RC23345 (W0409-140) September 22, 2004
Computer Science

IBM Research Report

Concepts for Describing Composition of Software Artifacts

William Harrison, Harold Ossher, Peri Tarr
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Concepts for Describing Composition of Software Artifacts

William Harrison, Harold Ossher, Peri Tarr
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
+1 914 784 7339
{harrisn,ossher,tarr}@watson.ibm.com

Abstract

This paper treats the "compositor" component as a new, distinct, kind of software component. This is analogous to recognizing that compilers, parsers, and UI-generators are distinct kinds of software components. Each has its own domain of discourse and base of concepts, its own structure for expressing desired results, its own internal solution structure, and its own set of research problems. This paper describes a base of concepts suitable for expressing composition and shows how a general composition engine realizing these concepts can be used to effect the composition needs of several existing AOSD approaches.

1.1. Introduction

Many approaches supporting Aspect-Oriented Software Development (AOSD) ultimately require the composition or weaving together of separate elements drawn from separately encapsulated concerns. This is true not only for programming-language artifacts, but for requirements, use case, and design artifacts like those expressed in UML, and for other build and run-time artifacts like test suites, images, audio streams, user-interface descriptions, and menu material, Make, ANT, or WSDL scripts, or even just for simple packaging constructs like directories or jar files.

Most AOSD support is described in terms of complete approaches. An approach is far more than a programming language, tool, or component, though they may lie at its bottom. An approach consists of:

1. A characterization of a problem domain for which the approach is suitable,
2. A description of how a developer is expected to structure solutions for problems in that domain, and then
3. A tool or component that supports problem-solving with that solution structure

AspectJ [12], for example, is an approach to AOSD:

1. For the problem domain of attaching additional behavior to an existing software base
2. By creating constructs called "aspects" containing, for example, "advice" that can be attached to the existing base at "join-points"
3. Supported by a programming language (AspectJ), designed for expressing those constructs and their attachment, and its compiler and run-time library.

Hyper/J[17] , Composition Filters[2], and Adaptive Programming[15] are other approaches to AOSD that can be described in similar terms, but differently address different parts of the problem space.

Though differing in the problem domains they address, their pedagogical structure, and the way their support is presented to their developers, these approaches rely on two common elements:

1. Language constructs that require creating additional artifacts to be woven with the developer-produced artifacts,
2. A "compositor" or "weaver" component to actually effect the desired composition.

For example, many of AspectJ's *pointcut* specifications employ a run-time *residue* [10] to filter the events indicating execution of a *join point* according to conditions that can be tested only at run-time. This generated residue amounts to an additional artifact that is woven into the base program at appropriate points. The AspectJ compiler also contains a weaver that interprets other information in the pointcut to compose the *advice* code and the residues with the base code. Hyper/J's compositor employs a set of *composition relationships* to govern how a multiplicity of bodies of source code are to be composed into a new body of software.

This paper treats the "compositor" component as a new, distinct, kind of software component that addresses the *second* of the elements introduced above. This is analogous to recognizing that compilers, parsers and UI-generators, for example, are distinct kinds of software components. Each has its own domain of discourse and base of concepts, its own structure for expressing desired results, its own internal solution structure, its own set of research problems. This paper describes a base of concepts suitable for expressing composition and shows how a general composition engine realizing these concepts can be used to effect the composition needs of several existing AOSD approaches. That is not to say that AOSD approaches are expected to manifest these concepts to developers in their full generality, but rather that the approaches' composition needs can be described in terms of these concepts and realized on a compositor that implements them.

A composition component based on these concepts has been developed and is available as part of the Concern Manipulation Environment (CME) [5], an Eclipse Open-source Technology Project. It underlies the composition capabilities made available by the Concern Explorer and other CME tools. A description of the implementation and the research problems it presents is beyond the scope of this paper, but some additional concepts needed in realizing Java™ Composition and Extraction (of new artifacts from existing ones) in the CME are discussed.

The remainder of the paper is organized as follows: Section 2 presents a *very small* example, used in explaining the application of the concepts. Section 3 describes a model for material to be composed, and Section 4 discusses the concepts that are used for describing composition. Section 5 briefly discusses the CME Concern Composition Component CCC). Section 6 describes how CCC could be employed to effect the composition needs of AspectJ, Hyper/J, Composition Filters, or Adaptive Programming.

2. A Very Small Example

Consider designing a feature for a hypothetical, pre-existing thermostatic control system as a separate concern. The existing "basic" system contains Sensor classes that record temperature, maintain an updated average, and report when asked. They have methods for "report" and "update." The system is implemented with many independent subclasses, most but not necessarily all of which are named "*Sensor," each with its own style of implementation.

