RC22686 (W0212-148) December 30, 2002
Computer Science

IBM Research Report

CAT: A Toolkit for Assembling Concerns

William H. Harrison, Harold L. Ossher,
Peri L. Tarr, Vincent Kruskal, Frank Tip
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

CAT: A Toolkit for Assembling Concerns

William Harrison, Harold Ossher, Peri Tarr, Vincent Kruskal and Frank Tip

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
{harrisn, ossher, tarr, kruskal, tip}@watson.ibm.com

1 Introduction

Aspect-oriented software development (AOSD) includes separating concerns, and then composing or weaving them to produce software that realizes appropriately-combined behavior. This might involve composition of two or more class hierarchies, for example, or weaving of aspect code into base code. Composition or weaving can be defined, specified and implemented in a wide variety of ways, and can be applied to software artifacts of various kinds. Yet, despite these differences, there are important similarities. The thesis of this paper is that an abstraction layer and toolkit supporting low-level *concern assembly* would enable AOSD tool builders to leverage many of these similarities, reducing development effort and increasing the scope and interoperation of their tools.

Concern assembly is the low-level, detailed manipulation of individual classes, members, and even chunks of code within members, that are needed to accomplish composition or weaving. In addition to the natural language meaning of "assembly," the connotations associated with term "assembly language" are intentional. Humans do not work at this level, but many AOSD tools need to perform concern assembly in their back ends.

This paper describes a *concern assembly toolkit* (CAT) that provides abstractions and support for concern assembly. The toolkit is intended:

- to provide a common set of abstractions, including directives for specifying assembly, that suits a variety of purposes and of AOSD approaches,
- to enable support for applying assembly to different software representations and artifacts,
- to support a variety of implementation strategies for AOSD, and
- to be an open framework, capable of extension to accommodate new approaches, artifacts and implementation strategies.

These goals are now elaborated.

For a variety of purposes and AOSD approaches: Many approaches describe composition or weaving using high-level linguistic constructs that involve pattern matching and sets. For example, Hyper/J [18] uses composition relationships such as mergeByName, which involves name and type matching, and AspectJ [12] uses pointcut designators, which are expressed as patterns. Each such high-level specification boils down to (usually many) assembly operations. Tools that implement these approaches must, in effect, expand the high-level specification into a set of lower-level assembly operations, and then apply those. So the first goal of our toolkit is to provide a common set of assembly directives that are suitable for realizing a wide variety of high-level approaches. They are provided both as Java interfaces and through an XML syntax. They provide a common basis for different approaches, which we hope can lead to interoperation and also to greater understanding of their similarities and differences. They also provide a common and suitable representation for tools to analyze, check and display the results of composition, such as applying program analysis techniques to check for behavioral interference [19]. Making these assembly directives explicit is also useful during tool development and debugging, because it becomes possible to see just what the tool is doing, at a level that is more convenient than examination of actual code (especially binary code).

For a variety of artifacts: In the absence of specialized runtime virtual machines, code composition is achieved by assembling fragments expressed in some existing language. Whether this is done as a separate tool, as a preprocessor, or within a compiler, some form of assembly is taking place. Even in systems that support runtime manipulation of aspects in standard runtime systems, some form of assembly is needed to prepare objects for dynamic aspect manipulation (e.g., addition of aspect pointers or role tables). Assembly can be applied to many different representations of software, such as Java source code, Java class files and bytecode streams, and to artifacts other than code, such as UML diagrams. Different representations are appropriate in different contexts, and, ideally, AOSD tools should handle multiple representations consistently. Most of a tool's user-level functionality does not depend on the representation being used, but performing the detailed assembly is representation specific. This second goal of our toolkit is then to provide a common interface defining the assembly abstractions, which can be implemented by pluggable *concern assemblers* for a variety of representations, allowing the tools that build upon it to ignore representation details. In developing the abstractions, we have been working both with Java and UML class diagrams. We expect that the abstractions will also be applicable to other object-oriented languages, such as C# and Smalltalk.

