On the Expressive Power of Classpects *

Hridesh Rajan
University of Virginia
hr2j@cs.virginia.edu

Kevin Sullivan
University of Virginia
sullivan@cs.virginia.edu

Abstract

The most prominent feature of AspectJ and derivative languages is the *aspect*. The aspect is a module-like construct that supports data abstraction but that is distinct from the class in several ways. First, aspects support new mechanisms: notably join points, pointcut descriptors and advice. At the same time, they lack some key features of classes, namely the ability to manage aspect instantiation using *new*. In earlier work [23, 25], we showed that it was possible and valuable to simplify the design of such languages without loss of expressiveness by unifying aspects and classes in a new construct that we called the *classpect*. The contribution of this paper is the formal demonstration, using Felleisen's notion of macro–eliminable programming language extensions [11], that the classpect language model, realized in the Eos language and compiler, is strictly more expressive than AspectJ in the important case in which aspect-like modules must advise other aspect modules.

1. Introduction

Aspect-oriented programming (AOP) has shown the potential to improve the ability of software architects to devise more effective modularizations for some traditionally non-modular concerns. The dominant family of languages in this domain is based on the AspectJ model [3, 7, 14, 16, 22, 28] or what Masuhara and Kiczales would call a Pointcut-Advice Model [21]. In this work, we study this model. The designs of AspectJ-like languages are unique in having shown that AOP can succeed in practice. Their success is due in part to the careful balancing of competing pressures in the design of the language. A major goal was industrial adoption. The pursuit of this goal led the designers of AspectJ to make important early design decisions that in some cases traded against orthogonality, uniformity and generality [20] in the language design in order to make the new constructs more acceptable to potential early adopters. Now that the model has shown promise, it makes sense to ask whether we can improve upon it by the application of more traditional principles of programming language design [20, 33].

One specific design decision that was the subject of our earlier work [25] was to introduce aspects as a new concept and as a separate abstraction mechanism. AspectJ-like languages [16] support two related but distinct constructs for modular design: classes and aspects. Aspects are promoted as means to modularize traditionally non-modular concerns such as execution tracing policy, cross-module optimization, use of common thread pool, security policy enforcement, component integration, etc. Generally, AspectJ programs are designed in a two-layered style: a *base* layer, typically of object-oriented code, is advised and extended by an aspect layer.

In previous work [25], we showed that some of the properties of aspects in the AspectJ style make it hard to go beyond such a two-layered style. While aspects can advise classes, classes cannot advise aspects, and aspects cannot advise other aspects in full generality. On one hand introduction of the aspect as an analytic category has caused people to struggle with the question, "what is an aspect," just as they struggled with the question, "what is an object, " when object-oriented languages and methods were introduced. On the other hand, it has consequences that are felt in the form of unnecessary constraints on designers, surprising non-compositionality properties, undue design complexity in resulting programs, and performance problems for which it does not appear optimizations are readily available.

We also proposed a unified model [25], embodied by the Eos language [23], which replaces the non-orthogonal and asymmetric abstractions of aspect and class, and advice and method, respectively with the more orthogonal notions of classpect, method, and binding. Based on the Eos programming model's ability to improve the modularization of integration [23] [29] and higher-order concerns [25], we made informal claims about its expressiveness.

This paper presents formal validation of the claim that the unified model is more expressive then the AspectJ language model. The validation is provided by a sound formal argument based on Felleisen's notion of macro-eliminable programming language extensions [11]. In particular, we show that key constructs in AspectJ are macro-eliminable in Eos, but not the other way around. Informally, we show that Eos retains the expressive power of AspectJ in essentially all major dimensions, and that in one important dimension Eos is strictly more expressive than the AspectJ-like languages.

The rest of this paper is organized as follows. In the next Section, we give background on aspect-oriented programming, the unified language model proposed in our earlier work, and Felleisen's formulation of the relative expressive power of Turning-complete languages. In Section 3, we formulate the expressiveness argument. In Section 4, we establish a representation to be used in the rest of the paper. In Section 5 and 6, we sketch a proof of the expressive power of the unified aspect language model with respect to the AspectJ language model. In Section 7, we illustrate the arguments presented in Section 5 and 6 through an example. Section 8 discusses related work, and Section 9 concludes.

2. Background

In this section, we briefly review the AspectJ and unified language model embodied by Eos. The focus is on their key differences. The AspectJ language model is described in detail by Kiczales et al.

* This research supported in part by NSF grant FCA-0429947.

```
1 aspect Tracing {
2   pointcut tracedCall():
3       execution(* *(..));
4   before(): tracedExecution() {
5       /* Trace the Execution */
6   }
7 }
```

Figure 1. A Simple Example Aspect

```
1 class Tracing {
2   pointcut tracedExecution():
3       execution(* *(..))&& !within(Trace);
4   static before tracedExecution(): Trace();
5   public void Trace() {
6       /* Trace the call */
7   }
8 }
```

Figure 2. A Simple Example Classpect

[16]. A complete language manual and compiler is available from the AspectJ web site [2] as of this writing. The unified language model is described in detail in our earlier work [25] and the Eos compiler is available from the Eos web site [10]. We also present the background on Felleisen's work on comparing the expressive power of Turing-complete languages [11].

2.1 The AspectJ Language Model

In this subsection, we will review basic concepts in the dominant aspect-oriented model, namely the AspectJ model. AspectJ [16] is an extension to Java [13]. The rest of this paper starts with this model. Other languages in this class include AspectC++ [28], AspectR [7], AspectWerkz [3], AspectS [14], Caesar [22], etc. While Eos [23] is not AspectJ-like, it is in the broader class of Pointcut-Advice-based AO languages [21]. The central goal of AspectJ-like languages is to enable the modular representation of crosscutting concerns.

AspectJ-like languages organize programs into a two-layered hierarchy. The concerns that can be modularized using the traditional OO modularization techniques, classes, are in the first layer. The first layer is also often called the base layer [19]. The modularized representation of the crosscutting concerns, aspects, are in the second layer. These aspects affect the behavior of the classes in the base layer.

These languages add five key constructs to the object-oriented model: join points, pointcuts, advice, inter-type declarations, and aspects. For the purpose of this paper, inter-type declarations are not relevant as they remain unchanged in the unified model. A simple example is shown in Figure 1 to make the points concrete.

An aspect (lines 1-7), modifies the behavior of a program at certain selected execution events exposed to such modification by the semantics of the programming language. These events are called join points. The execution of a method in the program in which the Tracing aspect appears is an example of a join point. A pointcut (lines 2-3) is a predicate that selects a subset of join points for such modification – here, execution of any method. An advice(see lines 4-6) is a special *method-like* constructs that effect such a modification at each join point selected by a pointcut. An aspect (lines 1-7) is a class–like module that uses these constructs to modify behaviors defined by the classes of a software system.

Like classes, aspects also support the data abstraction and inheritance, but they do differ from classes. First, aspects can use pointcuts, advice, and inter-type declarations. In this sense, they are strictly more expressive than classes. Second, instantiation of aspects and binding of advice to join points are wholly controlled by the Aspect language runtime. There is no *new* for aspects. Aspect instances are thus not first-class, and, in this dimension, classes are strictly more expressive than aspects. Third, although aspects can advise methods with fine selectivity, they can select advice bodies to advise only in coarse-grained ways.

In earlier work [23, 24], we addressed the limits of aspects with respect to instantiation and join point binding under program control, but we left aspects and classes separate and incomparable, and the resulting compositionality problems unresolved. We tackled this problem in the following work [25], leading to a unified language design in which advising emerged as a general alternative to overriding or method invocation.