The "alarm" feature to be added is to produce a fire alert if temperature exceeds some threshold. The Sensors must now know a "controller", and alert it when needed. So they need have an added field: "controller". This feature is to intercept updates, and generate an alert when necessary.

Composition is the process of creating a new artifact or set of artifacts from a set of input artifacts by combining the content of the input artifacts according to some given specifications. For example, sensors that alert the controller must be created by composing the basic sensors and the alarm enhancements. Composition is thus only *part* of many AOSD approaches, and would not, for example, include the generation of code to track control flow as needed by AspectJ's "cflow" specifications. The artifacts to be composed fit a general model that allows the meaning of the specifications to be given. The model and the specifications are discussed in Sections 3 and 4.

3. A Model for Material to be Composed

Any description of how artifacts are to be composed presumes a common underlying/abstract representation for the artifacts themselves. The artifacts can be data objects, such as directories of files, or meta-data objects, such as programs and UML diagrams. It turns out that composition is similar across both, though meta-data material requires some extra concepts, such as typing. We present a single model, based on the richer, "meta-" context, with the understanding that not all capabilities of the definitions are applicable to all artifacts. We use Java programming elements in the discussion for illustrative purposes only—the model is applicable to artifacts defined in many languages and formalisms.

Spaces. A *space* encapsulates a body of material with a well-defined interpretation of all names used to reference other elements within it. Spaces simply contain named container definitions. They are artificial elements, not expected to be first-class elements in any of the material being composed, but to provide ways of dealing with a multiplicity of separately-defined corpora of first-class elements within which the same names might be used for different purposes. In dealing with Java programs, for example, a space may be defined by a *classpath,* consisting of all classes on that path.

Container Definitions. A *container* definition specifies a collection of named software elements that are its *members*. Members can be nested containers or interpretable material definitions, but not spaces. In dealing with Java programs, for example, classes and interfaces are treated as containers.

Interpretable Material Definitions. Interpretable material definitions provide the "meat" of the material subject to composition. Each interpretable element definition has a *body* whose content has a meaning, or interpretation, and whose correct interpretation may require proper resolution of by-name references to other elements. Each interpretable element definition may also identify the name of a container definition indicating, for example the *type* defining the result of interpreting the body. The use of the model for metadata implies the existence of an "execution" time later than the time when name resolution takes place.

Purely-procedural Interpretables. Interpretable definitions are purely-procedural if they have no execution constraint other than their interpretability; for example, no state. They can be combined and may be rewritten differently for different uses when needed. For example, one can rewrite a method to delegate its call to another method. It is important that references to them be parameterized to achieve different effects on different executions, so references may be qualified by a *signature*, consisting of a sequence of container names, for example for representing parameter types. This signature is considered to be part of the interpretable's name. In dealing with Java programs, for example, methods are examples of purely-procedural interpretables.

Data-bearing Interpretables. Interpretable definitions are data-bearing if they indicate more than just the interpretable material, but also, for example, a place in storage to which values may be assigned. Although there may still be material whose references need to be interpreted, such as an initialization expression, the presence of their additional constraints limits flexibility of composition. For example, unlike methods, one cannot rewrite the value of a field to indicate that the real value is in another place. In dealing with Java programs, for example, fields are treated as data-bearing interpretables, with their bodies interpreted for initialization.

References. Interpretable material may contain references, by name, to itself or other elements within the same space. In dealing with Java programs, for example, references can be found to types, fields and methods.

Methoids. It is frequently the case that a developer performing composition needs to work with constructs within element bodies. For example, if an element body is the text of a paper, there might be a need to compose each page footer with a copyright notice. Or in the coding sphere, it may be that additional behavior is needed whenever the value of some field is written or read; the field access might have been written as a call to a get/set method, available for composition, but it was not. Such "might-have-been" elements can be treated as explicit elements by characterizing them with some pattern to be matched in the body and asserting that occurrences of this pattern should be treated as references to synthesized elements called *methoids*.

An extended treatment of the characterization and handling of methoids is beyond the scope of this paper. The key point, however, is that query, extraction, and composition mechanisms can manipulate content structure *within* element bodies and that doing so is described and effected in the same way as for "regular" elements. An example of the identification of a methoid in this way appears in the next section at 4.1.1[3].

Uninterpreted information: Modifiers and Attributes. Elements may have additional information, not requiring interpretation of references within it. Examples of this information include *modifiers* like "public", "private", "synchronized" in Java, and *attributes*, like "association name" in UML. This information is represented and available for composition, but its composition follows a rather simple model and tends to be handled in ways that depend on the particular kind of artifact being manipulated.

4. Concepts for Describing Composition

To describe composition, it is necessary to identify *what* elements are to be joined, and to specify *how* those elements are to be joined. We do this by means of *correspondences* and *weaving models*, respectively. Together, these make up *weaving directives*. This section describes these concepts, discusses how multiple weaving directives interact, and then discusses the nature of implicit assumptions made by developers using composition, and how those implicit assumptions are made explicit.

