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 [13], 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. [21] 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 [22-6, 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 \( \phi \) is a regular expression abstracting the program’s history, and \( \alpha \) 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 \texttt{Send} operations after a \texttt{FileRead} has been executed [16].

\[
\neg \text{call}(\texttt{FileRead})
\]

\[
\begin{tikzpicture}
  \node (qfr) at (0,0) [shape=circle,draw] {qfr};
  \node (qf) at (1,1) [shape=circle,draw] {qf, \{E_q\}};
  \draw [->] (qfr) -- node[above] {\text{call}(\texttt{FileRead})} (qf);
\end{tikzpicture}
\]

Our presentation differs slightly from that of [16] in that we attach an error handling aspect \( E_q \) to state \( q_f \). Its task is to watch for and intercept an attempted \texttt{call(Send)}. We attach an aspect to the state instead of transitioning into a new “error” state because transitions represent committed function calls—our intent is to block the \texttt{call(Send)}, 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 \texttt{call(\texttt{FileRead})}, and as a result, the automaton is in state \( q_f \). 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 \texttt{FileRead} is called. One possible automaton for this policy is shown below:

\[
\neg \text{call}(\texttt{FileRead})
\]

\[
\begin{tikzpicture}
  \node (rfr) at (0,0) [shape=circle,draw] {rfr};
  \node (rf) at (1,1) [shape=circle,draw] {r_f, \{E_r\}};
  \draw [->] (rfr) -- node[above] {\text{call}(\texttt{FileRead})} (rf);
\end{tikzpicture}
\]

Here, \( E_r \) is the error handling advice which monitors for a \texttt{call(\texttt{Login},A)} after seeing a \texttt{call(\texttt{FileRead})}. 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 \texttt{call(\texttt{Login},A)}. If we “play back” the program history \( (\text{call(\texttt{FileRead}),call(\texttt{Login},A)}) \) on the product automaton, advice \( \{E_r\} \) will fire, which is incorrect according to our interpretation—the quarantine policy was implemented after the \texttt{call(\texttt{FileRead})}. 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 X for an ordered sequence of X’s.

We define a polyadic variant of μABC, introduced in [5]. The earlier paper followed the style of object-oriented languages; each message “p − q : ℓ” 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, (p₁, . . . , pₙ) (equivalently (p)) , with triadic messages as a special case “(p, q, ℓ)”.

For simplicity, in this paper, we look at a single-threaded variant. At each moment in runtime, there is a single event (p) 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 a, which is waiting to process the event. Thus a₁, . . . , aₙ(p) indicates that advices named aᵢ are waiting to process event p. We say that aᵢ advised p, and that a(p) is an advised event. For consistency with the precedence of declarations, we read the advice list from right to left; thus aₙ is the first advice to process the event.

The special advice call initiates advice lookup. When call(p) executes, all the advice triggering on (p) is listed, resulting in a new execution state: a(p). To determine whether an advice is triggered, we use the pointcut a. 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(α) = u( x) N is parameterized both on the event 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 D ⊢ (p) sat α, described in the next subsec-
syntax requires that "\( \bar{b}, a(\bar{p}) \)" be parsed as "\((\bar{b}, a)(\bar{p})\)". Further note that the substitution \( \bar{y}/y \) results in a well-formed term because free advice names can only appear in the context of a sequence.)

**Evaluation**  
\( \bar{D} \triangleright M \rightarrow \bar{E} \triangleright N \)

\[
\begin{align*}
\text{(eval-call)} & \quad \bar{D} \triangleright E; M \rightarrow \bar{D}, E \triangleright M \\
\text{(eval-adv)} & \quad \bar{D} \triangleright adv \alpha \triangleright u(\bar{x}) \triangleright N \\
\text{(eval-dec)} & \quad \bar{D} \triangleright \bar{b}, call(\bar{p}) \rightarrow \bar{D} \triangleright \bar{b}, \bar{a}(\bar{p}) \\
\text{(eval-adv)} & \quad \bar{D} \triangleright \bar{b}, a(\bar{p}) \rightarrow \bar{D} \triangleright N\{\bar{y}/y, \bar{x}/x\}
\end{align*}
\]