For a variety of implementations: Even ignoring software representation, there are various ways of implementing composition or weaving to achieve a desired result. For example, tracing code can be inserted directly into the code being traced, or calls to it can be inserted, or new "stub" methods can be generated that substitute for original methods and call the tracing methods and the original methods. Developers of AOSD tools might want to experiment with such different approaches, and might even

want to make choices available to their users. Implementing these on even a single software representation is onerous. The third goal of our composition toolkit, then, is that such different approaches be convenient to specify in terms of the assembly directives.

Via an open framework: We currently have concern assemblers for Java class files and UML class diagrams represented as XMI files. We also have a "serializer" assembler that produces the XML form of the assembly directives, and an XML reader that consumes it. This is just a start. The toolkit is intended to be an open framework, permitting new and different concern assemblers to be developed and plugged in. These will often be facades on other, existing toolkits, such as those that support manipulation of Java class files (e.g., [15][3]). For example, our assembler for Java class files is built on the JikesBT bytecode manipulation toolkit [12].

The rest of this paper describes concern assembly with CAT, using a running example introduced in Section 2 for illustration throughout. Sections 3, 4 and 5 describe the assembly abstractions and directives, and Section 6 describes the structure of the toolkit itself. Section 7 evaluates the toolkit by discussing how it can be used to implement some representative AOSD approaches and implementation strategies. The paper concludes with discussions of related and future work.

2 Introducing an Example

Consider a small example based on a piece of software that aids in the filing and processing of disability insurance claims, abridged from [5]. One concern defines the class `ClaimDoc`, which is created by/for claimants and goes through several processing steps. In this illustration, primitive access control must be added to claim documents. The access control protects the claim document to ensure that only authorized role-players invoke certain operations, specifically: Claim Assessors can invoke `approveClaim()`, and Claimants can invoke `setClaimRequest`, `setName`, `setID`, and set the `address` variable.[1] If one of these is invoked by the wrong role-player, an exception will be thrown. The problem thus has two concerns, which we can imagine as the Java shown in Figure 1.

The Claim Document Concern:

```
Class ClaimDoc {
  void approveClaim() {...}
  void setClaimRequest(float amount) {...}
  void setName(String name) {...}
  String name() {...}
  void setID(int id) {...}
  int iD() {...}
  String address;
}
```

The Protected Claim Document Concern:

```
Class ProtectedClaimDoc {
  boolean isClaimAssessor(){...}
  boolean isClaimant() {...}
}
```

Figure 1: Example Concerns to be Assembled

The objective of concern assembly is to compose these two concerns such that the appropriate test is applied before each protected method. We illustrate this is using the Hyper/J approach, in which the two classes are composed to form a single composed class, which is used at runtime. A (slightly simplified) extract of the desired composed class is shown in Figure 2.

```
Class ClaimDoc {
  void originalApproveClaim() {
    /* copy of input ClaimDoc.approveClaim */ }
  void isClaimAssessor() {
    /* copy of input
       ProtectedClaimDoc. isClaimAssessor */ }
  void approveClaim() {
    if (isClaimAssessor())
       originalApproveClaim();
    else
        throw new ProtectionError();
    }
  …
  String address;
}
```

Figure 2: Example Assembled Result (Extract)

3 The Assembly Process

With the Concern Assembly Toolkit, a *concern assembler* processes software artifacts in input concerns, to produce separate, composed artifacts in output concerns, as illustrated in Figure 3. For example, a concern assembler for Java binary would process input "class" files and produce a new set of output, composed class files. The composed software is created partly by copying and transforming input software elements and partly by synthesizing new software elements. *Assembly directives* control the operation of the concern assembler, specifying the details of the composition desired. They can be written in the *concern assembly language*, a usage of XML. Alternatively, they can be issued as calls to a Java API implemented by all concern assemblers. In either case, there is a *concern assembly protocol* that constrains the ordering of directives to ensure proper operation of concern assemblers.