4.1. Identifying Correspondences

The first component of a weaving directive establishes elements to be joined. The elements to be identified with one another for composition purposes may be indicated explicitly or implicitly, and a name must be given to the composite result.

The n-tuple of *input elements* that are to be joined, along with the name of the *result element* to be produced by the join, is called a *correspondence*.

4.1.1. **Explicit Identification**. Explicit correspondences result from queries, which simply mention the item by name. Each query produces a set of correspondences. This model allows us to subsume the query capabilities of a variety of existing AOSD languages and tools. Correspondences supporting AspectJ advice consist, for example, of 3 parts: the "base" to which the aspect or advice is being attached, the aspect or advice itself, and the result (which is by default given the name of the "base"). The structure of AspectJ is such that the set of correspondences is formed by applying a single advice to a set of "base" elements indicated by a query, called a "pointcut." Hyper/J's "by-name" matching structure, on the other hand, produces a set of correspondences for elements with matching names.

The query languages employed for identification in the CME is described elsewhere[8]. However, it is important to note that the query processor is capable of forming tuples containing elements matched by a unification-based search. This allows embedding of both AspectJ's queries (pointcuts) and queries required to support Hyper/J's capabilities into a query structure much more powerful than that provided by either. Examples of queries that may be useful for the very small Thermostatic Control example described above are[1]:

[1] **(class basic:*Sensor, alarm:SensorAddition)**
which produces a set of correspondences, each having a pair of inputs consisting of a class in the space "basic" whose name ends with "Sensor" and the class named "SensorAddition" in the space "alarm", and defaulting the result class's name to the one in "basic." The nature of the result class is determined by the weaving model used in the weaving directive.

[2] **(method basic:*Sensor.update(<type>),**
alarm:SensorAddition.update(<type>))
which produces the set of correspondences, each identifying an "update" method of a class whose name ends with "Sensor" in the space "basic" and that takes a single parameter of any type, and the method named "update" in the class "SensorAddition" in the space "alarm" that has the same type signature as the one in "basic," also defaulting the result method's name to the one in "basic." The nature of the result method is determined by the accompanying weaving model.

[3] **(method basic:[set <type> av<suffix>],**
alarm:sensorAddition.update(<type>))
as setAv<suffix>
which produces the set of correspondences, each identifying a "set" methoid for a field whose name starts with "av" in the space "basic" and the method named "update" in the class "SensorAddition" in the space "alarm" that has the same type signature as the type of the field in "basic", with the result method's name based on the name of the field in "basic." The nature of the result is determined by the accompanying weaving model. An actual method can be produced, called in place of each assignment to the variable, or what would be the method body can be inlined at each assignment.

[1] The syntax used here is simply a direct reflection of the underlying concepts for expressing composition, and not meant to be suggested as an actual language in use by any approach to AOSD.

4.1.2. **Implicit Identification.** Implicit correspondences result from implicit elaboration of container definitions. Unless inhibited, a correspondence established between two container definitions also establishes implicit correspondences between members of the containers, so that the resulting containers will contain contents equivalent to the originals. Depending on the developer's expectations (see section 4.4), these correspondences can either apply to like-named members or can simply reflect "copying" of the individual definitions from the inputs in the correspondence. The names assigned in the output are generally the same as those used in the inputs, except where name-clashes arise. As described in section 1.1.1.1, it is possible to diagnose such clashes as erroneous, if desired.

4.2. Weaving Models

The weaving model is the part of a weaving directive that provides directions on how the output named in a correspondence is to be derived from the inputs. There are two fundamental aspects to a weaving model: *selecting* from the inputs, and describing how the selected elements fit into the result's *structure*.

4.2.1. **Selection.** Each selection of inputs is governed by an ordering that applies to the elements that it selects from. We describe selection modes, then ordering.

4.2.1.1. *Selecting Elements from the Inputs.* Not all inputs in a correspondence are necessarily intended to be part of the result. One obvious case of this is often called *override*, where of the inputs is intended to replace another entirely. Another obvious case arises from the desire to indicate that name clashes are not allowed. This selection is called *unique*. Other kinds of selection occur when one of the inputs is selected to be wrapped *around* other members, or when *any* member is slected from a set of equivalent inputs .

A single weaving model can contain multiple selections, such as several override selections applying to different inputs. When no special selection is to apply, the remaining inputs are all selected to be combined. This default is called *merge* selection.

When applied to container definitions, an ambiguity can arise – does the selection apply to the whole container, or are the containers intended to be merged but to have the selection apply to like-named members? As a result, many selection modes are available as pairs, like *override* and *overridemember*.