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**

\[
P, Q ::= \quad \text{Role Pattern} \\
\rho & \quad \text{Exact Role} \\
p & \quad \text{Sub Role} \\
\bar{\alpha}, \bar{\beta} ::= \quad \text{Atomic Event Pointcut} \\
\bar{P} & \quad \text{Call Event} \\
\bar{P}, * & \quad \text{Call Event, varargs} \\
\alpha \lor \beta & \quad \text{Disjunction} \\
\neg \alpha & \quad \text{Negation} \\
\sigma, \rho ::= \quad \text{Atomic Event}
\]

Define 1 as \( * \); define 0 as \( \neg 1 \); and define \( \alpha \lor \beta \) as \( \neg(\neg \alpha \lor \neg \beta) \). We write \( \bar{D} \triangleright r \prec \bar{p} \) 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**  
\( \bar{D} \triangleright \sigma \triangleright u(\bar{x}) \triangleright N \)

\[
\begin{align*}
\text{(sat-call-any)} & \quad \bar{D} \triangleright (\bar{q}) \triangleright (\bar{p}) & \quad \text{(sat-call-empty)} & \quad \bar{D} \triangleright (\bar{p}) \triangleright (\bar{q}) \\
\text{(sat-call-exact)} & \quad \bar{D} \triangleright (\bar{q}) \triangleright (\bar{p}) & \quad \text{(sat-call-sub)} & \quad \bar{D} \triangleright (\bar{p}) \triangleright (\bar{q}) \\
\bar{D} \triangleright (\bar{r}, \bar{q}) & \quad \text{sat} (\bar{r}, \bar{Q}) & \quad \bar{D} \triangleright (\bar{q}) & \quad \text{sat} (\bar{Q}) & \quad \bar{D} \triangleright r \leq p \\
\end{align*}
\]

4. **Temporal Pointcuts**

We extend \( \mu \text{ABC} \) with temporal pointcuts. To do this, we modify the language of advice to include a temporal formula \( \phi \) in addition to the atomic formula \( \alpha \). Intuitively, the pointcut fires when \( \phi \) matches the past and \( \alpha \) 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 \( \phi \) and the potential future \( \alpha \). The past is specified as a regular expression over atomic event pointcuts; the potential future is specified as an atomic event pointcut.

**Syntax**

\[
\begin{align*}
D, E & ::= \cdots \\
\text{Decl} & ::= \text{Declare Advice} \\
\text{adv} & ::= \phi \triangleright u(\bar{x}) \triangleright N \\
\phi, \psi, \chi & ::= \quad \text{Temporal Pointcuts} \\
\alpha & ::= \quad \text{Atomic Event Pointcut} \\
\epsilon & ::= \quad \text{Empty Sequence} \\
\phi^* & ::= \quad \text{Kleene Star} \\
\phi + \psi & ::= \quad \text{Disjunction} \\
\sigma, \rho & ::= \quad \text{Atomic Events}
\end{align*}
\]

The semantics of temporal formulas \( \bar{D} \triangleright \sigma \triangleright u(\bar{x}) \triangleright N \) is defined in the standard way (recalled in Appendix A) over strings of events, building on the semantics of atomic events \( \bar{D} \triangleright \sigma \triangleright u(\bar{x}) \triangleright N \). Note that the regular expression \( \theta \) is represented here as the atomic event pointcut \( 0 \). We define the language of the formula as follows: \( L_{H}(\bar{D}, \phi) = \{ \sigma \mid \bar{D} \triangleright \sigma \triangleright u(\bar{x}) \triangleright \phi \} \).

We now give the evaluation semantics for the language with temporal advice. We augment the semantics to record an execution history. We write \( \mid \sigma \) for the length of string \( \sigma \). We define \( \alpha^{\epsilon} = \alpha \) and \( \alpha^{0} = \epsilon \), where \( \alpha^{\epsilon} \). We write "\( \text{adv} a[\alpha] = u(\bar{x}) \triangleright N \)" as shorthand for "\( \text{adv} a[\alpha] = u(\bar{x}) \triangleright N \)". 