We confine our attention to object-oriented software, though the approach, and even many of the details, also apply to other paradigms. To support multiple object-oriented artifact representations, including different languages (e.g., UML and Java), the directives are designed to apply to the object-oriented family of languages as a whole, often by allowing open-ended specification of elements like signatures and modifiers as strings. Except for basic structure, such as comma-separated lists, of the details of these strings are not laid down by the interface.

Concern assembly thus clearly involves manipulation of object-oriented software elements, such as classes and members. In addition, we introduce three other kinds of elements:

- *organizational* elements, which provide for grouping and organization of the other elements,
- *method combination graphs*, which support specification of control flow among combined methods, and

[1] The original example used getter/setter methods for address also; we changed it to enable us to illustrate intra-method assembly in Section 5.

- *methoids*, which allow chunks of material within method bodies to be treated as methods.

The rest of this section describes the object-oriented software and organizational elements, along with associated directives and protocol. Method combination graphs are described in Section 4 and methoids in Section 5. These descriptions all reference Figure 4, which contains an extract of the XML assembly directives needed for the example introduced in Section 2. Descriptions are informal and not fully detailed, but should serve to convey and illustrate the key concepts. Full or formal definition is beyond the scope of this paper.

3.1 Universes of Type Spaces

Concern assembly takes place in a *universe*, managed by a single concern assembler. A universe contains a collection of named *input type spaces* and *output type spaces*, each containing types (e.g., classes, interfaces, and primitives). An input type space is associated with some existing source of definitions for the material to be composed, such as a Java classpath or a UML/XMI diagram. An output type space is generally an empty space into which the assembled result will be put, associated with a target container, such as a directory.

Assembly directives specify input and output type spaces and their associated directories or other external structures (e.g., Figure 4, lines 4, 14 and 18). These are specified as strings, to which no specific syntax or meaning are ascribed; each assembler interprets them as appropriate. A directive can also specify a single place to find types that are common to all the spaces (e.g., Java library classes, line 2).

Each type space is a closed collection of named types: interpreting a reference not defined within the space is invalid. The elements in a type space are treated as though stored in a "flat" name space, using name qualification to denote containment. Specific concern assemblers are free to store types in flat or nested fashion.

3.2 Types

There is a directive to name a type in an input type space, indicating that is to be used (lines 5, 14) and a directive to create a type in an output type space (line 19). The latter is given *modifiers*, like "public", "interface", "abstract", etc., but is created without any contained members.

Different object-oriented languages support different modifiers. Modifiers in directives are therefore treated as an open-ended set of keywords to be interpreted by specific assemblers.

3.3 Members

Types have members of two kinds: fields and methods. Each member has a name and a type-characterization, which together must be unique within the type. Each member also has modifiers and, perhaps, a body. In Java, field bodies are the initialization expressions, executed when the field is initialized, and method bodies are executed when the method is invoked. In UML, the bodies may specify OCL constraints [20].

There are two kinds of directives for creating members: creation-from-scratch, and creation-by-copy. With creation-from-scratch, the directive completely specifies the new member by providing the name, type-characterization, modifiers, and other information. In the case of methods that are not abstract, the body is specified using method combination graphs, described in Section 4 (lines 48–50). With creation-by-copy, the new member specification is augmented with a reference to a similar member in an input space, from which the body and additional modifier information are drawn. For example, field copy is shown in lines 32–34, method copy without renaming in lines 42–45, and with renaming in lines 37–39.

Method modifiers, like type modifiers, are an open-ended set of keywords. Type characterizations are also open-ended strings, to accommodate multiple languages, but with some restrictions (e.g., comma-separated lists for method signatures).

3.4 Mappings

Field and method bodies contain references to other types and members. When material that contains references is copied from an input space to an output space, the references must be altered to refer to appropriate elements in the output space. The translation from input to output elements is specified in *mappings*. A global mapping is associated with each output space, for use when copying material into types in that space. In some cases, such as when crosscutting is implemented by copying the same material (e.g., logging or synchronization code) into multiple classes, the desired translation may vary depending on the specific output type into which the material is copied. To accommodate this need, a local submapping is also associated with each output type, and takes precedence over the global mapping.