We make no claim that this list is exhaustive, so development of a generally-useful composition component like that discussed in Section 5 should provide for extensibility of the selection modes made available.

An example of a simple weaving directive that might be used in the Thermostatic Control example described above is:

merge (class basic:*Sensor, alarm:SensorAddition)

which uses a query described in section 4.1.1 and a weaving model using the "merge" selection. It indicates that the material from the SensorAddition class in "alarm" is to be merged with the material from the class in "basic" to make the composite result class.

4.2.1.2. *Ordering the Selected Elements.* The orderings provided for *override* or *around* govern which input overrides or is wrapped around which others. The ordering provided for *merge* applies in the case of interpretable elements, such as methods, and determines the order in which they are combined within the result.

The simplest general model for ordering is that of a *partial order.* The exact manner in which an ordering is specified is another point at which a generally-useful component should try to provide for extensibility. Supposing that names like "before" can be used to indicate previously-defined orderings, an example useful for the Thermostatic Control example is:

merge before
(method basic:*Sensor.update(<type>),
alarm:SensorAddition.update(<type>))

which uses a query described in section 4.1.1 and a weaving model using the "merge" selection, putting the method from the alarm feature before the corresponding method from the basic system.

Since a single weaving model can contain multiple selections, each with its own ordering, it will often be necessary to harmonize independently-specified orderings that apply to the same inputs. This issue will be discussed in section 5.1 where we describe a manner of expressing orderings that we have found particularly useful.

4.2.2. **Specifying the Result Structure**. When a composite if formed from several inputs, there are many issues that may or must be resolved about its *structure* – about how the individual inputs participate in the composite. Though exactly which issues are important to a particular composition approach vary, the ability to specify and control structure is necessary. In one analysis [7] we discuss the issues that concern the identity, the lifetime, and the delegation relationships among participants in a group, but other issues can apply as well, including specialized linkage conventions or the use of particular run-time representations. As with the other aspects of the weaving model, the exact manner in which the structure is established for a correspondence is a point at which a general compositor component should try to provide for extensibility. Using simple names explained below, like *aspect, facet,* or *copy*, to indicate the structure allows us to phrase some of the compositions needed for the Thermostatic Control example:

[1] **merge (class basic:*Sensor as facet,**
feature:SensorAddition as aspect)

which merges the classes as described in the example in section 1.1.1.1, but adds to that the specification that the base classes are to be treated as "facets" – object components with the same lifetime and identity as the composite object itself – while the additions are to be treated as "aspects" having a separate lifetime and identity.

[2] **merge before**
(method basic:*Sensor.update(<t>) as facet,
feature:SensorAddition.update(<t>) as copy)

which merges the methods as described in the example in section 1.1.1.1, but adds to that the specification that the "update" method is to be treated as a "copy" – an equal partner copied from an original, with the copy having the same lifetime and identity as the composite (but, being a copy, not as the original).

4.2.3. **The Weaving Model as a Point of Extension.** The weaving model is a part of the composition specification likely to have great natural variation. Particular needs of an approach may require weaving models that bundle choices together in particular ways. But, as described in section 5, CME's Concern Composition Component directly and openly provides independent choices for selections, orderings and structure.

4.3. Resolving Multiple Weaving Directives

A composite element's characteristics may be specified as part of multiple directives. The need to resolve separate directives requires that it be possible to provide information about how the directives themselves are related. There are two aspects to the relationship among directives for a result element: *precedence* and *exclusivity*. In addition, it is often necessary to indicate characteristics that apply to any output composed of particular inputs. These specifications are called *conditional weaving* directives.

4.3.1. **Precedence**. In this case, it is clear that the second directive is more specific than the first. In other cases, however, it is not clear which directive is more specific. Section 4.2.2's example [1] is a directive that adds the capabilities of the SensorAddition class to all classes with names ending in "Sensor." But the sensor subclasses provided for use in Alaska may need a different addition to accommodate the cold weather. This

can be indicated with a second weaving directive:
merge (class basic:*AlaskaSensor as facet,
alarm:ColdWeatherAddition as aspect)

We cannot simply rely on the circumstance, obvious here, that one specification is narrower than the other, and build this in as a rule, because instances often arise where the question of which of two rules is more general is not quite so clear. For generality's sake we can fall back once again on the use of partial orderings to establish precedence among weaving directives. One way to specify them is by means of "except" clauses. If several directives apply to the same result element, the precedence and exclusivity together determine the outcome.

4.3.2. **Exclusivity**. Not all information about a composition product need be provided by a single directive. In fact, when the queries used to form correspondences are complex, each directive may direct the formation a set of composition products, and the sets produced by different directives may overlap in non-nested ways. Attaching an indication of exclusivity -- one choice from among the three alternatives of *exclusively*, *inclusively* and *initially* -- to each directive allows expression of these relationships.