**Evaluation**  
\( \sigma; \bar{D} \triangleright M \rightarrow \bar{p}; \bar{E} \triangleright N \)

\[
\begin{align*}
\text{(eval-dec-role)} & \quad \sigma; \bar{D} \triangleright M \rightarrow \bar{p}; \bar{E} \triangleright N \\
\text{(eval-dec-adv)} & \quad \sigma; D \triangleright \bar{b}, \text{commit}(\bar{p}) \rightarrow \sigma; \bar{p}; \bar{D} \triangleright \bar{b} \\
\text{(eval-call)} & \quad \bar{D} \triangleright \bar{p} \rightarrow \sigma; \bar{D} \triangleright \bar{a}(\bar{p}) \\
\text{(eval-adv)} & \quad \bar{D} \triangleright \bar{b}, \alpha(\bar{p}) \rightarrow \sigma; \bar{D} \triangleright N\{\bar{y}/y, \bar{x}/x\}
\end{align*}
\]

\[1\] In the full version we also allow vararg parameters in advice declarations, ie \( \text{adv} a[\alpha] = u(\bar{x}, \star) \triangleright N \).
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 $\text{adv } a[\phi \alpha]$, where $\phi$ is a regular expression abstracting the relevant events in the program history, and $\alpha$ is the triggering atomic event. For each advice $\text{adv } a[\phi \alpha]$, we construct the automaton for $\phi$. 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 $\alpha$ 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:

<table>
<thead>
<tr>
<th>Advice States $(\phi \check{\alpha})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi \check{\alpha}$</td>
</tr>
<tr>
<td>$\epsilon \check{\alpha}$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Transition Relation $(\phi \xrightarrow{\alpha} \psi)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi \xrightarrow{\alpha} \phi'$</td>
</tr>
<tr>
<td>$\psi \xrightarrow{\alpha} \psi'$</td>
</tr>
<tr>
<td>$\epsilon \xrightarrow{\alpha} \epsilon'$</td>
</tr>
</tbody>
</table>

Transitions between states are taken on commits.

We write $\phi \overset{\alpha}{\rightarrow}$ for the reflexive transitive closure of $\rightarrow$.

Next, we formally state how to derive an automaton from an advice $\text{adv } a[\phi \alpha]$:

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

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

**AUTOMATON SYNTAX**

$$
\Phi, \Psi ::= \phi \mid \phi, \Phi
$$

$$
\mathcal{A} ::= (\Phi, \Psi, \mathcal{A})
$$

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

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

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 $\langle\phi_1,\psi_0\rangle \xrightarrow{\text{call(FileRead)}} \langle\phi_1,\psi_1\rangle$. To make the presentation more readable, we elide advice when a state has none associated with it. That is, we write the state “$\langle\phi,\emptyset\rangle$” simply as $\phi$.

We can modulate the transition relation from atomic event pointcuts to atomic events: define $D \vdash \phi \xrightarrow{\alpha} \phi'$ if $\phi \triangleleft \sigma \triangleleft \alpha \vdash \phi$. Further we can lift the definition to automaton states: $D \vdash \phi_0, \ldots, \phi_n \overset{\alpha}{\rightarrow} \psi_1, \ldots, \psi_n$ if $D \vdash \phi_i \xrightarrow{\alpha} \psi_i$ for all $i$ between 1 and $n$. Finally we lift the resulting relation $(D \rightarrow \Phi \xrightarrow{\alpha} \Psi)$ to event sequences: $D \vdash \phi_0, \ldots, \phi_n \overset{\alpha}{\rightarrow} \Phi_n$ if $D \vdash \phi_{i-1} \overset{\alpha}{\rightarrow} \phi_i$ 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 $\text{adv } a[\phi \alpha]$ with an existing automaton $\mathcal{A}$. Namely, we construct the automaton for the advice, and create the product automaton:

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

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.
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 \( \rho \) being called. If a matching advice is found, then the call \( \bar{p} \) 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 2 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 \( \langle \sigma, D \rangle \) to an automaton-based configuration \( \langle A, \Phi, E \rangle \) as follows: given a history \( \sigma \) and a set of declarations \( D \), we first construct an intermediate automaton \( A' \) using the aspect declarations in \( D \). We compute the state \( \Phi \) by simulating the history \( \sigma \) on \( A' \). Finally, we convert the intermediate automaton \( A' \) to the final automaton \( A \) 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 \( \text{adv} \, a[\phi[\alpha]] \) is declared at time \( n \), then in the history-based semantics, the pointcut is noted as \( \text{adv} \, a[1^n \phi[\alpha]] \), and the corresponding automaton in the automaton-based semantics will have a string of \( n \) “placeholder” states \( \pi_1, \ldots, \pi_n \), where \( \pi_i \overset{1}{\rightarrow} \pi_{i+1} \) for \( i \) between 1 and \( n - 1 \), and \( \pi_n \overset{1}{\rightarrow} \phi \), as shown below:

![Diagram](https://via.placeholder.com/150)

**CONVENTION.** In constructing an automaton for an advice \( \text{adv} \, a[\phi[\alpha]] \) declared at time \( n \), we label the states used as placeholders for time \( 1 \) through \( n \) as \( \pi_0, \ldots, \pi_n \), and we refer to these as \( \pi \)-states.

Strictly speaking, we must account for the fact that for an advice \( \text{adv} \, a[\phi[\alpha]] \), \( \phi \) may in fact begin with a string of leading \( 1 \)s. We can easily get around this by syntactically differentiating between those \( 1 \)s 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 \( \Phi = \langle \phi_1, \psi_1, \ldots, \chi_i \rangle \) is such that none of \( \phi_1, \psi_1, \ldots, \chi_i \) are \( \pi \)-states, we say that \( \Phi \) is \( \pi \)-free. We will need to project the \( \pi \)-free states of an automaton, so we formally define this operation:

\[
P_{\pi}(A) = \{ \langle \Phi, \bar{a} \rangle \in A | \Phi \text{ contains no } \pi \text{ states} \}
\]

**LEMMA 2.** For two automata \( A \) and \( B \), \( P_{\pi}(A \times B) = P_{\pi}(A) \times P_{\pi}(B) \)

**Proof.** Immediate.

To construct \( A' \), we take the product of the automata induced by each advice in \( D \). To construct \( \Phi \), we simulate the program history \( \sigma \) on \( A' \). For instance, in the example in Section 2, \( A' \) is the product of the following automata, with \( \phi_0, \phi_1, \psi_0 \), and \( \psi_1 \) defined as in Section 5:

\[
\begin{align*}
\text{call}(\text{FileRead}) & \\
\phi_0 & \quad \phi_1, (E_1) & 1 \\
\text{call}(\text{FileRead}) & \\
\psi_0 & \quad \psi_1, (E_2) & 1
\end{align*}
\]

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

We compute \( A \) by removing from \( A' \) any states containing a \( \pi \) state. In our example, this amounts to removing from the product automaton states \( \langle \langle \phi_0, \pi_0 \rangle \rangle \) and \( \langle \langle \phi_1, \pi_0 \rangle, \{E_1\} \rangle \). The result is equivalent to the product automaton \( A_Q \times A_R \), where \( A_Q \) and \( A_R \) are as in Section 2.
We now formalize the translation just discussed. That is, given a history, declaration pair \((\vec{\sigma}, \vec{D})\), we formally show how to construct the corresponding automaton, state, declaration triple \((A, \Phi, \vec{E})\).

