Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs

Raffi Khatchadourian\textsuperscript{1*}, Johan Dovland\textsuperscript{2}, and Neelam Soundarajan\textsuperscript{1}

\textsuperscript{1}The Ohio State University, USA
\textsuperscript{2}University of Oslo, Norway
*Partially administered during visit to Lancaster University, UK
• AOP enables modular implementation of cross-cutting concerns.

• Both formal and informal reasoning about AOP presents unique challenges especially in respect to evolution.

• As components enter, exit, and re-enter software, conclusions about behavior of components may be invalidated.
• Desire a *compositional* reasoning approach, however the invasive nature of AOP makes this difficult.

• In the worst case, changes made to a single component require reexamining the entire program.
Questions

• Can we draw meaningful conclusions about component code *without* considering the *actual* advice code?

• Can we specify the behavior of components without any particular advice in mind?

• Can we *parameterize* specifications over *all* possibly applicable aspects?

• Can we suitably constrain the behavior of aspects as the software evolves?
Using interface is one answer (e.g., XPIs, Open Modules)

But it would be nice to have a way to derive the *enriched* behavior of the base plus the aspects at compile time.
• AO programs inherently enjoy *plug-n-play* capabilities [Laddad03]

• Crosscutting features can be *plugged-in* to *enrich* the behavior of advised components.

• Likewise, can we specify components so that we can derive their behaviors in a similar fashion?
Behavior of $C + A_1$

Behavior of $C + A_1 + A_2$

Behavior of $C + A_2$
Obstacles

● **Usefulness**
  ● Is it possible to draw meaningful conclusions from such incomplete information?

● **Obliviousness**
  ● Specifications contain “slots” for applications of crosscutting concerns.
• Abstraction
  • Competing forces:
    • Specs abstract internal details components, aspects directly manipulate them.

• Composition
  • Which pegs go into which holes?
  • How to deal with dynamic and lexical pointcuts?

• Complexity
  • What if no advice is applicable?
• May need to make assumptions about the behavior of evolving components.

• **Specification pointcuts**
  
  • *Pointcut interfaces* [Gudmundson01] annotated with behavioral specifications.
  
  • “Exported” internal semantic events within the component.

• Adopt a *rely-guarantee* approach [Xu97] from concurrent programming to constrain the behavior of all possibly applicable advice using a *rely* clause.

• A *guar* clause may be used to constrain components.
$\sigma$ the set of all variables of the program

$\sigma_i, \sigma_j, \ldots$ states in which each variable has a particular value
\( \sigma_1 \)

\( \sigma'_2 \)

\( \sigma_3 \)

Aspect
The state at a point in the execution of a component is $\sigma_a$.

$\text{rely}(\sigma_a, \sigma_b)$

The state when the class gets control back from an aspect is $\sigma_b$. 
This is “Harmless” [D&W POPL’06]

\[ \text{rely}(\sigma, \sigma') \equiv (\sigma = \sigma') \]

Forbids any applicable advice from making any changes in the state!

The entire state of \( \mathcal{C} \)
• Constraining parameterized behavior reduces complexity, but ...
  • How are *formal* parameters expressed?
  • How are *actual* parameters deduced?
  • How are the specifications *composed*?
• Aspects are typically used to *enrich* the behavior of the an underlying component.
• Thus, we want to deriving the *actual* behavior of components with the aspects.
A Join Point Trace (JPT) variable is introduced to track the flow-of-control through various join points within a component.

A JPT is used as a parameter over the actions of all possibly applicable aspects.

Method post-conditions will references to the JPT.

Informally, a JPT is used to refer to the actions and resulting values taken by advice at certain join point.
• The JPT is composed of several components that are associated with each join point.