If the there is a unique highest-precedence directive that is exclusive, it alone is used. Otherwise, if the unique highest-precedence non-inclusive directive is initial, it is used along with any inclusive directives with the same or higher precedence. Otherwise, all inclusive directives are used whose precedence is greater than the unique highest-precedence exclusive directive. If the partial ordering renders any of the above statements undefined, an error is reported.

4.3.3. **Conditional Weaving Directives**. It is necessary on occasion to provide directives that constrain the relationship among the inputs composed to produce a result element without actually describing the result. These are specified by means of *conditional weaving directives.* For example,

whenever (A, B) use before

which specifies that whenever inputs "A" and "B" participate in the same result, as dictated by other directives, they are to be related by the "before" ordering.

4.4. Making Implicit Assumptions Explicit

Prior experience with use of Hyper/J has indicated that, in ways described below, developers have different expectations of a composition tool, reflecting their own knowledge of the software they are manipulating. Failure to take these different expectations into account leads to results which may be expected by some but surprise others. "Software surprise" is a situation to be shunned. This section discusses two of the most common areas of differing expectation, with ways to make the expectations explicit, to avoid surprise.

4.4.1. **Encapsulation**. The implicit correspondence of like-named elements is a great convenience for developers creating new software as extensions of other software or as concurrently-developed features for later, pre-planned integration. On the other hand, these development scenarios presume some level of familiarity with the internal details of the material being composed. The use of composition to produce artifacts for further use by developers (versus for runtime execution only) also makes this presumption. When developers treat software to be composed as "black boxes," however, as they might if it is purchased or subject to change, the presumption that name correspondence has meaning may be entirely inappropriate.

The simplest way of exposing the implicit by-name correspondence assumption is to make it explicit, by means of queries, but this is often too onerous. When the expectation is that the material is co-developed or when the correspondences involve complex uses of precedence, the implicit correspondence of like-named members may suit better. So, to control the application of implicit correspondence, each weaving directive has a property that indicates what level of *encapsulation* the developer expects – are like-named containers to correspond implicitly (type encapsulation), or like-named members (member encapsulation), or nothing at all (space encapsulation)?

4.4.2. **Opacity**. A more subtle skein of issues is illustrated by a small example. The expected result of composing the presumed-corresponding methods shown in the classes in Figure 1 is clear, but not so in Figure 2. Developers who know only about the leaf classes, and have no awareness of the inheritance structure, would expect "aA," as in Figure 1, whereas developers who are fully aware of the inheritance structure would expect just "a."

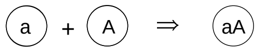

Figure 1 - Clear Composition Expectation

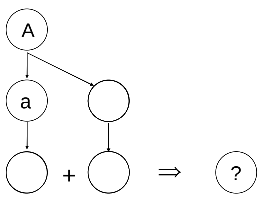

Figure 2 - Unclear Composition Expectation

We can make these implicit expectations explicit by indicating whether a space is *opaque* or *exposed*. For an opaque space, the developer disavows any claim to know how its classes were implemented – the treatment of their members is the same whether they are inherited, implemented, or reimplemented in the class. For an exposed space, the developer expects both to know the implementation structure and that the same knowledge will be used by the composition software.

5. CME's Composition Component

The conceptual base described in section 4 has been used in designing and implementing the Concern Composition Component of CME, called CCC. As discussed above, this component is intended to support a wide variety of approaches to AOSD, and rather directly reflects the concept base. For reasons of length, this paper cannot be a description of the choices made for CCC, and we will not list explicitly the particular alternatives provided for aspects of the weaving model: selections, orders, or structure. However, section 1.1.1.1 contains an implicit claim that a useful, composable, model for order specifications exists that must be supported. In addition, there are implicit assertions that the component can be specialized for tasks other than simply composition, such as extraction of encapsulated source, and that it can be specialized to accommodate the quirks of particular languages like Java. This section is intended to support those assertions, by reference to CCC as an example.

5.1. Specifying and Reconciling Orderings for a Selection

While CCC supports a variety of ways to create and express them, all selection orderings it deals with must ultimately be represented as *combination graphs*. A combination graph has two parts: an *abstract combination graph* and a *population*. An abstract combination graph is a directed acyclic graph, each node of which 1) can be labeled with a name, called its *role* and 2) can be ”pre-filled” with predefined content, such as a fixed library class to include. If there are multiple nodes with the same name (including unlabelled), they must all have the same in and out edges; this ensures that all nodes for a role are treated uniformly. The population maps graph nodes to selected input elements. Not all graph nodes need be mapped. *Method combination graphs* are a specialized form of combination graph with additional information attached to each edge, such as conditions for following the edge based on the value returned by (or the exception thrown by) the method that is mapped to the node from which the edge emanates. The method combination graphs in the CME Concern Assembly Toolkit [9] also realize this concept.