2.2 The Unified Language Model

The new language model unifies aspects and objects in three ways. First, it unifies aspects and classes as *classpects*. A *classpect* has all the capabilities of classes, all of essential capabilities of aspects in AspectJ–like languages, and the extensions to aspects needed to make them first class objects. Second, the unified model eliminates advice in favor of using methods only, with a separate and explicit join-point-method binding construct. Third, it supports a generalized advising model. To the usual object-oriented mechanisms of explicit or implicit method call and overriding based on inheritance we add implicit invocation using before and after advice, and overriding using around advice, respectively.

To make the point concrete we revisit the example presented in the previous section in Figure 2. A classpect (lines 1-8), similar to the aspect in the previous section, declares a pointcut (lines 2-3) to select the execution of any method and then composes it with the *within(Trace)* pointcut expression to exclude the methods in itself to avoid recursion. A static binding (line 4) binds the method Trace (lines 5-7) to execute before all join points selected by the pointcut tracedExecution. Note that by binding statically join points in all instances are affected. A non-static binding binds to instances selectively. The key difference in this implementation is that all concerns are modularized as classpects and methods. The crosscutting concerns, however, uses bindings to bind the method containing the implementation of the crosscutting concerns to join points.

The AspectJ dichotomy between object-oriented and aspect modules, with the latter able to advising the former but not vice-versa promotes a two-level, asymmetric design style. AspectJ does provide limited mechanisms for aspects to advise other aspects, so two-level architecture is not strictly enforced, but it is awkward to write aspects that advise aspects. We view advising as a general module composition mechanism and believe that there is value supporting it as a first-class mechanism. It is in this dimension that we show that Eos is more expressive than the AspectJ-like languages.

2.3 On the Expressive Power of Programming Languages

In response to the lack of formal framework for specifying and verifying statements about the expressiveness of programming language, Felleisen in his work, *On the expressive power of programming languages* [11] proposed a formal notion of expressiveness and showed that his formal framework captures many informal ideas on expressiveness. His approach adopts the ideas from the formal systems to programming languages. In particular, he adopts the notion of expressible or eliminable systems proposed by Kleene [17] and extensions of Kleene's notion by Troelstra [30].

In order to understand the adaptation in the programming language context, let us first look into the notion of expressible systems in the logic/formal systems paradigm. The next subsection

presents essential definitions. These definitions are originally from Troelstra's work. Felleisen adopted them to the programming language context.

2.3.1 Formal Systems

A formal system L is a 3-tuple of sets {Expressions(L), Formula(L), Theorems(L)} such that $Expressions(L) \supset Formula(L) \supset Theorems(L)$. Expressions either are terms, or generated by applying logical and non-logical operators over other expressions. A formula is a recursive subset of the set of expressions and satisfies well-formedness criteria. Theorems are formula defined to be true in the formal system.

Conservative Extension: A conservative extension of a formal system L' is a formal system L if the following relationships hold between the expressions, formula and theorems in these two formal systems.

1. $Expressions(L) \supset Expressions(L')$
2. $Formula(L) \cap Expressions(L') = Formula(L')$
3. $Theorems(L) \cap Expressions(L') = Theorems(L')$

The first relationship denotes that the conservative extension L contains all the expressions in L' and possibly more. The richer sets of operations in the extension generate these additional expressions. The next two relationships express the fact that formula and theorems in the original formal system L' can be expressed in the extension L.

Definitional Extension: A definitional extension L of L', is a conservative extension for which there exists a relation $R : Expressions(L) \rightarrow Expressions(L')$ such that:

1. $\forall f \in Formula(L), R(f) \in Formula(L')$
2. $\forall f \in Formula(L'), R(f) = f$
3. R is homomorphic in all logical operators [1].
4. $L \vdash t \iff L' \vdash R(t)$
5. $L \vdash t \leftrightarrow R(t)$

The existence of the relation R that satisfies the properties described above makes the symbols that generate additional expressions in the extension L eliminable. Felleisen adopted this notion from the formal systems context to the programming language world to assess their relative expressiveness. The next subsection presents the definition of programming languages and their definitional and conservative extension from his work.

2.3.2 Programming Languages

Programming Language: A programming language is modeled as a 3-tuple of sets {Phrases(L), Programs(L), Semantics(L)} where these sets are defined as follows:

- *Phrases*(L) is a set of abstract syntax trees freely generated from function symbols $F, F_1, \ldots$ with arities $a, a_1, \ldots$.
- *Programs*(L) is non-empty recursive subset of the Phrases(L).
- *Semantics*, $eval_L$, is a recursively enumerable predicate on Programs(L). The program terminates iff $eval_L$ holds for that program.

Conservative Extension (Restriction) of a Programming Language: Let us assume a programming language L' is extended with additional symbols $\{F_1, \ldots, F_n, \ldots\}$ to yield an extension L. This extension is represented as $L = L' \backslash \{F_1, \ldots, F_n, \ldots\}$, and is a conservative extension if:

1. constructors of L are constructors of L' plus additional $\{F_1, \ldots, F_n, \ldots\}$.
2. $Phrases(L') \subset Phrases(L)$ with no constructs in $\{F_1, \ldots, F_n, \ldots\}$.
3. $Programs(L') \subset Programs(L)$ with no constructs in $\{F_1, \ldots, F_n, \ldots\}$.
4. Semantics(L') is a restriction of Semantics(L).

Eliminable Programming Constructs: Let $L = L' \backslash \{F_1, \ldots, F_n, \ldots\}$ be a conservative extension of L'. The new constructs $\{F_1, \ldots, F_n, \ldots\}$ in the language L are *eliminable* if there exists a mapping $R : Phrases(L) \rightarrow Phrases(L')$ such that:

1. $\forall P \in Programs(L), R(P) \in Programs(L')$
2. R is homomorphic in all constructs of L', i.e. $R(F(e_1, \ldots, e_a)) = F(R(e_1), \ldots, R(e_a))$ for all constructs F of L'.
3. $\forall P \in Programs(L), eval_L(e)$ is true $\leftrightarrow eval_{L'}(R(P))$ is true.
4. $Programs(L') \subset Programs(L)$ with no constructs in $\{F_1, \ldots, F_n, \ldots\}$.
5. $\forall P \in Programs(L'), eval_{L'}(P)$ is true $\leftrightarrow eval_L(P)$ is true.

Macro Eliminable Extension of a programming language: Let $L = L' \backslash \{F_1, \ldots, F_n, \ldots\}$ be a conservative extension of L', i.e. L' is conservative restriction of L. The constructs $\{F_1, \ldots, F_n, \ldots\}$ are *macro eliminable* if they are eliminable and if eliminating mapping R satisfies the following condition as well.

- $\forall F \in \{F_1, \ldots, F_n, \ldots\} \exists \alpha - ary$ syntactic abstraction, A, over L' such that $R(F(e_1, \ldots, e_\alpha)) = A(R(e_1), \ldots, R(e_\alpha))$.

In other words, the original language can locally express each additional construct in the conservative extension. If such an abstraction does not exists the language extension is not macro eliminable, therefore it is more expressive. Felleisen described this expressiveness relationship as follows:

> Given two universal programming languages that only differ by a set of programming constructs, $c_1, c_2, \ldots, c_n$, the relation holds if the additional constructs make the larger language more expressive than the smaller one. Here *more expressive* means that the translation of a program with occurrences of one of the constructs c_i to the smaller language requires a global reorganization of the entire program.

In the next section, we apply this notion of expressive power to compare the unified model proposed by our earlier work [25] and the AspectJ model [16].