Directives are employed to add to the mappings the individual translations of input types and members to their corresponding output types and members. For example, global type mappings are shown in lines 23–26, and local member submappings in lines 54–67.

3.5 Relationships

Relationships, such as extends and implements between types and throws between methods and types are specified by means of separate directives, rather than as part of the definitions of the related entities (lines 70–72).

3.6 Protocol

It is important that the construction of output types be clear, unambiguous and well-formed. While it is, strictly speaking, only necessary to ensure that the mapping for a type or member is specified just prior to the first time it is interpreted and that it be unchanged after that first reference, we believe a slightly more restrictive protocol will avoid bugs and confusion. The directives are therefore constrained to be used in the following sequence:

1. Type-creation directives (lines 2–21)
2. Mapping-construction directives for type translations (lines 23–26)
3. Member-construction directives (lines 30–52)
4. Mapping-construction directives for member translations (lines 54–67)
5. Relationships creation directives (lines 70–72)

6. Copying and translation of member bodies (performed implicitly by the language processor at the end).

4 Method Combination Graphs

Creation-from-scratch of new methods can either create an abstract method, which has no body, or a method whose body combines other methods. The latter is specified by means of a *method combination graph*, which consists of:

- a set of declarations for method combination graph variables,
- a set of nodes, each of which has an operation (a single method call, a reference to a variable, or an exit), and
- a set of edges with conditions, controlling the order and circumstances under which the nodes are interpreted.

Figure 5 shows the method combination graph describing the combination of `isClaimAssesor` with `approveClaim` discussed in Section 2. The possible flows should be clear from the graph. If `approveClaim` is executed, its result is stored in `v`, whether normal or exception, so that the whole graph can exit in exactly the same way that `approveClaim` did.

Nodes are interpreted in any order, subject to constraints imposed by the edges. A method call node specifies a call to a method, complete with target object and arguments. No two nodes in a single graph may call the same method with the same parameters, allowing call nodes to be identified by their calls. When a call node is interpreted, the associated method call is executed and its result is noted as either a normal value or an exception value, depending on whether the method returned normally or threw an exception. The expressions available for use in calls include "this" and its instance variables and static fields, "super", the parameters of the method being defined by the graph and thus provided by its caller, method combination graph variables, literals and special variables containing reflective information, such as class name, method name or packaged arguments. When needed, a special *proceed object* parameter is available to implement "around" wrappers that are usable in many different contexts. It refers to the wrapped method and packages up arguments for use when called. The method to which it is passed can call its `proceed` method to have the wrapped method executed.

When a variable node is interpreted, its value is noted as a normal value. When an exit node is interpreted, an exit from the entire graph takes place; the expression in the exit node specifies the value to be returned or the exception to be thrown.

Each edge connects a predecessor node with a successor node and identifies a condition governing the connection. Cycles are not allowed. The presence of an edge has two effects: it indicates that the successor cannot be interpreted while its predecessor is yet to be interpreted and it indicates that, if the condition is true after executing the predecessor, then the successor should eventually be interpreted. The condition tests the value noted upon interpretation of the predecessor node, and the edge may also indicate a method combination graph variable in which this value is to be recorded. The conditions are not arbitrary predicates, but are constrained to be easy to analyze and fast to execute. They include "*" (true), *returned*, true for any normal exit, *threw T*, true if an exception of type *T* was thrown, and tests for the values of Booleans, integers and strings, the signs of integers, and the nullity of object references. If more complex conditions are required, they can be encoded in methods and evaluated in method call nodes, the results of which can then be used in edge conditions. It is interesting to note that manifesting types or other dispatch criteria as integers or strings allows method combination graphs to be used to accomplish multiple dispatch.

Variables are of two kinds: dual variables or accumulator variables. A dual variable can hold both an exception value and a normal value. Exiting with reference to a dual variable will either throw the exception or return the value, depending on the last value assigned to it. Accumulator variables allow a succession of normal values from various nodes to be accumulated and passed to a method for reduction to a single value before exiting.