Figure 3 shows an abstract combination graph called PrePost, which represents the the composition of a method with precondition and postcondition checks.

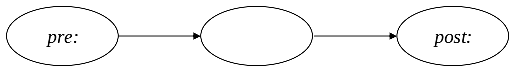

Figure 3 - PrePost Abstract Combination Graph

One combination graph using it might have the population (pre:A, :B). Another might have (:B, post:C). As described in sections 4.3.2 and 4.3.3, a result can be created according to several weaving directives. It must be possible to construct from them a merged combination graph that embeds all of them within it. In the case just mentioned, that would be a PrePost combination graph with the population: (pre:A, :B, post:C).

The ability to merge combination graphs is one of the driving reasons for adopting this form of ordering. The constraints governing the result of the merge are:

1) Each node in any input graph is assigned a node in the result graph with the same population and role (if specified).
2) Nodes of input graphs that are populated with the same input are assigned the same result node.
3) Pre-filled nodes with the same roles specified for them are expected to have the same contents, and are assigned the same result node.
4) Other nodes of input graphs are assigned different nodes.
5) If there is an edge between two nodes in an input graph, there is an edge between the nodes assigned them in the result.
6) If there an edge of a node with a specified role in any input graph to or from another node, there is an edge of each result node with that specified role to or from the node assigned the other node.
7) The resulting graph must be a valid combination graph – i.e. it must be a directed acyclic graph. Additional constraints may apply to merging specializations like method combination graphs.

These constraints produce the result described for PrePost, above, or, for example, the more complex result of combining the two graphs in Figure 4, shown in Figure 5. (Primed edges are introduced by rule 6.)

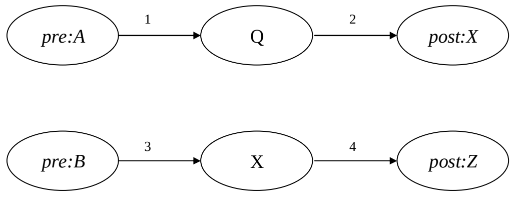

Figure 4 - Two Combination Graphs

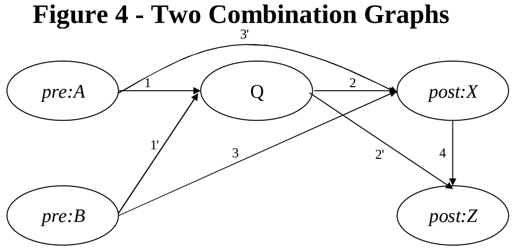

Figure 5 -- Combined Combination Graph

5.2. Specializing CCC for a purpose

Not all behavior needed for a tool like extraction or composition is embedded in CCC. The tool may countermand or supplement decisions made by the "natural" composition process, for example to add declarations for referenced elements during extraction. The tool may have additional implicit rules that, for example, preserve characteristics important to its approach, and it may have linguistic structures that use subsets of the weaving model, provide particular orderings or graphs, or that apply the directives with specifically orchestrated precedences. These characteristics of a tool are expected to be provided by extending the CCC implementation class. This extension can also provide implementations invoked at important predefined points of processing flexibility.

5.3. Accounting for artifact peculiarities

The process of tailoring CCC's behavior at these points is called *rectification*. In addition to its use to adapt the component to various functional needs, it is used to "make right" the composite result. The target system for the composition activity may have linguistic rules, like prohibition of multiple inheritance, that must be applied to the natural composition result. The rectification plug-in for Java, for instance, adapts for multiple inheritance, for construction protocols, and for proper behavior of the "instanceof" operator. The analysis and transformation needed for Java rectification presents a collection of significant technical problems of its own, and is left as the topic of a separate paper.

6. Supporting Existing Approaches

Treating the compositor as a component facilitates the definition, implementation, integration, and comparison of a wide spectrum of aspect-oriented languages, formalisms, and paradigms. To help demonstrate how this can be done, this section briefly describes the mapping of constructs contained in some existing aspect-oriented approaches to the core composition concepts described in this paper. Due to space constraints, we have not exhaustively elaborated the full mappings here, but rather, we highlight the mappings of some particularly interesting and key features of these approaches to CCC.

6.1. Hyper/J

Hyper/J[17] supports the representation and composition of concerns whose contents are standard Java classes and interfaces. The concerns may overlap, in the sense that multiple concerns may contain definitions for corresponding classes, interfaces, or members. It uses *composition relationships* to specify correspondences and the manner of composition of the Java material. The composition relationships are specified separately from the concerns, in a manner analogous to that of module interconnection formalisms. Composition involves the integration of multiple Java type hierarchies in a way that satisfies the composition relationships, to produce a set of composed Java types that contain the woven material.