3. Expressive Power of The Unified Model

The unified model (L_{Eos}) proposed in this work differs from the AspectJ model (L_{AJ}) by the constructs {*advice, aspect, binding, classpect, class*} as shown in the Table 1. The symbol *X* denotes the presence of a construct in the model. For brevity, the table only shows the constructs of interest. The row *Other* denotes other constructs like *events, indexers, inter-type declaration* etc. that are present in both language model.

As depicted in Table 1, L_{AJ} contains advice, aspect, and class that are eliminated in L_{Eos} in favor of classpect and binding. The rest of the constructs are present in both models, so they will not be discussed in the rest of this section, unless relevant. Let us construct a third model $L_{Combined}$, where constructs in $L_{Combined}$ = constructs in $L_{Eos} \cup$ constructs in L_{AJ}. The language $L_{Combined}$ can be represented as $L_{Combined} = L_{Eos} \setminus \{aspect, advice, class\}$ or $L_{Combined} = L_{AJ} \setminus \{classpect, binding\}$, i.e. as conservative

[1] A mapping R from formal system L to L' is homomorphic with respect to some hypothetical binary logical operator l if $a \in Expressions(L)$, $b \in Expressions(L)$, $R(l) = o \iff R(alb) = R(a)oR(b)$.

$\varepsilon \in E$	(Expression)
$\zeta \in E$	(Method call expression)
$\xi \in E$	(Advice or method body)
$\sigma \in E$	(Argument)
$r \subset E$	(Method return expressions)
$proceed \in E$	(Proceed expression)
$j \in J \subset (E \times E \times E)$	(Join points)
$\wp(J)$	(Power set of J)
$ej \in EJ$	(Method execution join points)
$aj \in AJ$	(Advice execution join points)
$id \in E$	(Identifier expression)
$reflect \in E$	(Reflective information at the join point)
$\rho \in \wp$	(Pointcut expression)
$methodexecution \in \wp_m(J) \subset \wp(J)$	(Method execution pointcut designators)
$adviceexecution \in \wp_a(J) \subset \wp(J)$	(Advice execution pointcut designators)
$pattern \in E$	(Pattern expression)
j ::= $id_j\ reflect_j\ \varepsilon_j$	(Join point)
$\xi ::= \varepsilon_\xi \mid ej_\xi \mid aj_\xi$	{body}
methodexecution: (pattern × J) → { ej : ∀ ej ∈ J, $id_{ej} \mapsto$ pattern }	(Execution point cut matching.)
adviceexecution: J → {aj: ∀ aj ∈ J}	(Advice execution point cut matching.)
$\wp(J) = \wp_m(J) \cup \wp_a(J)$	

Figure 3. Pointcut and Join Points

$\alpha \in AS$	(Aspects)
$c \in C$	(Classes)
$\hat{c} \in \hat{C}$	(Classpects)
$ctor \in (E \times E)$	(Constructors)
$f \in E$	(Fields)
$m \in M \subset (E \times E)$	(Methods)
$a \in A \subset (\tau \times E \times E)$	(Advice)
$\tau ::= before \mid after \mid around$	(Temporal specification)
$b \in (\tau \times E \times E \times E)$	(Bindings)
$a ::= \tau_a\ \sigma_a\ \rho_a\ \xi_a$ Where $\exists\ \sigma_a \mapsto reflect_{\rho_a}$.	Advice form
$b ::= \tau_b\ \rho_b\ \zeta_b\ \sigma_a$ Where $\exists\ \sigma_b \mapsto reflect_{\rho_b} \wedge \exists\ \sigma_b \mapsto \sigma_{\zeta_b}$.	Binding form
$m ::= id_m\ \sigma_m\ \xi_m$	Method form
$\alpha ::= [a_\alpha]^*\ [f_\alpha]^*\ [m_\alpha]^*\ [\rho_\alpha]^*$	Aspects
$c ::= [ctor_c]^*\ [f_c]^*\ [m_c]^*\ [\rho_c]^*$	Classes
$\hat{c} ::= [b_{\hat{c}}]^*\ [ctor_{\hat{c}}]^*\ [f_{\hat{c}}]^*\ [m_{\hat{c}}]^*\ [\rho_{\hat{c}}]^*$	Classpects

Figure 4. Aspect and Classpect Members

extensions of L_{Eos} and L_{AJ}. To support the informal claims about expressiveness, it will be sufficient to prove the Theorem 3.1.

Theorem 3.1. *L_{Eos} is more expressive then L_{AJ}.*

Lemma 3.1. *The constructs $\{aspect,\ advice,\ class\}$ are macro-eliminable from the language $L_{Combined} = L_{Eos} \setminus \{aspect, advice, class\}$.*

Lemma 3.2. *The constructs $\{classpect,\ binding\}$ are not macro–eliminable from the language $L_{Combined} = L_{AJ} \setminus \{classpect, binding\}$.*

Theorem 3.1 is self-explanatory. Informally, the Lemma 3.1 means that the extra constructs aspect, advice and class construct can be eliminated because they do not add additional expressiveness to the combined model relative to the unified model. The Lemma 3.2 means that the constructs classpect and binding add additional expressiveness to the combined model relative to the AspectJ model, so they cannot be eliminated without sacrificing expressiveness. To prove Theorem 3.1, it suffices to prove the Lemma 3.1 and 3.2 and to compose the two results. The next three sections will sketch the proof. The next section provides a representation of the language models. The rest of this paper uses that representation.

4. Representation of the Language Models

Figure 3 shows a representation of the relevant expressions, pointcuts, and join point matching process. A join point (j) is an expression that evaluates to an identifier expression (tag), an expression that evaluates to the reflective information, and an expression that evaluates to the body that constitutes the join point. A method-execution join point (ej) is a join point that evaluates to a tag equal to the name of the method, reflective information expression, and an expression equivalent to the method body. An advice-execution join point (aj) evaluates to a tag equal to null (advice is anonymous), reflective information expression, and an expression equivalent to the advice body.

A pointcut expression (ρ) can be an execution pointcut expression or an advice execution pointcut expression. To keep the for-

Table 1. Difference in Constructs

AspectJ Model: L_{AJ}
Unified Model: L_{Eos}
Combined Model: $L_{Combined}$

Construct	L_{AJ}	L_{Eos}	$L_{Combined}$
Advice	X		X
Aspect	X		X
Binding		X	X
Classpect		X	X
Class	X		X
Constructor	X	X	X
Field	X	X	X
Method	X	X	X
Pointcut	X	X	X
Others	X	X	X

malism brief and relevant to the current discussion, we have elided other pointcut expressions except method-execution and advice-execution join points in both models. The formalization only considers simple pointcut expressions. Extending it to complex pointcut expressions by adding operators is trivial and not relevant to the current discussion as both AspectJ model and Eos model support exactly the same pointcut sub-language. An execution pointcut expression evaluates to a subset of execution join points such that the pattern of the pointcut expression matches the tag of the execution join point. An advice-execution pointcut expression evaluates to all advice execution join points.

There are other expressions such as method call expression ζ, method and advice body as expression ξ, method and advice arguments as expression σ, method return expression r, and proceed expression p. The *proceed* expression can only be a sub-expression of advice body expression.

Figure 4 presents representations of aspect, class, advice, classpect, and binding. The notation X_Y means that X is associated with Y and the notation $[X_Y]^*$ means that one or more X's are associated with or contained in Y.