• Just as there are different kinds of join points (e.g., call, execution), there are different kinds of JPT entries.
JPT Method Call Completion Element

Called Method

Argument Values

State Vectors

\((oid, mid, aid, args, res, \sigma, \sigma')\)

\(\sigma[oid]\)  Called Object

\(\sigma'[oid]\)  State of object \(oid\) after completion of method \(mid\)

\(\sigma'[aid]\)  State of object \(oid\) after completion of aspect \(aid\)

No applicable advice \(\implies \sigma = \sigma'\)
Rule for method specification

\[ C.m:: \langle \text{pre}, \text{post}, \text{guar}(), \text{rely}() \rangle \]

- Normal pre-condition
- Post-condition, may include references to portions of JPT
- R/G Clauses
Rule for method specification

\[ \textit{pre} \land [\lambda \tau = \langle \textit{inv}, \texttt{C.m} \rangle] \Rightarrow p \]

Invocation of \texttt{C.m} on the \textit{local JPT}

\texttt{C.m} :: \langle \textit{pre}, \textit{post}, \textit{guar}(), \textit{rely}() \rangle
Rule for method specification

\[ \text{pre} \land [\lambda \tau = \langle \text{inv}, \text{C.m} \rangle] \Rightarrow p \]
\[ \{p\} S\{q\} \]

\[ \text{C.m} :: \langle \text{pre, post, guar()}, \text{rely()} \rangle \]
Rule for method specification

Don’t forget about the guarantee

\[ pre \land [\lambda \tau = \langle (inv, C.m) \rangle] \Rightarrow p \]
\[ \{ p \} S\{ q \} \]
\[ q \Rightarrow guar() \]
Rule for method specification

\[
\begin{align*}
&\text{pre} \land [\lambda \tau = \langle \text{inv, C.m} \rangle] \Rightarrow p \\
&\{p\} S\{q\} \\
&q \Rightarrow \text{guar}() \\
&q \land \text{rely}(\sigma, \sigma') \Rightarrow
\end{align*}
\]

\[C.m :: \langle \text{pre, post,} \rangle\]

If when \( q \) holds and applicable advice behaves properly implies that ...

25
Rule for method specification

Not sure which aspect is applicable yet, so we’ll leave this blank

Replace all occurrences of $\sigma$ with $\sigma'$

... our post-condition holds with a a new entry in the local JPT
\{ p \} \text{ob.m(args)} \{ q \}
Rule for Method Calls

Substitute actuals for formals

\[ p \Rightarrow C.m.pre[pars/args] \]

\[ \{ p \} ob.m(args) \{ q \} \]
Rule for Method Calls

\[
p \Rightarrow C.m.\text{pre}[p]
\]
\[
C.m.\text{post} \Rightarrow q[\lambda \tau_1/\lambda \tau_1 \hat{\lambda} \tau_2, \text{args/pars}]
\]
\[
\{ p \}
\]

Local JPT for caller

Substitute formals for actuals
Rule for Aspect Application (Simple)

\[ \{pre \land ap \} \ C.m() + A \{post \land aq \} \]
Rule for Aspect Application (Simple)

\[
\begin{align*}
\{ \text{guar} (\sigma) \land \text{ap} \} & \mathcal{A}_{\text{adv}} \{ \text{rely}[\sigma/\sigma@\text{pre}, \sigma'/\sigma] \land \text{aq} \} \\
\{ \text{pre} \} & \mathcal{A}_{\text{post}} \{ \text{post} \land \text{aq} \}
\end{align*}
\]

Base-code satisfies \textit{guar}

Advice body

State vector immediately prior to the execution of the advice
\[
\{ \text{guar}(\sigma) \land ap \} \ A_{adv} \ \{ \text{rely}[\sigma/\sigma\@pre, \sigma'/\sigma] \land \text{aq} \} \\
\text{C.m} :: \langle \text{pre, post, guar, rely} \rangle \\
\frac{\{ \text{pre} \land ap \} \ \text{C.m}() + A \ \{ \text{post} \land \text{aq} \}}{}
\]
On-going work (hopefully thesis worthy! ;) )

Complete formal model (suggestions here?)

Sound axiomatic proof system

Curbing notational complexity via predicates.

Integration with IDE/theorem provers.

Complement the Eclipse AJDT with a *behavioral* cross reference view?

Integration with languages (e.g., via annotated pointcuts, JML)