Hyper/J's concerns map to CCC *spaces* (Section 0), containing Java classes and interfaces. Hyper/J's joinpoints are classes, interfaces, and their members (fields, operations, constructors, and types).

The composition relationships in Hyper/J specify a wide variety of weaving directives. Some control the establishment of correspondences. *NonCorresponding* and *ByName* specify whether like-named elements of related concerns should correspond. These map to CCC's space- and member-encapsulation (Section 4.4.1) mechanisms, respectively. Other composition relationships indicate how corresponding elements should be integrated. *Merge* and *override* map directly to CCC's *merge* and *overridemember* selection modes (Section 1.1.1.1) and the *facet* result structure (Section 4.2.2). The execution order of merged elements is specified with *before* and *after* order constraints on correspondences. These are expressed using CCC's combination graphs (Section 5.1). *Bracket* specifies a "before" method and an "after" method for the same set of inputs, and would be defined as a new, predefined method combination graph) in CCC. The before and after methods are composed with the *copy* structure, allowing them to bracket many different methods. Hyper/J also supports *summary functions*, methods whose parameters are the return values of a set of composed methods, and which perform some computation over them, returning a single value as the result of the composed method. An example summary function is boolean "and," which returns true if all the composed methods return true. Summary functions are realized in CCC with method combination graphs. Edges exiting from nodes in these graphs can contain "accumulator variables," and each node can add a value to the accumulator. At the end is a method node that calls the summary function, passing it the accumulator.

6.2. AspectJ

AspectJ [12] is a Java extension that adds the *aspect* construct to represent concerns that cut across multiple Java classes. Aspects are class-like entities that can define their own behavior and state (standard Java fields and methods), behavior and state to be introduced into other classes (*intertype declarations*), and *advice* to be attached as specified by *pointcuts* (queries that identify the applicable join points). Advice constructs can be treated as weaving directives coupled with the code (represented as methods) that is to be woven with Java methods. Weaving involves the insertion of code to attach aspect objects to Java objects and to trigger advice, so as to satisfiy the advice and other weaving directives, notably, *declare* specifications.

The types (aspects, classes, and interfaces) that are to be woven are listed in AspectJ's "`.lst`" files. Each "`.lst`" file specifies a single CCC input space (Section 3), containing the set of types to be composed.

AspectJ's pointcuts describe execution-time events, but these events occur at a set of points in the program's static structure. As noted in Section 1.1, the generation of code to collect runtime information or perform runtime tests on dynamic state is an activity separate from composition. The composition activity involves joining that code, which AspectJ compiler produces, together with the applicable aspect code, at the relevant points in the program's static structure [12]. These points are specified in *correspondences* (Section 4.1.1) in CCC, using CME's query facility.

If an advice is to be woven at some point in a class, the aspect containing the advice is woven with the class itself, using a specialization of CCC's *aspect attachment* [7] structure. This means the aspect is represented as a separate object with separate identity from the "base" object(s) to which it is attached. The lifetime of the aspect attachment depends on the aspect's "per" specification. By default, the lifetime is CCC's *singleton*, meaning that there is one aspect instance for all of the classes with which the aspect is woven. The other AspectJ "per" specifications—percflow, percflowbelow, perthis, and pertarget—are specified as having CCC's *dynamic* lifetime. All of these specifications depend on dynamic residue, which is, as noted earlier, treated separately from composition.

AspectJ supports three types of advice: *before*, *after*, and *around*. The before and after advice from an aspect must be run as a "bracket" around the advised join point, so the same method combination graph solution is used as for Hyper/J's *bracket* directive (Section 6.1). Two variants of after advice, *after throwing* and *after returning*, are realized using edge conditions in these method combination graphs. Around advice is not simply an ordering constraint, but rather, a different selection mode, called *around*, causing the advice to be "wrapped around" another element. Around advice can include a special language construct, *proceed()*, which executes the wrapped element. One common implementation of proceed() [10] employs *AroundClosure* objects which are created, passed, and used in the composed code. The run-time conventions appropriate to the continuation-related code is specific to the chosen implementation of AspectJ, and therefore, it is realized in CCC as part of AspectJ's rectification (Section 5.3).

Most of AspectJ's *declare* specifications and intertype declarations are handled as compile-time checks or by treating them as though they were written as Java classes with the desired characteristics (parents, fields, etc.) and composed using "merge," as described in Section 6.1. One exception is *declare precedence*, which specifies order constraints that apply to the weaving of advice. If aspect A2 is declared to have precedence over aspect A1, then at any join point, *j*, that both aspects advise, the order is: [$before_{A2}$, -$before_{A1}$, j, $after_{A1}$, $after_{A2}$]. The method combination graph would be generated to ensure the required precedence semantics.