Method combination graphs provide a way of describing the results of a composition that are themselves suitable as input for later re-composition. As such they provide for describing the constraints on the flow among methods without over-constraining it, as would be done if combination were expressed directly in Java or in a deterministic graph.

It is not intended that method combination graphs be interpreted directly during execution of the composed software. Rather, the concern assembler can generate method bodies with the semantics specified by the graphs, and has the opportunity to optimize them. A method call node can be annotated to indicate that the method should be generated in-line. In addition, an in-line annotation can be associated with the graph itself, indicating that the code generated for the graph, including any in-lined methods it calls, is to be placed in-line at all invocation sites. Except as indicated in Section 4, in-lining annotations are assertions of preference and do not require that the in-lining be performed.

5 Extracted Methods and Methoids

The capabilities described so far permit assembly of object-oriented software at member granularity. As long as all concerns of interest are realized as sets of entire methods, with no concern tangling within methods, this is adequate. Unfortunately, it is not often enough the case. Especially when dealing with additional concerns not apparent or of interest to the original developer, there are situations in which chunks of code within method bodies need to be augmented.[2] For example, implementation of a first-failure data capture concern, to record useful information about failures as early as possible, might include the need to augment throw statements with additional function, and perhaps even instance variable sets. Some of the join points in AspectJ [1], such as calls, throws, catches, gets and sets, address this need.

[2] Inserting new code at some point within existing code is considered augmentation of the chunk of code before or after that point, as appropriate.

Our approach to this issue is to consider such chunks of code to be potential methods, which we call *methoids*. Once the desired methoids have been identified within the code in input type spaces, they can, in principle, be treated as methods for assembly purposes, being mapped to output methods or methoids as desired and with composite behavior specified by means of method combination graphs.

There is a practical restriction, however. Method composition is symmetrical: a given method can have its behavior augmented by other methods (so that they are executed whenever it is), or it can be itself be used to augment the behavior of other methods (so that it is executed whenever they are). In the case of a methoid, which originates as in-line code, augmenting its behavior can be accomplished by inserting code around it in place, but using it to augment the behavior of other methods requires that it actually be extracted as a real method.[3] In most of the commonly used cases this is easy, but general method extraction is difficult, and the likelihood that an extracted method will be widely useful decreases with the complexity of its characterization which can manifest itself as the size and novelty of its signature or by the number of free variables read and written within it. We therefore support both *free* methoids, which can be extracted, and *tied* methoids, which cannot. Free methoids can be composed in any way, whereas tied methoids are asymmetrical: their behavior can be augmented by other metho(i)ds, but can not be included in augmenting behavior elsewhere.

Directives identify and name methoids within input types, specifying whether they are free or tied. Characterizing methoids is a complex issue. Generally speaking, a regular method is characterized by its name and signature and by an informal description of its intent. The signature size is generally small, especially in relation to the length of the body, and developers try to avoid entangling too many concerns in a single method. We should expect the same of methoids. They are also characterized by their names and signatures, but as the developer has not separated and named them, the directives must specify them in some other way. There is a wide spectrum of possibilities here, from linguistic primitives to program-slicing [21] and other program refactoring technologies [6], each of which imposes requirements on the concern assemblers that must implement them. Experience with AspectJ[1] and HyperProbe [15] indicate that the linguistically-defined primitives, like get/set of instance variables, used of instanceof, throws, method calls, entries and exits, synchronization block entries and exits and catch blocks, are necessary as methoids, cover many important cases, and can be detected without significant analysis. These are therefore included in the initial definitions of the methoid directives, although the methoid characterizations remain open to accommodate future expansion.

For example, lines 6–11 in Figure 4define a free methoid called `setAddress` within the `ClaimDoc` input class. It's kind is `set`, indicating that it encapsulates sets of an instance variable, `address` of type `java.lang.String` in this case, as specified by the properties on lines 8–9. Concern assemblers built on CAT can register handlers for different kinds of methoids, including new kinds introduced by them, and each handler can support whatever properties it desires.