Our translation makes use of the following functions:

\[
T_{\text{dec}}(\vec{D}) = \{ \text{adv } a(\vec{a})| a \in D \} \\
T_{\text{state}}(\vec{\sigma}, \vec{D}) = \Psi,
\]

where

\[
\vec{D} = \langle \text{adv } \bar{a}, \bar{\Phi}_1, \bar{\Psi}_1, \ldots, \text{adv } \bar{a}, \bar{\Phi}_n, \bar{\Psi}_n \rangle \\
\bar{\Phi} = (\bar{\Phi}_1, \ldots, \bar{\Phi}_n)
\]

We are now in a position to define the function \(T\) which translates a history-based configuration \((\vec{\sigma}, \vec{D})\) to an automaton-based configuration \((A, \Phi, \vec{E})\):

**Definition 3.** \(T(\vec{\sigma}; \vec{D}) = A; \Phi; \vec{E}\), where

\[
A = \sum_{a|a \in D} t(\phi, a) \\
\Phi = T_{\text{state}}(\vec{\sigma}, \vec{D}) \\
\vec{E} = T_{\text{dec}}(\vec{D})
\]

We define the language of the formula as follows:

\[
L_A(\vec{D}, \phi) = \{ |D| \phi \Rightarrow \phi', \phi \vdash \phi' \}
\]

**Lemma 4.** For all \(D\) and \(\phi\), \(L_A(\vec{D}, \phi) = L_A(\vec{D}, \phi)\).

**Proof.** By induction on the structure of \(\vec{\sigma}\).

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

- \(\vec{\sigma}; \vec{D} \vdash M \rightarrow \vec{\sigma}'; \vec{D}' \vdash M'\)
- \(T(\vec{\sigma}, \vec{D}) = A; \Phi; \vec{E}\)
- \(A; \Phi; \vec{E} \vdash M \rightarrow A'; \Phi'; \vec{E}' \vdash M'\)
- \(T(\vec{\sigma}', \vec{D}') = A''; \Phi''; \vec{E}''\)

then \(A' = A'', \Phi' = \Phi''\), and \(\vec{E}' = \vec{E}''\), as shown below:

\[
\begin{array}{ccc}
\vec{\sigma}; \vec{D} & \vdash \vec{\sigma}'; \vec{D}' \\
T & \downarrow & T \\
A; \Phi; \vec{E} & \vdash A'; \Phi'; \vec{E}'
\end{array}
\]

**Proposition 5.** If \(\vec{\sigma}; \vec{D} \vdash M \rightarrow \vec{\sigma}'; \vec{D}' \vdash M'\) and \(T(\vec{\sigma}, \vec{D}) = A; \Phi; \vec{E}\), then \(A; \Phi; \vec{E} \vdash M \rightarrow A'; \Phi'; \vec{E}' \vdash M'\), where \(T(\vec{\sigma}', \vec{D}') = A'; \Phi'; \vec{E}'\).

**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-DECL-ADV) to the automaton to obtain the next configuration \((A', \Phi', \vec{E}')\). We then translate the right hand side into the automaton based semantics and show that the result equals \((A', \Phi', \vec{E}')\).

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

\[
\vec{\sigma}; \vec{D} \models \text{adv } b[\psi | b|] \rightarrow \vec{\sigma}; \vec{D} \models \text{adv } b[\psi | b|] \rightarrow M
\]

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

\[
A \models \Phi = \prod_{a|a \in D} t(\phi, a) \\
\vec{E} = T_{\text{dec}}(\vec{D}), \phi \models \phi
\]

By Eval-DECL-ADV in the automaton semantics, \(A; \Phi; \vec{E} \models \text{adv } b[\psi | b|] \rightarrow A'; \Phi'; \vec{E}' \rightarrow M\), where

\[
A' = \sum_{a|a \in D} t(\phi, a) \\
\phi' \models \phi
\]

Finally, we must show that \(T(\vec{\sigma}', \vec{D}') \models \text{adv } b[\psi | b|] = A'; \Phi'; \vec{E}'\). By definition, \(T(\vec{\sigma}', \vec{D}') \models \text{adv } b[\psi | b|] = A'; \Phi'; \vec{E}'\), where

\[
A' = \prod_{a|a \in D} t(\phi, a) \\
\phi' \models \phi
\]

and hence that the automaton is preserved by Eval-DECL-ADV.