- An *aspect* α is a collection of zero or more advice constructs, fields, methods, and, pointcuts. Note that according to the AspectJ programming guide [2], *aspects are not directly instantiated with a new expression, with cloning, or with serialization. Aspects may have one constructor definition, but if so it must be of a constructor taking no arguments and throwing no checked exceptions.*
- A *class* c is a collection of zero or more constructors, fields, methods, and pointcuts. For simplicity, inner classes are not considered.
- A *classpect* $\hat{c}$ is a collection of zero or more bindings, constructors, fields, methods, and pointcuts.
- An *advice* evaluates to a 4–tuple: temporal specification, advice argument expression, pointcut expression, and body expression, such that there exists a translation between advice arguments and the reflective information exposed by the pointcut expression.
- A *method* evaluates to a 3–tuple: identifier expression, method argument expression, and body expression.
- A *binding* evaluates to a 3–tuple: temporal specification, pointcut expression, and method call expression, such that there exists a translation between the binding arguments and the reflective information exposed by the pointcut expression, and between the binding argument and the argument to the method call expression.

To prove the Lemma 3.1, it is sufficient to provide a textually local translation of the constructs $\{aspect, advice, class\}$ from $L_{Combined}$ to L_{Eos}. A translation from a language L to another language L' is defined as a meaning-preserving syntactically defined map that transforms programs from L to L' [26]. To prove the Lemma 3.2, it is sufficient to provide a translation of the constructs $\{classpect, binding\}$ from $L_{Combined}$ to L_{AJ}, to show that it is not textually local, and to prove that any other translation will preserve this property. The next section presents a translation of constructs $\{aspect, advice, class\}$ from $L_{Combined}$ to L_{Eos}

5. Translation from $L_{Combined}$ to L_{Eos}

Lemma 5.1. *A class is macro–eliminable in* $L_{Combined} = L_{Eos} \setminus \{aspect, advice, class\}$.

$$\begin{array}{l}\Gamma : C \mapsto \hat{C} \iff \\ \quad \forall c \in C \; \exists \, \hat{c} \in \hat{C} \\ \qquad [[ctor_c]^* \equiv [ctor_{\hat{c}}]^* \\ \qquad \wedge [f_c]^* \equiv [f_{\hat{c}}]^* \\ \qquad \wedge [m_c]^* \equiv [m_{\hat{c}}]^* \\ \qquad \wedge [\rho_c]^* \equiv [\rho_{\hat{c}}]^*]\end{array}$$

Figure 5. Translation of a Class

The translation $\Gamma : C \mapsto \hat{C}$, of a class to a classpect proves the Lemma 5.1. This translation is trivial as shown in Figure 5. The translation from the members of a class to equivalent members in the classpect is also trivial. In practice, this translation is not even necessary in Eos as classpects are also represented using the keyword *class*. As can be observed, this translation is textually local and therefore classes are macro–eliminable in the conservative extension $L_{Combined}$.

Lemma 5.2. *An aspect is macro-eliminable in* $L_{Combined} = L_{Eos} \setminus \{aspect, advice, class\}$.

$$\begin{array}{l}\ddot{\Gamma} : \alpha \mapsto \hat{c} \iff \\ \quad \forall \alpha \in AS \; \exists \, \hat{c} \in \hat{C} \\ \qquad [[f_\alpha]^* \equiv [f_{\hat{c}}]^* \\ \qquad \wedge [m_\alpha]^* \equiv [m_{\hat{c}}]^* \\ \qquad \wedge [\rho_\alpha]^* \equiv [\rho_{\hat{c}}]^* \\ \qquad \wedge [\Pi(a_\alpha)]^* \equiv [(b \times m)_{\hat{c}}]^*]\end{array}$$

Figure 6. Translation of an Aspect

The translation $\ddot{\Gamma}$ from an aspect in the combined language model to a classpect in the unified model (See Figure 6) has two parts: translation Π of an advice to it's equivalent in the classpect and translation of other members to their equivalent. The second part of the translation is trivial, as it just requires straightforward translation (in practice textual copy) of the other members to the classpect. The first part of the translation is non-trivial, and complete aspect to classpect translation exists if and only if a valid translation from advice exists. This translation is textually local, so the construct aspect is also macro-eliminable if and only if advice is macro-eliminable.

Lemma 5.3. *An advice is macro-eliminable in* $L_{Combined} = L_{Eos} \setminus \{aspect, advice, class\}$.

The translation Π of an advice to an equivalent method and a binding is shown in Figure 7. This translation exists if and only if:

- The temporal specification of the advice is equivalent to the temporal specification of the binding.

$\Pi : A \mapsto (B \times M) \iff$
$\forall a \in A\ \exists (b, m) \in (B \times M)$
$[\tau_a = \tau_b$
$\wedge \rho_a \equiv \rho_b$
$\wedge \xi_a \equiv \xi_m$
$\wedge \zeta_b \equiv id_m$
$\wedge \sigma_a \equiv \sigma_b$
$\wedge (\xi_a$ depends on $reflect_{\rho_a} \rightarrow$
$reflect_{\rho_b} \mapsto \sigma_b \wedge reflect_{\rho_b} \mapsto \sigma_{\zeta_b})$
$\wedge (\xi_a$ depends on $proceed_{\rho_a} \rightarrow$
$proceed_{\rho_a} \mapsto \sigma_b \wedge proceed_{\rho_a} \mapsto \sigma_{\zeta_b})]$

Figure 7. Translation of an Advice

- There exists a translation from the pointcut in the advice to the pointcut in the binding.
- There exists a translation from the advice body to the method body.
- There exists a translation from the binding call expression to the method's identifier expression.
- There exists a translation from advice parameters to the binding parameters.
- In addition, if the advice body expression depends on reflective information expression or *proceed* expression, there exists a translation from them to binding argument expression and binding call expression.

All these sub-translations, except the sub-translation from advice body to method body, are trivial. For example the temporal specification of the advice *before/after/around* will be textually translated to the temporal specification of the binding *before/after/around*. Only the translation from the advice body to the method body is slightly involved, because AspectJ advice can use implicit reflective variables like *thisJoinPoint* and special forms like *proceed* to invoke the inner join points. If the advice body does not use any of these special construct, the translation is equivalent to textual copy. If the advice body uses the implicit variables, the translation from advice body to method body has three steps. First, a unique argument is added to the method. Second, all occurrences of the implicit variable in the advice body are replaced by the new argument, and third, the argument is bound to the reflective information in the binding.

If the advice body uses the special form *proceed*, the translation has three steps. First, a unique argument to represent delegate chain is added to the method. Second, all occurrences of the special form in the advice body are replaced by the call on the argument to invoke the next delegate in the chain, and third, the argument is bound to the delegate chain in the binding. For advice body with both implicit variable and binding, a combination of the two translations described above can be used. All these translations are textually local, so the construct advice is macro-eliminable, and therefore construct aspect is also macro-eliminable proving Lemma 5.2 and Lemma 5.3.

The Lemma's 5.1, 5.2, and 5.3 together complete the proof of Lemma 3.1. If we recall, Lemma 3.1 shows that $\{class, advice, andaspect\}$ can be eliminated from the combined language model without sacrificing expressiveness. In other words, if a language model contains $\{classpect, binding\}$; addition of $\{class, aspect, advice\}$ to it doesn't enhances its expressiveness. All new programs generated by $\{class, aspect, advice\}$ can be translated to an equivalent program using $\{classpect, binding\}$.

To prove the Lemma 3.2, the next section provides a translation of the constructs $\{classpect, binding\}$ from $L_{Combined}$ to L_{AJ}, shows that it is not textually local, and proves that any other translation will preserve this property.