The restrictions inherent in tied methods constrain the assembly directives that apply to them. As with characterization, the circumstances under which a methoid can be extracted presents a spectrum of possibilities, trading increasing flexibility in recomposition (symmetry) against the effort required from the concern assemblers. The toolkit therefore gives concern assemblers the freedom to constrain which methoids may be declared as free. In general, it is expected that simple methoids, like get/set and throws, can be free, whereas those involving arbitrary blocks of code, like catch blocks, must be tied. A method copy directive applied to a tied methoid may only copy it into an output class into which the input method containing it is also copied. Only it can be mapped to such a copy, or to any method combination graph that invokes such a copy. These restrictions ensure asymmetrical usage of tied methoids, discussed above.

6 The Concern Assembly Toolkit

The concern assembly toolkit is intended to be open, encouraging expansion, addition, and exploitation. As illustrated in Figure 6, the toolkit consists of interfaces, frameworks, and concern assemblers. The interfaces are for use by exploiters, usually higher-level AOSD tools, and are implemented by the concern assemblers. Concern assemblers often make use of outside toolkits for manipulating artifacts. The frameworks are intended to lessen the work involved in writing new concern assemblers by providing significant functionality for dealing with the new concepts that are not expected to be supported by such toolkits.

6.1 Interfaces

Exploiters use the composition toolkit through either of two interfaces: a Java API and an external XML format called the Concern Assembly Language. Exploiters thus have the flexibility of invoking it directly or of producing and saving XML to be used later.

The Java API is a collection of Java interfaces through which assembly directives can be issued. It can be thought of in four parts for defining and manipulating the different kinds of elements introduced in Section 3: object-oriented software elements, organizational elements (e.g., type spaces), methoids and method combination graphs. The part supporting object-oriented software elements provides a uniform, language-neutral interface to constructs found in most OO languages and systems, and Java in particular. Experience has shown the importance of articulating toolkits as a collection of pure interfaces, avoiding abstract and concrete classes whose presence would reduce flexibility when providing alternative implementations.

The XML provides both a persistent form in which assembly directives can be saved and a form suitable for direct manipulation with various XML visualization and editing tools. The toolkit provides a parser, the Concern Assembly Language Processor, which processes directives in XML format and makes calls on the API to execute them.

[3] Alternatively, it could be copied from its context and transformed for reuse in the new context, but that is no easier than extracting it as a method.

6.2 Concern assemblers

Concern assemblers are the "plug-in" components that implement the toolkit interfaces for various representations of artifacts to be assembled. We currently have assemblers for Java class files and UML class diagrams represented as XMI files, and an XML serialization assembler, which stores the directives it receives as a concern assembly language file. We plan to add further assemblers, including for Java source code and one load-time assembly of Java bytecodes, and the toolkit supports addition of assemblers by third parties.

Much of the guts of a concern assembler, especially the support for manipulating object-oriented software elements, is specific to the artifact representation. Fortunately, other toolkits are often available for general-purpose manipulation of such representations, and concern assemblers can be built as façades on these. For example, our Java class file assembler is written as a collection of subclasses of classes provided by JikesBT, a toolkit for manipulating class files that supports extensions via subclassing and abstract factories. Our UML assembler is written as a set of classes that are composed using Hyper/J with an XMI model manipulator generated from the UML standard specifications using Tengger, our tool for feature-based design [1]. These assemblers also build upon the frameworks described below. In both cases, the size of the façade is small, validating the usefulness of the API and framework definitions in producing new concern assemblers for various artifacts

6.3 Frameworks

Some functionality within an assembler is much less representation-specific, such as handling of organizational elements and construction and processing of methoids and method combination graphs. The Concern Assembly Framework implements this functionality as cascaded packages, successively providing deeper support over a narrower focus. A few of the classes in these packages are abstract classes that depend on concrete subclasses filling in the few representation-specific details.