To show that the state \(\Phi\) is preserved by Eval-DECL-ADV, we simulate \(\vec{\sigma}\) on the intermediate automaton \(\prod_{a|a \in D} t(\phi, a) \times t(\psi | b|)\). It immediately follows that the resulting state \(A' = (\Phi, \psi) = \Phi'\). The declarations \(\vec{E}\) are trivially preserved by Eval-DECL-ADV.

We now consider the case of Eval-CALL. We must show that in a history-based configuration \((\vec{\sigma}; \vec{D})\), for any declared advice \(a|a|\), if \(\vec{D} \models \vec{\sigma}\) sat \(\phi\) and \(\vec{D} \models \tilde{\rho}\) sat \(\alpha\) where \(\tilde{\rho}\) is the role vector being called, then \(\text{adv } a(\vec{a})\) is associated with the state \(\Phi\) in \(T(\vec{\sigma}, \vec{D})\). This follows directly from Lemma 2 if \(\vec{D} \models \vec{\sigma}\) sat \(\Phi\) in the history-based semantics, then in the automaton based semantics, \(\phi \models \phi'\), so \(\phi', \tilde{\rho} \models \phi\).
Finally, the case of EVAL-COMMIT is trivial. Recall the evaluation rule in the history based semantics: \( \bar{\sigma}; \bar{D} \triangleright M. \) commit(\( \bar{p} \)) \( \rightarrow \bar{\sigma}; \bar{p}; \bar{D} \triangleright M. \) and in the automaton-based semantics:

\[
\begin{align*}
\text{(EVAL-COMMIT)} \\
\bar{A}; \Phi; \bar{E} \triangleright \bar{h}, \text{commit(} \bar{p} \text{)} \\
\rightarrow \bar{A}; \Psi; \bar{E} \triangleright \bar{h}
\end{align*}
\]

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

\[
\bar{A} = P_x \left[ \prod_{a \in \text{adv}} t(\phi, a) \right] \\
\bar{E} = \bar{t}_{\text{dev}}(\bar{D})
\]

What remains is to show that \( \Psi = \Phi' \). In doing so, we will have succeeded in showing that \( \bar{A}, \Phi, \) and \( \bar{E} \) are all preserved by EVAL-COMMIT. By definition of \( T \), simulating \( \bar{\sigma} \) on the intermediate automaton \( \prod_{a \in \text{adv}} t(\phi, a) \) places \( \bar{A} \) in state \( \Phi \). To derive \( \Phi' \) 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 \( \Phi \rightleftharpoons \Phi' \), and hence that \( \Psi = \Phi' \), which is what we needed to show.

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

**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.** \( \bar{\sigma}; \bar{D} \triangleright M \rightarrow^* \bar{p}; \bar{E} \triangleright N \) if and only if \( T(\bar{\sigma}; \bar{D} \triangleright M) \rightarrow^* T(\bar{\sigma}; \bar{E} \triangleright N) \).

**Proof.** By Propositions 5 and 6 and induction on the length of \( \rightarrow^* \).

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 \( \mu\text{ABC} \). We have already developed untyped translations; finding type-preserving translations presupposes a suitable typing systems for \( \mu\text{ABC} \).

Acknowledgments

The comments of the anonymous referees were helpful in sharpening the focus of the paper. Alan Jeffrey and Radha Jagadeesan contributed to early discussions of this work.