6. Translation from $L_{Combined}$ to L_{AJ}

Lemma 6.1. *Classpect is not macro–eliminable in* $L_{Combined} = L_{AJ} \backslash \{classpect, binding\}$.

$\hat{C} = \{\hat{c} : \hat{c}$ is a classpect$\}$
$\hat{C}_1 = \{\hat{c} : n([b_{\hat{c}}]^*) = 0 \wedge n([ctor_{\hat{c}}]^*) \neq 0\}$
$\hat{C}_2 = \{\hat{c} : n([b_{\hat{c}}]^*) \neq 0 \wedge n([ctor_{\hat{c}}]^*) = 0 \wedge \neg HO(\hat{c})\}$
$\hat{C}_3 = \{\hat{c} : n([b_{\hat{c}}]^*) \neq 0 \wedge n([ctor_{\hat{c}}]^*) \neq 0 \wedge \neg HO(\hat{c})\}$
$\hat{C}_4 = \{\hat{c} : n([b_{\hat{c}}]^*) \neq 0 \wedge n([ctor_{\hat{c}}]^*) = 0 \wedge HO(\hat{c})\}$
$\hat{C}_5 = \{\hat{c} : n([b_{\hat{c}}]^*) \neq 0 \wedge n([ctor_{\hat{c}}]^*) \neq 0 \wedge HO(\hat{c})\}$
Here $n([b_{\hat{c}}]^*)$ and $n([ctor_{\hat{c}}]^*)$ are the number of bindings and constructors in $\hat{c}$.

$HO(\hat{c}) = \exists b_{\hat{c}} : [\rho_{b_{\hat{c}}} \cap \{ej : ej \in EJ_{L_{Eos}}$
$\wedge ej \mapsto aj \wedge aj \in AJ_{L'}\} \neq \phi]$
$\hat{C} = \hat{C}_1 \cup \hat{C}_2 \cup \hat{C}_3 \cup \hat{C}_4 \cup \hat{C}_5$

Figure 8. Partition of Classpects

A binding selects a subset of join points and binds a method to execute at these join points. Let us call these join points *subjects* of binding and the bound method *handler* of binding. For the purpose of the rest of this paper, a binding is a *higher-order binding* if the *subjects* of the binding are *handlers* of other bindings or contained in *handlers* of other bindings. A classpect is a *higher-order classpect* if it contains one or more *higher-order bindings*. In other words, it modularizes concerns that are scattered and tangled with other crosscutting concerns. We call these concerns *higher-order concerns* [25]. The Boolean function $HO(\hat{c})$ checks the condition, *Is* $\hat{c}$ *a higher-order classpect?*. It checks if the set of method execution join points selected by any binding maps to an advice execution join point in the current model.

The translation of a classpect in the unified model to constructs in the unified model (Ω) is non-trivial. The domain $\hat{C}$, is divided into five disjoint partitions, $\hat{C}_1$, $\hat{C}_2$, $\hat{C}_3$, and $\hat{C}_4$, as shown in Figure 8. This partition is based on two properties: *instantiation* and the presence of *higher-order bindings*. The classpects in the partition $\hat{C}_4$ and $\hat{C}_5$ are higher–order classpects (i.e. $HO(\hat{c})$ is true for these classpects) and the classpects in the partition $\hat{C}_1$, $\hat{C}_2$, and $\hat{C}_3$ are not. The partition $\hat{C}_1$ contains classpects with no bindings but may contain one or more constructors (i.e. these classpects may be instantiated). The second partition $\hat{C}_2$ contains classpects with one or more bindings but no constructors (i.e. classpects that are never instantiated). The third partition $\hat{C}_3$ contains classpects with one or more bindings and constructors that are not higher-order classpect. The fourth partition $\hat{C}_4$ contains classpects with one or more bindings but no constructors that are higher-order classpects. The fifth partition $\hat{C}_5$ contains classpects with one or more bindings and one or more constructors that are higher-order classpects. These partitions and corresponding translations are systematically analyzed in the rest of this section.

Five translations Ω_1, Ω_2, Ω_3, Ω_4, and Ω_5 map $\hat{C}_1$, $\hat{C}_2$, $\hat{C}_3$, and $\hat{C}_4$ to C, AS, $(C \times AS)$, $AS \hookleftarrow \{[\alpha]^+ : \alpha \in AS\}$, and $(C \times AS) \hookleftarrow \{[\alpha]^+ : \alpha \in AS\}$ respectively (See Figure 8). The notation $[\alpha]^+$ denotes a collection of one or more aspects. The notation $AS \hookleftarrow \{[\alpha]^+ : \alpha \in AS\}$ denotes an aspect, correct implementation of which requires changes in $[\alpha]^+$ to expose right join points.

$$\Omega = \Omega_1 \cup \Omega_2 \cup \Omega_3\Omega_4 \cup \Omega_5$$
$$\Omega_1 : \hat{C}_1 \mapsto C$$
$$\Omega_2 : \hat{C}_2 \mapsto AS$$
$$\Omega_3 : \hat{C}_3 \mapsto (C \times AS)$$
$$\Omega_4 : \hat{C}_4 \mapsto AS \hookleftarrow \{[\alpha]^+ : \alpha \in AS\}$$
$$\Omega_5 : \hat{C}_5 \mapsto (C \times AS) \hookleftarrow \{[\alpha]^+ : \alpha \in AS\}$$

Figure 9. Five Sub-Translations of a Classpect

- The first translation's domain contains classpect with no bindings. These classpects can be translated to classes (C).
- The second translation's domain contains classpects with one or more bindings but that are never instantiated. These classpects can be translated to aspects.
- The third translation's domain contains classpects with one or more bindings and constructors (i.e. they are instantiated) that are not higher-order classpect. These classpects cannot be translated to classes, because classes cannot advise. They cannot be directly translated to aspects, because aspects cannot be instantiated. These classpects can be translated to a (class, aspect) 2-tuple, where the class handles instantiation and aspect handles advising.
- The fourth translation's domain contains classpects with one or more bindings but no constructors that are higher-order classpects. These classpects can be translated to aspects, but successful translation requires that all join points required by this aspect are still quantifiable in the translation. By quantifiable we mean that there exists a set of pointcut expression in the translation to select all join points required by the aspect.
- The fifth translation's domain contains classpects with one or more bindings and one or more constructors that are higher-order classpects. Similar, to the third translation, these classpects are translated to (class, aspect) 2-tuple, provided that all join points required by the aspect are still quantifiable.

$$\begin{array}{l}\forall \hat{c} \in \hat{C}_1 \; \exists \; c \in C \\ \quad [[ctor_{\hat{c}}]^* \equiv [ctor_c]^* \\ \quad \wedge [f_{\hat{c}}]^* \equiv [f_c]^* \\ \quad \wedge [m_{\hat{c}}]^* \equiv [m_c]^* \\ \quad \wedge [\rho_{\hat{c}}]^* \equiv [\rho_c]^*]\end{array}$$

Figure 10. Translation of a Classpect to a Class

The translation Ω_1 shown in Figure 10 is trivial. It simply maps the members of a classpect to equivalent members in the class. This translation is textually local, so the construct classpect is macro-eliminable if it does not contain any bindings. This result is not surprising as a classpect with no bindings in the unified model is equivalent to a class in the AspectJ model.

The translation Ω_2 shown in Figure 11 has two parts: the translation Θ_1 from a (method, binding) pair in a classpect to an equivalent advice in the aspect, and the translation of the rest of the members from classpect to aspect. Therefore, Ω_2 exists if and only if Θ_1 exists. This translation is textually local if and only if Θ_1 is textually local.