6.3. Other Major AOSD Technologies

A number of other major AOSD technologies and languages exist, particularly for implementing aspect-oriented code. Space constraints preclude additional detailed mappings of these technologies to CCC, but we believe that the key features of all of them are covered by the descriptions of AspectJ and Hyper/J. For example, AspectWerkz [1]and JBoss [14] support an AspectJ-like composition model, but they differ in their specification languages (standard Java, with XML or tags for specifying composition), and both provide more extensive support for dynamic attachment of aspects. Support for of dynamic responses to events requires little accommodation by a composition engine, so these distinctions do not significantly affect the mapping to CCC. Mixin layers [16] are mapped to CCC like Hyper/J's concerns, with corresponding classes in different layers combined using *around* wrapping and support for *super()* in extensions that resembles the support for *proceed()* for AspectJ. Composition filters [2] are realized in a way similar to AspectJ, but with different attachment semantics and a variety of method combination graphs to realize different filter semantics. Detailed mappings of these and other important AOSD technologies and paradigms is left for future papers.

7. Summary and Related Work

This paper presented a base of concepts suitable for expressing composition of artifacts in a general setting independent from language or AOSD approach. It provides examples of many useful choices that can be provided or used in particular cases. It offers additional material about an existing open-source implementation based on this concept base, and shows how that implementation can be used to realize the composition needs of several existing AOSD approaches.

There are existing tools for manipulation of Java classes, usually at load time, such as Javassist [3], JMangler [13], JOIE [4] and Binary Component Adaptation (BCA) [11]. These operate at a lower level than the kind of composition engine described above, generally applying to single types or methods, only combining at the level of types and not providing direct support for the combination of interpretable elements discussed above.

Section 6 also discussed other AOSD technologies that provide a high-level approach and embed a composition engine to perform their composition needs. But these existing tools and approaches, while making use of individual particularizations of the concepts presented here (like selection or ordering), do not treat them as concepts subject to the abstraction needed for creating a generalized composition engine. Furthermore, these systems are all specific to single kinds of artifacts like Java bytecodes or source. This paper has identified and described key concepts needed to address composition of the variety of artifacts encountered throughout the development lifecycle.

8. References

[1] AspectWerkz web site, http:aspectwerkz.codehaus.org

[2] M. Aksit, L. Bergmans and S. Vural. "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach." Proc. European Conference on Object-Oriented Programming, 1992.

[3] S. Chiba, "Load-time Structural Reflection in Java." Proc. 2000 European Conference on Object Oriented Programming, LNCS 1850, Springer Verlag, 2000

[4] G. Cohen and J. Chase, "Automatic Program Transformation with JOIE", USENIX Annual Technical Conference, June, 1998

[5] Concern Manipulation Environment web site, http://www.eclipse.org/cme

[6] W. Harrison and H. Ossher. "Subject-Oriented Programming: A Critique of Pure Objects." Proc. 8th conference on Object-oriented programming systems, languages, and applications, 411-428 (1993).

[7] W. Harrison and H. Ossher, "Member-Group Relationships Among Objects", at Workshops on Foundations of Aspect Languages, on Aspect-Oriented Design, and on UML in Aspect-Oriented Software at International Conference on Aspect-Oriented Software Development, March 2002

[8] W. Harrison, H. Ossher, P. Tarr, "Pervasive Query Support in the Concern Manipulation Environment", *submitted for ICSE'05*.

[9] W.H. Harrison, H.L. Ossher, P.L. Tarr, V. Kruskal, F. Tip, "CAT: A Toolkit for Assembling Concerns" IBM Research Report RC22686, December, 2002

[10] E. Hillsdale and J. Hugunin, "Advice Weaving in AspectJ", Proc. 3rd International Conference on Aspect-Oriented Software Development, 26-35 (2004)

[11] R. Keller, U. Hölzle, "Binary Component Adaptation," Proc. 1998 European Conference on Object Oriented Programming, LNCS 1445, Springer Verlag, 1998.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, Jeffrey Palm and William G. Griswold. "An Overview of AspectJ." Proc. 15th European Conference on Object-Oriented Programming, 327-353 (2001).

[13] G. Kniesel, P. Constanza, M. Austermann, "JMangler – A Framework for Load-Time Transformation of Java Class Files, November 2001. IEEE Workshop on Source Code Analysis and Manipulation (SCAM),

[14] JBOSS web page, http://www.jboss.org

[15] M. Mezini and K. Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development." Proc. Conference on Object-oriented Programming: Systems, Languages, and Applications, 1998.

[16] Smaragdakis and Batory, Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs, ACM Transactions on Software Engineering and Methodology, April 2002.

[17] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton Jr. "N Degrees of Separation: Multi-Dimensional Separation of Concerns." Proc. 21st International Conference on Software Engineering, 107-119 (1999).