References


A. Semantics of Temporal Pointcuts

**Temporal Pointcut Satisfaction** \( (\bar{D} \models \sigma \text{ sat } \alpha) \)

\[
\begin{align*}
\text{(SAT-ATOM)} & \quad \bar{D} \models \sigma \text{ sat } \alpha \\
\text{(SAT-SEQ)} & \quad \bar{D} \models \sigma \text{ sat } \alpha + \psi \\
\text{(SAT-STAR)} & \quad \bar{D} \models \sigma \text{ sat } \alpha^* \\
\text{(SAT-SEQ-EMPTY)} & \quad \bar{D} \models \epsilon \text{ sat } \epsilon \\
\text{(SAT-STAR-EMPTY)} & \quad \bar{D} \models \epsilon \text{ sat } \epsilon^*
\end{align*}
\]

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:

\[\mathcal{A}; \Phi; E \triangleright \text{adv } b[\psi \beta], M \rightarrow \mathcal{A}'; \Phi'; E' \triangleright M\]

where

\[\mathcal{A}' = \mathcal{A} \times t(\psi, b) \quad \Phi' = \Phi \cup \psi \quad E' = E \cup b\]

In the history-based semantics, we have

\[\bar{\sigma}; \bar{D} \triangleright \text{adv } b[\psi \beta], M \rightarrow \bar{\sigma}; \bar{D}, \text{adv } b[1^{|\bar{\sigma}|} \psi \beta] \triangleright M\]

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

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

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

\[
\begin{align*}
\text{(EVAL-COMMIT)} & \quad \bar{D} \models \Phi \rightarrow \Psi \\
& \quad \mathcal{A}; \Phi; E \triangleright b, \text{commit}(\bar{p}) \\
& \quad \rightarrow \mathcal{A}; \Psi; E \triangleright b
\end{align*}
\]

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

\[\mathcal{A} = T_{\text{dec}}(\bar{D})\]

Recall the rule in the history-based semantics:

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

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

C. Derived Forms

To give a feel for the language, we define a few derived forms and discuss their execution.

This is an encoding of let that uses roles for continuations. In this encoding we require an additional reserved role continue.

**Derived Forms (Functions) (f and x fresh)**

- $\lambda x. N \triangleq \text{role } f$; \hspace{1cm} Abstraction
  
  $\text{adv}[\langle f, \text{top}, \text{top} \rangle] = (\_, x, c) \, N \{ \text{'continue} \}$
  
  $\text{ret } f$

- $L \, M \triangleq \text{let } f = L; \text{let } x = M; \text{ call } \langle f, x, \text{continue} \rangle$ \hspace{1cm} Application

For example, we have the following.

$(N \, L) \, M = \text{let } f = N \, L; \text{ let } x = M; \text{ call } \langle f, x, \text{continue} \rangle$

---

**Derived Forms (LET) (c fresh)**

<table>
<thead>
<tr>
<th>role $p \triangleq \text{role } p &lt; \text{top} $</th>
<th>Trivial Role</th>
</tr>
</thead>
</table>

let $x = N; \, M \triangleq \text{role } c$; \hspace{1cm} Let

$\text{adv}[\langle c, \text{top} \rangle] = (\_, x) \, M$;

$N \{ \text{'continue} \}$

ret $p \triangleq \text{call } \langle \text{continue}, p \rangle$ \hspace{1cm} Return

$M; N \triangleq \text{let } \_ = N; \, M$ \hspace{1cm} Sequencing

For example, we have the following.

let $x = N; \text{ let } y = L; \, M$

$= \text{role } c$;

$\text{adv}[\langle c, \text{top} \rangle] = (\_, x) \, L \, M$;

$N \{ \text{'continue} \}$

$= \text{role } c$;

$\text{adv}[\langle c, \text{top} \rangle] = (\_, x) \, \text{role } d$;

$\text{adv}[\langle d, \text{top} \rangle] = (\_, y) \, M$;

$L \{ d \text{'continue} \}$;

$N \{ \text{'continue} \}$

For example, we have the following.

let $x = \langle \text{let } y = L; N \rangle; \, M$

$= \text{role } c$;

$\text{adv}[\langle c, \text{top} \rangle] = (\_, x) \, M$;

$\text{let } y = L; \, N \{ \text{'continue} \}$

$= \text{role } c$;

$\text{adv}[\langle c, \text{top} \rangle] = (\_, x) \, \text{role } d$;

$\text{role } d$;

$\text{adv}[\langle d, \text{top} \rangle] = (\_, y) \, L \{ d \text{'continue} \}$;

$N \{ d \text{'continue} \}$