhd M$$

where $\mathcal{T}(\bar{\sigma}; \bar{D}) = \mathcal{A}; \Phi; \bar{E}$. What remains is to show that $\mathcal{T}(\bar{\sigma}; \bar{D}, \text{adv } b[1^{|\bar{\sigma}|}\psi\beta]) = \mathcal{A}'; \Phi'; \bar{E}'$, which we already proved in Proposition 5.

In the case of EVAL-CALL, if a $\text{call}\langle \bar{p} \rangle$ is replaced by the body of some advice $\text{adv } a[\phi\alpha]$, this must mean that advice a is associated with the current state of the automaton, and that $\bar{D} \vdash \bar{p} \text{ sat } \alpha$. We must show that in the history-based semantics, (*i*) the advice $\text{adv } a[\phi\alpha]$ is declared (trivial), (*ii*) that $\bar{D} \vdash \bar{p} \text{ sat } \alpha$ (given), and that (*iii*) $\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi$. Point (*iii*) follows directly from Lemma 4: since $\text{adv } a[\alpha]$ is associated with the current state, it must mean that $\phi \stackrel{\bar{\sigma}}{\Longrightarrow} \phi'$, and $\phi' \checkmark$. By Lemma 4, it immediately follows that $\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi$.

Finally, in the case of EVAL-COMMIT, recall its evaluation rule in the automaton-based semantics:

(EVAL-COMMIT)

$$\frac{\bar{D} \vdash \Phi \stackrel{\bar{p}}{\Longrightarrow} \Psi}{\begin{array}{c}\mathcal{A}; \Phi; \bar{E} \rhd \bar{b}, \text{commit}\langle \bar{p} \rangle \\ \to \mathcal{A}; \Psi; \bar{E} \rhd \bar{b}\end{array}}$$

If $\mathcal{A}; \Phi; \bar{E} \rhd \bar{b}, \text{commit}\langle \bar{p} \rangle \to \mathcal{A}; \Psi; \bar{E} \rhd \bar{b}$, then it must be the case that $\Phi \stackrel{\bar{p}}{\Longrightarrow} \Psi$. Now, let $\bar{\sigma}; \bar{D}$ be the history-based configuration such that $\mathcal{T}(\bar{\sigma}; \bar{D}) = \mathcal{A}; \Phi; \bar{E}$. Then by definition of $\mathcal{T}$,

$$\mathcal{A} = \mathcal{P}_{\bar{\pi}}\left[\prod_{\text{adv } a[\phi\alpha] \in \bar{D}} \iota(\phi, a) \right] \qquad \bar{E} = \mathcal{T}_{dec}(\bar{D})$$

Recall the rule in the history-based semantics:

$$\bar{\sigma}; \bar{D} \triangleright M, \texttt{commit}\langle \bar{p}\rangle \to \bar{\sigma}, \bar{p}; \bar{D} \triangleright M$$

We must show that $\mathcal{T}(\bar{\sigma}, \bar{p}; \bar{D}) = \mathcal{A}; \Phi'; \bar{E}$ where $\Phi' = \Psi$. $\mathcal{A}$ and $\bar{E}$ follow immediately from the definition of $\mathcal{T}$.

Furthermore, by definition of $\mathcal{T}$, simulating $\bar{\sigma}$ on the intermediate automaton $\prod_{\text{adv } a[\phi\alpha] \in \bar{D}} \iota(\phi, a)$ puts the automaton in state Φ. To derive Φ' from $\bar{\sigma}, \bar{p}; \bar{D}$, we simply carry the simulation one step further, taking transition $\bar{p}$. By EVAL-COMMIT in the automaton-based semantics, we know that $\Phi \overset{\bar{p}}{\Longrightarrow} \Phi'$, and hence that $\Phi' = \Psi$, which is what we needed to show. $\qquad\square$

C. Derived Forms

To give a feel for the language, we define a few derived forms. First, we provide an encoding of let, using roles for continuations. In this encoding we require an additional reserved role continue. The end value of a let expression must be explicitly marked with a return.

DERIVED FORMS (LET) (c fresh)

role $p \overset{\triangle}{=}$ role $p <$ top	Trivial Role
let $x = N$; $M \overset{\triangle}{=}$ role c; $\quad$ adv$[\langle c, +$top$\rangle] = (\_, x)\ M$; $\quad N\{^c/_{\text{continue}}\}$	Let
return $p \overset{\triangle}{=}$ call$\langle$continue, $p\rangle$	Return

For example, we have the following.

let $x = N$; let $y = L$; M
$\quad =$ role c;
$\qquad$ adv$[\langle c, +$top$\rangle] = (\_, x)$ let $y = L$; M;
$\qquad N\{^c/_{\text{continue}}\}$
$\quad =$ role c;
$\qquad$ adv$[\langle c, +$top$\rangle] = (\_, x)$ role d;
$\qquad\qquad\qquad\qquad\quad$ adv$[\langle d, +$top$\rangle] = (\_, y)\ M$;
$\qquad\qquad\qquad\qquad\quad L\{^d/_{\text{continue}}\}$;
$\qquad N\{^c/_{\text{continue}}\}$

The following encoding of functions uses triples to communicate the function name, argument and continuation.

DERIVED FORMS (FUNCTIONS) (f and x fresh)

$\lambda x. N \overset{\triangle}{=}$ role f; $\quad$ adv$[\langle f, +$top$, +$top$\rangle] = (\_, x, c)\ N\{^c/_{\text{continue}}\}$; $\quad$ return f	Abstraction
$L\ M \overset{\triangle}{=}$ let $f = L$; $\quad$ let $x = M$; $\quad$ call$\langle f, x, $continue$\rangle$	Application

For example, we have the following.

$(N\ L)\ M = $ let $f = N\ L$;
$\qquad\qquad\quad$ let $x = M$;
$\qquad\qquad\quad$ call$\langle f, x, $continue$\rangle$
$\qquad\quad = $ let $g = N$;
$\qquad\qquad\quad$ let $y = L$;
$\qquad\qquad\quad$ let $f = $callcc$\langle g, y\rangle$;
$\qquad\qquad\quad$ let $x = M$;
$\qquad\qquad\quad$ call$\langle f, x, $continue$\rangle$