The translation Θ_1 maps a (method, binding) pair in a classpect to an equivalent advice in the aspect. Figure 11 describes this translation. It maps a (method, binding) pair to an advice such that the arguments and body expressions of the method is equivalent to the arguments and body expressions of the advice, the temporal specification of the advice is the same as temporal specification of the binding, and the pointcut expression of the binding is equivalent

$$\begin{array}{l}\Omega_2 : \hat{C}_2 \mapsto AS \iff \\ \quad \forall \hat{c} \in \hat{C}_1 \; \exists \; \alpha \in AS \\ \quad\quad f_{\hat{c}}]^* \equiv [f_\alpha]^* \\ \quad\quad \wedge [m_{\hat{c}}]^* \equiv [m_\alpha]^* \\ \quad\quad \wedge [\rho_{\hat{c}}]^* \equiv [\rho_\alpha]^* \\ \quad\quad \wedge [\Theta_1((b,m)_{\hat{c}})]^* \equiv [a_\alpha]^*\end{array}$$

$$\begin{array}{l}\Theta_1 : (B \times M) \mapsto A \iff \\ \quad \forall \text{ (b,m)} \in \text{(B} \times \text{M)} \; \exists \; a \in A \\ \quad\quad \wedge \tau_b = \tau_a \\ \quad\quad \wedge \rho_b \equiv \rho_a \\ \quad\quad \wedge \xi_m \equiv \xi_a \\ \quad\quad \wedge \sigma_b \equiv \sigma_a]\end{array}$$

Figure 11. Translation of a Classpect to an Aspect

to the pointcut expression of advice. All these sub-translations are trivial except in the following cases:

- When the binding's temporal specification is equal to *around* and the pointcut expression in the binding exposes the delegate chain and supplies it as an argument to the method to invoke inner delegates. In other words, when the method bound around uses Eos's equivalent of *proceed*. In this case, the delegate chain argument is eliminated from the binding's pointcut expression and the handler method. All occurrences of the invocation inside the method body is replaced by a *proceed* call. This translation is textually local in that it affects the code segment inside the body.
- When the pointcut expression in the binding exposes the reflective variables *thisJoinPoint*, etc. In this case, the argument is eliminated from the pointcut expression and the handler method. All reference to the explicit arguments are replaced by the reference to the implicit variable. This translation is also textually local.

This translation is overall textually local, therefore Ω_2 is also textually local leading to another result. The construct classpect is macro-eliminable if it is not explicitly instantiated using *new* and it is not a higher-order classpect.

Lemma 6.2. *A non-higher-order classpect that can be explicitly instantiated is not macro–eliminable.*

$\because n([ctor_\alpha]^*) = 0 \therefore [ctor_{\hat{c}}]^* \mapsto [ctor_c]^*$
All instantiation capabilities have to be emulated by a class.
Aspects may not be explicitly instantiated.

$\because n([a_c]^*) = 0 \therefore [a_{\hat{c}}]^* \mapsto [a_\alpha]^*$
All advising has to be done by a separate aspect.
Classes may not advise.

$$\begin{array}{l}\Omega_3 : \hat{C}_3 \mapsto C \times AS \iff \\ \quad \forall \hat{c} \in \hat{C}_1 \; \exists \; (c,\alpha) \in C \times AS \\ \quad\quad [[ctor_{\hat{c}}]^* \equiv [ctor_c]^* \\ \quad\quad \wedge [f_{\hat{c}}]^* \equiv [f_c]^* \\ \quad\quad \wedge [m_{\hat{c}}]^* \equiv [m_c]^* \\ \quad\quad \wedge [\rho_{\hat{c}}]^* \equiv [\rho_\alpha]^* \\ \quad\quad \wedge [\Theta_1((b,m)_{\hat{c}})]^* \equiv [a_\alpha]^*\end{array}$$

Figure 12. Translation of a Classpect to a Class and an Aspect

The translation Ω_3 shown in Figure 12 is subtle as it involves partitioning of roles between an aspect and a class. The pointcuts

are mapped to the equivalent pointcut in the aspect. The method binding pair in this case is mapped to an advice in an aspect by the translation Θ_1. The translation Θ_1 is textually local, but Ω_3 is not textually local as it results in the classpect represented as two abstractions an aspect and a class. *Therefore, a non-higher-order classpect with constructors (i.e. a classpect that can be instantiated) is not macro-eliminable proving Lemma* 6.2.

Lemma 6.3. *Higher-order Classpect is not macro-eliminable in* $L_{Combined} = L_{AJ} \backslash \{classpect, binding\}$.

$$
\begin{array}{l}
\Omega_4 : \hat{C_4} \mapsto AS \hookleftarrow \{[\alpha]^+ : \alpha \in AS\} \iff \\
\quad \forall \hat{c} \in \hat{C_4} \exists \alpha \in AS \\
\qquad [\forall b \notin HOB(\hat{c}) \Theta_1((b,m)_{\hat{c}})]^* \equiv [a_\alpha]^* \\
\qquad [\forall b \in HOB(\hat{c}) \Theta_2((b,m)_{\hat{c}})]^* \equiv [a_\alpha]^* \\
\qquad \wedge [f_{\hat{c}}]^* \equiv [f_\alpha]^* \\
\qquad \wedge [m_{\hat{c}}]^* \equiv [m_\alpha]^* \\
\qquad \wedge [\rho_{\hat{c}}]^* \equiv [\rho_\alpha]^* \\
\text{Where,} \\
HOB(\hat{c}) = \{b : b \in [b_{\hat{c}}]^* \\
\qquad \wedge [\rho_b \cap \{ej : ej \in EJ_{L_{Eos}} \\
\qquad \wedge ej \mapsto aj \\
\qquad \wedge aj \in AJ_{L'}\} \neq \phi]\}
\end{array}
$$

Figure 13. Translation of a Higher-Order Classpect

$$
\begin{array}{l}
\Theta_2 : (B \times M) \mapsto A \hookleftarrow \{[a]^+ : a \in A\} \iff \\
\qquad \wedge [\forall b \in HOB(\hat{c}), \forall m \in X(b), \exists b', \zeta_{b'} \equiv id_m \rightarrow \\
\qquad\quad \exists a \in [a_\alpha]^*, \exists m' \in [m_\alpha]^*, \\
\qquad\qquad \tau_a = \tau_{b'} \wedge \rho_a \equiv \rho_{b'} \wedge \xi_m \equiv \xi_{m'} \wedge \xi_a \equiv \zeta_m]] \\
\text{Where,} \\
X(b) = \{m : \forall m \in M_{L_{Eos}}, \\
\qquad \exists j \in \{\rho_b \cap \{ej : ej \in EJ_{L_{Eos}} \\
\qquad\quad \wedge ej \mapsto aj \\
\qquad\quad \wedge aj \in AJ_{L_{AJ}}\}, id_j = id_m\}
\end{array}
$$

Figure 14. Translation of a Higher-Order Binding

The translation Ω_4 (See Figure 13) exists if translations Θ_1 and Θ_2 exists to map (non-higher-order binding, method) and (higher-order binding, method) in the classpect to corresponding advice. The pointcuts, fields, and methods similarly should have their equivalent in the aspect. The non–trivial part of this translation is the translation Θ_2, which translates the higher-order bindings of the classpect. In the translation, the relation $HOB(\hat{c})$ yields a subset of bindings of the classpect $\hat{c}$ such that each binding in that subset is a higher-order binding. To recall, a binding is a *higher-order* if its *subject* join points are or contained in *handlers* of other bindings in the program. An important property of these *subjects* is that either they or their containers are eventually translated to advice constructs in L_{AJ}. The relation $HOB(\hat{c})$ selects higher-order bindings using this property. Note that a binding that has *subjects*, only parts of which are eventually translated to advice constructs, can be split into a *higher-order* and a normal binding. We call such bindings, *partial higher-order bindings*.