The Concern Assembly Framework includes the analysis and target-independent optimization parts of a method combination graph compiler, but cannot include the representation-specific generation part. One useful way of aiding the developers of many assemblers, however, is to produce Java source code corresponding to the graphs. We therefore also provide the Method Graph Java Source Generator Framework, cascaded on the Concern Assembly Framework. Producing Java source can help not only as a path to generating the combined methods (a first version of our Java class file assembler compiles the generated source and then assembles the result), but can assist in documenting and debugging their behavior.

7 Evaluation

This section partially evaluates CAT by assessing its use to address some interesting points in the AOSD space. The points were chosen to be representative, not comprehensive. They include important features of four common, but rather distinct, AOSD approaches, Hyper/J [18], AspectJ [20], Composition Filters [5], and role modeling [2]. The evaluation is based on the written descriptions of the approaches cited above. Though they are yet not implemented on CAT, and we would expect to have some issues to resolve in doing so, we believe that the core concepts are correct and appropriate, and that relatively small changes will suffice.

Hyper/J: Hyper/J provides a symmetric, merge-based model of class hierarchy composition, illustrated in Figure 2. We believe that it is straightforward to map all of Hyper/J's composition capabilities to CAT; Table 1 summarizes the realization of some of them.

Table 1: Summary of Mapping of Some Hyper/J Compositions to CAT.

Join Point or Composition	Hyper/J Designator	Realization in CAT
Merge classes, interfaces, methods, fields	Merge, mergeByName, equate	Create new output classes or interfaces and new members in them; forward references from input classes to output class
Override method *a* with *b*	Override	Forward references from *a* in the input to *b* in the output
Bracket (weave before and/or after a method) *a* with *b*	Bracket	Use method combination graph for *a* in the output that invokes *a* from the input and *b* from the input
Partly constrain method invocation order	Constrain order	Use appropriate method combination graph edges
Put a summary function, *s*, onto method *a*, which is composed of *b* and *c*	Set summary function	Use appropriate method combination graph and accumulator variable.

AspectJ: AspectJ provides an asymmetric model in which aspect code is invoked at join points characterized in terms of programming-language execution events. In a CAT realization, instances of aspects would be inserted at the appropriate locations within classes, and code that delegates to those instances would be woven into the appropriate join points within the classes that the aspects crosscut (see Table 2 for details). AspectJ also provides a *cflow* designator, enabling dynamic aspect selection based on a characterization of the system state in terms of its current call flow. Because the instrumentation and characterization of system state required for many selections depends entirely on the anticipated uses, we do not believe it is appropriate to include it directly in CAT. Instead, we envision the use of CAT both to insert instrumentation recording interesting state characterizations and to use it in method combination graphs to identify "dynamic" join points. Higher-level tools or compo-

nents, possibly part of a CAT-based AspectJ implementation, would generate the CAT directives and the code (e.g., side stack implementations) needed.

Table 2 summarizes the realization of some of AspectJ's capabilities in CAT; it does not examine all, but we believe they can all be realized in terms of the CAT abstractions.

Table 2: Summary of Mapping of Some AspectJ Compositions to CAT.

Join Point or Composition	AspectJ Designator	Realization in CAT
Introduce	Variable or method declaration	Create new member in named output class
Weave advice before and/or after a join point, *j*	before, after	Create method combination graph for *j* that invokes *j* and the advice. If *j* is an intra-method join point (e.g., calls, sets, throws), define it as a methoid.
Put advice *a* around a join point, *j*	around, proceed	Forward references from *j* in the input to *a* in the output, and use the *proceed object* (see Section 4). If *j* is an intra-method join point (e.g., calls, sets, throws), define it as a methoid.

Composition Filters: Composition Filters (CF) is essentially a delegation-based approach, in which compositions occur by aggregating composed objects into a composite object, and defining different types of *filters* that control invocation of, and return from, the composed objects' methods. CFs can be used to express multiple inheritance, dynamic inheritance, delegation, and crosscutting behaviors. Some commonly used types of filters and their CAT realizations are described in Table 3.