Lemma 6.4. *Higher-order Binding is not macro-eliminable in* $L_{Combined} = L_{AJ} \backslash \{classpect, binding\}$.

The relation $X(b)$ shown in Figure 14, selects the *subjects* of a higher-order binding b. All subjects of a higher-order binding are translated to an advice or are contained in an advice. An advice is anonymous; therefore, selecting individual advice using pointcuts is not possible in general. The translation requires that the subject join point be quantifiable [12]. Note that a pointcut expression selects a join point by matching its pattern with the identifier expression associated with the join point. To enable quantification using pointcuts, an identifier expression must be associated with the subject join points. One possible way to do it is to consistently use the delegation pattern. In this case, advice body is moved to a method and advice just calls that method. Another possibility is to associate distinguishing attributes with advice bodies. The figure uses the delegation pattern as an example.

The translation Θ_2 exists if for every *higher-order binding* b and for each *subject* of that binding, if there are any other bindings (e.g. b') for which the *subject* is either the *handler* or contained in the *handler*, then there must exist an aspect α that contains this *subject* or the method containing this *subject* and an advice such that the advice delegates to the method.

This translation requires global changes. For each higher-order binding b, $n(X(b))$ advice constructs are to be modified. *A higher-order binding with no constructors is thus not macro-eliminable proving Lemma* 6.4. *Therefore, a higher-order classpect with no constructors is also not macro-eliminable proving Lemma* 6.3. The translation Ω_5 is a composition of the translations Ω_3 and Ω_4, therefore a higher-order classpect with constructors is also not macro-eliminable. The proof of Lemma 6.1 follows from the proof of Lemma 6.2 and Lemma 6.3. If we recall, Lemma 6.1 shows that $\{classpect, binding\}$ cannot be eliminated from the combined language model without sacrificing expressiveness. In other words, addition of $\{classpect, binding\}$ enhances the expressiveness of L_{AJ}.

In this paper we have shown that if a language model contains $\{classpect, binding\}$; addition of $\{class, aspect, advice\}$ to it doesn't enhances its expressiveness. All new programs generated by $\{class, aspect, advice\}$ can be translated to an equivalent program using $\{classpect, binding\}$. However, if a language model just has $\{class, aspect, advice\}$ addition of $\{classpect, binding\}$ enhances its expressiveness. The programs generated by $\{classpect, binding\}$ cannot be generated using $\{class, aspect, advice\}$. In terms of language models, we have proved that the combined model is more expressive then the AspectJ model and the combined model has the same expressive as the Eos programming model. From the two results it follows that Eos is more expressive then AspectJ.

7. Discussion

$b_{P_i} = \tau_{b_{P_i}}\ \rho_{b_{P_i}}\ \zeta_{b_{P_i}}\ \sigma_{b_{P_i}}$
$m_{P_i} = id_{m_{P_i}}\ \sigma_{m_{P_i}}\ \xi_{m_{P_i}}$
$\tau_{b_{P_i}}$ = around
$\rho_{b_{P_i}}$ = pointcut to select execution of method $m_{\hat{c}_i}$
$\zeta_{b_{P_i}}$ = call expression, target method is m_{P_i}
$\sigma_{b_{P_i}} \mapsto reflect_{\rho_{b_{P_i}}}$ /* Binding arguments map to the reflective variables exposed by pointcuts.*/
$\sigma_{b_{P_i}} \mapsto \sigma_{\zeta_{b_{P_i}}}$ /* Binding arguments map to the arguments of the method call expression.*/

Figure 15. The Retry Policy Binding and Method

In this section, we present an informal discussion of the formalism described in the previous section. For the purpose of this discussion, we use a simple but representative example. We use the notation described in Figure 3 and 4. Our example system models error-prone resources and fault-tolerance policies. There are *n* different types of resources in the system, $R_1, R_2, \ldots, R_n$. A requirement is to construct a fault-tolerant system with these error-prone

$\hat{c}_O$ = b_i *ctor f* m_i
$b_i = \tau_{b_i}\ \rho_{b_i}\ \zeta_{b_i}\ \sigma_{b_i}$
$m_i ::= id_{m_i}\ \sigma_{m_i}\ \xi_{m_i}$
τ_{b_i} = after
ρ_{b_i} = pointcut to select execution of method m_{P_i}
ζ_{b_i} = call expression, target method is m_i
$\sigma_{b_i} \mapsto reflect_{\rho_{b_i}}$ /* Binding arguments map to the reflective variables exposed by pointcuts.*/
$\sigma_{b_i} \mapsto \sigma_{\zeta_{b_i}}$ /* Binding arguments map to the arguments of the method call expression.*/

Figure 16. The Overhead Computation Classpect

resources. To construct a fault-tolerant system, a retry policy type P_i is defined for each resource type R_i.

To implement this system using L_{Eos}, the resource types will be modeled as classpects $\hat{c}_i \in \hat{C}$. For simplicity, let us assume that each basic resource type R_i provides exactly one operation op_i. We can always model a resource type that provides more then one operation as a combination of basic resource types. These operations are modeled as methods, $m_{\hat{c}_i} \in M$. The retry policy types are also modeled as classpects $\hat{c}_{P_i} \in \hat{C}$. The retry policies are implemented using an around binding and a method. Each retry policy classpect $\hat{c}_{P_i}$ declares a binding b_{P_i} and a method m_{P_i} as shown in Figure 15. The binding b_{P_i} selects the execution of the method $m_{\hat{c}_i}$ and binds the method m_{P_i} to execute around it. When the method $m_{\hat{c}_i}$ is called to perform operation op_i on resource type R_i the method m_{P_i} bound around is invoked instead. The around method retries the original method specified number of times.

To translate this system to L_{AJ}, the resource types will be modeled to equivalent classes $c_i \in C$ as shown in the Figure 13. The retry policies will be mapped to equivalent aspects $\alpha_i \in AS$ such that the binding b_{P_i} and a method m_{P_i} is mapped to an around advice a_{P_i} essentially performing the same function as shown in Figure 13.

Let us consider an evolution scenario in which the system needs to compute the overhead of each retry. For each retry policy type P_i there is an associated overhead O_i. In L_{Eos}, to compute the overhead for retry policy, an overhead computation module is implemented as a classpect $\hat{c}_O \in \hat{C}$ as shown in Figure 16. The classpect provides a field to store the overhead, a constructor to initialize it to zero, a binding b_i and a method m_i corresponding to each retry policy P_i. The binding binds the method m_i to execute after the method m_{P_i} to record the overhead.

To translate the overhead requirement to L_{AJ}, one would think of translating the overhead classpect $\hat{c}_O$ to an aspect α_O with each binding b_i and method m_i pair translated to an after advice a_i. To pursue this straightforward implementation technique, selecting individual advice-execution join points a_{P_i} is required. We cannot select individual advice-execution join points because advice is not a named construct. An alternative is to perform a translation from a_{P_i} to another around advice a'_{P_i} and a method m'_{P_i} such that the around advice a'_{P_i} calls the method m'_{P_i}. Another alternative is the upfront translation from the binding b_{P_i} and the method m_{P_i} to the around advice a'_{P_i} and the method m'_{P_i}. Both these translations require global changes showing that L_{Eos} is more expressive compared to L_{AJ} in this dimension.

8. Related Work

To the best of our knowledge, this is the first paper to formally relate the expressiveness of variants of aspect-oriented languages, in general, or of AspectJ-like languages, in particular. Many languages and approaches have been compared using Felleisen's notion of expressive power of programming languages. For example, Brogi et al. compare the expressive power of three classes of coordination models based on shared dataspaces [4]. Among others, Roychoudhury et al. [27] informally compare aspect-oriented languages with meta-programming techniques.

There has been significant interest in formalizing aspect-oriented programming. To name a few: Andrews [1] take a process algebraic view of AOP in which aspects are modeled as processes and aspect interaction is modeled as process interaction. Douénce et al. [9] formalize aspect interaction using execution traces. Aspects monitor the execution traces and insert additional behaviors to it. Jagdeesan et al. [15] propose a calculus and an operational semantics for AOP. Clifton et al. [8] describe a parameterized calculus for AOP. Bruns et al. [6] give a name-based calculus in which aspect is the primitive unit of modularity and *named* advice is the primary unit of procedural abstraction. Most recently, Wand et al. [32] give a denotational semantics for a mini-language that embodies most features of AspectJ-like languages. Walker et al. [31] take an entirely different approach. They give semantics to AOP by defining a compact well-defined core language and a type-directed translation from the external language to the core language. There core language extends lambda calculus with explicitly defined join points and first-class advice.

Most of the formal models that we are aware of, subscribe to the dominant language model of AOP in which classes and aspects are non-orthogonal and asymmetric. The view taken by Bruns et al. [6] is particularly interesting as it takes completely opposite view of aspect-oriented programming eliminating classes and methods in favor of aspect and advice. The proposed model retains the coupling between *where the advice executes* and *what the advice does*. In this work, we do not claim to formalize aspect-oriented programming languages. Instead, we show the application of previously defined techniques to compare expressiveness of programming languages to the AO languages. We may be able to use one or more models of AOP discussed above as a baseline for our comparisons, however, we expect that it will require some adaptation to be able to represent the new design space created by the unification of aspect- and object-oriented constructs [25]. In addition, we expect significant simplification in these formal models due to the unification but we have not explored it yet.

There is another significant body of work on verifying aspect-oriented programs. For example, Krishnamurthi et al. [18] propose a modular technique that does not require re-verification on every change in advice constructs. Briefly, they construct a state-machine of the base program and the advice; identify labels to plug-in state-machine of the advice using pointcuts, and use model checking to verify the program. We hypothesize that our unified model eases the task of this kind of verification of aspect-oriented programs.

9. Concluding Remarks

We showed that our informal claims that the unified model is more expressive then the programming model of AspectJ-like languages is valid. In particular, we showed that the translation of certain programs from a language based on classpect and binding to a language based on class, aspect, and advice requires non-local transformations in the program. The set of programs that require non-local transformations contain at least two different types of classpects. First, classpects that are instantiated and that has one or more binding. Second, classpects that contain one or more higher-order binding. One way to try to account for these improvements is by appeal to the idea of conceptual integrity in design. Brooks wrote,

> ...that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect

one set of design ideas, then to have one that contains many good but independent and uncoordinated ideas. Simplicity and straightforwardness proceed from conceptual integrity. Every part must reflect the same philosophies and the same balancing of desiderata. Every part must even use the same techniques in syntax and analogous notions in semantics. Ease of use, then, dictates unity of design, conceptual integrity." [5](pp 42-44).

The additional expressive and compositional power of classpects emerged when this kind of design unity is enforced. It forced aspect–like constructs to support all of the capabilities of classes–notably new. It forced classes to support aspect–oriented advising as a generalized alternative to traditional invocation and overriding. By driving out anonymous advice in favor of methods as the sole mechanism for procedural abstraction, it also pushed a previously submerged but important abstraction to the fore: the join–point–method binding. A binding turned to be at the core of the aspect–oriented advising replacing asymmetric and non–orthogonal aspect and advice. What we have presented, then, is an aspect-oriented language model that is both significantly simpler than the AspectJ-like languages and formally more expressive in a dimension that really matters.

References

[1] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 187–209, London, UK, 2001. Springer-Verlag.

[2] AspectJ programming guide. http://www.eclipse.org/aspectj/.

[3] Jonas Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 5–6, New York, NY, USA, 2004. ACM Press.

[4] Antonio Brogi and Jean-Marie Jacquet. On the expressiveness of coordination via shared dataspaces. *Sci. Comput. Program.*, 46(1-2):71–98, 2003.

[5] Fredrick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison Wesley, Reading, Mass., second edition, 1995.

[6] Glen Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. muabc: 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] Avi Bryant and Robert Feldt. AspectR - simple aspect-oriented programming in Ruby, Jan 2002.

[8] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Technical Report 03-13, Iowa State University, Department of Computer Science, October 2003.

[9] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM Press.

[10] Eos web site. http://www.cs.virginia.edu/ eos.

[11] Matthias Felleisen. On the expressive power of programming languages. In N. Jones, editor, *Proceedings of the third European symposium on programming on ESOP '90*, volume 432, pages 134–151, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[12] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-oriented Software Development*, pages 21–35. Addison-Wesley Professional, 2004.

[13] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[14] Robert Hirschfeld. Aspects - aspect-oriented programming with squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[15] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In *Proc. European Conf. Object-Oriented Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 415–427. Springer-Verlag, 2003.

[16] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Budapest, Hungary, June 2001.

[17] Stephen Kleene. *Introduction to Metamathematics*. Number 1 in Bibliotheca mathematica. North-Holland, 1952. Revised edition, Wolters-Noordhoff, 1971.

[18] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. *SIGSOFT Softw. Eng. Notes*, 29(6):137–146, 2004.

[19] John Lamping. The role of base in aspect-oriented programming. In Cristina Videira Lopes, Andrew Black, Liz Kendall, and Lodewijk Bergmans, editors, *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, June 1999.

[20] Bruce J. MacLennan. *Principles of programming languages: design, evaluation, and implementation (2nd ed.)*. Holt, Rinehart & Winston, Austin, TX, USA, 1986.

[21] Hidehiko Masuhara and Gregor Kiczales. Modular crosscutting in aspect-oriented mechanisms. In Luca Cardelli, editor, *ECOOP 2003—Object-Oriented Programming, 17th European Conference*, volume 2743, pages 2–28, Berlin, July 2003.

[22] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[23] Hridesh Rajan and Kevin Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press.

[24] Hridesh Rajan and Kevin Sullivan. Need for instance level aspect language with rich pointcut language. In Lodewijk Bergmans, Johan Brichau, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, mar 2003.

[25] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[26] Jon G. Riecke. Fully abstract translations between functional languages. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–254, New York, NY, USA, 1991. ACM Press.

[27] Suman Roychoudhury, Jeff Gray, Hui Wu, Jing Zhang, and Yuehua Lin. A comparative analysis of meta-programming and aspect-orientation. In *41st Annual ACM SE Conference*, pages 196–201, Savannah, GA, March 2003.

[28] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[29] Kevin J. Sullivan and David Notkin. Reconciling environment

integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–68, July 1992.

[30] Anne S. Troelstra. *Metamathematical investigation of intuitionistic arithmetic and analysis*, volume 344 of *Springer LNM*. Springer-Verlag, Berlin, 1973.

[31] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, New York, NY, USA, 2003. ACM Press.

[32] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

[33] William Wulf. Trends in the design and implementation of programming languages. *IEEE Computer*, 13(1):14–24, 1980.