The CF model is, mostly, straightforward to implement using CAT, using method combination graphs to compose filter. It explicitly prohibits composition below the method level, to prevent encapsulation violations on the existing objects, so methoids are not required. The CF *superimposition* capability [3], which permits the imposition of a given filterinterface on multiple concerns to realize crosscutting, requires no additional capabilities at the CAT level, but rather the generation of the appropriate CAT directives for each affected class. The *wait* filter poses problems, however, because its implicit looping behavior is at odds with the acyclic nature of method combination graphs. We believe that it can be implemented by means of generated methods to perform waiting and queuing and callable from graphs.

Table 3: Summary of Mapping of Some Composition Filters Compositions to CAT.

Join Point or Composition	Composition Filters Designator	Realization in CAT
Combine classes	Concern definition	Create new output class/interface/methods with the appropriate fields for the composed classes; delegate references as prescribed by filters
Error filter for one or more methods *a, b, …*, based on a condition *c*	Error	Create method combination graphs for *a, b, …*, in which a test for *c* occurs before the call to *a, b*, etc.; if *c* is not true, report an error.
System state dependent filter for *a*	Meta	Instrument the software to create and maintain the required characterization of the system state. Use the this information as prescribed by the filter in the method combination graph.
Dispatch filter	Dispatch	Delegate to specified composed object's methods

Roles: Many different approaches to the modeling and implementation of roles exist. One representative model [2] introduces a *role table* instance variable into each class, to associate role names with role instances. Methods to add and remove roles to/from an instance of the class, and to retrieve a given role object from the instance, are also defined. Through its ability to introduce members, CAT supports this implementation of roles, as well as many others. The incorporation of role tables is a general mechanism for supporting some important forms of dynamic and instance-based aspect composition; indeed, some such mechanism is required in the absence of language runtime support for dynamic composition.

8 Related Work

The development of a low-level concern assembly language was motivated in part by our desire to provide behavioral guarantees for composed class hierarchies. Composition and weaving are powerful techniques that enable one to modify existing systems by inserting or modifying code fragments in arbitrary user-specified locations. In modern object-oriented programs that rely heavily on late binding, the impact of such modifications can be highly nonlocal (e.g., the insertion of an assignment in one method may impact the points-to relations and hence method dispatch behavior in another method). Our low-level language was carefully designed to enable the construction of program-

analysis-based tools that can report possible behavioral interference. Some initial work in this area was recently reported [19].

There are three primary categories of related work: languages and tools for composition/weaving, toolkits for manipulating software representations, and program transformation systems.

There are now several composition/weaving approaches, including Adaptive Plug-and-Play Components [17], AspectJ [12], Composition Filters [1], Hyper/J [18], and LAC [10]. These all provide support for AOSD for direct use by humans, at a considerably higher level than CAT. We see CAT as a suitable common layer for implementing the back ends of such approaches, and experimenting with variations, new approaches, and interoperation of approaches.

For most common software representations, there are toolkits or libraries for manipulating them. For example, Eclipse provides interfaces for manipulating Java source code, there are several toolkits for manipulating Java class files (e.g., BCEL[3] and JikesBT[12]). These provide support for general-purpose manipulation of software artifacts, and are therefore an excellent basis upon which to built concern assemblers. They do not, however, provide a common abstraction layer that is neutral to artifact representations, nor do they contain abstractions and functionality specifically suited to concern assembly, such as multiple type spaces, method combination graphs and methoids.

There are several tools for transformation of Java classes, usually at load time, such as Javassist [6], JMangler [5], JOIE [6] and Binary Component Adaptation (BCA) [12]. These are higher-level than the toolkits discussed above, because they provide abstractions that support program transformation, either as a language (e.g., BCA) or interfaces to be implemented or used by transformation components written in Java (e.g., JMangler and JOIE). However, these abstractions are designed to support transformation rather than assembly, and the systems are all specific to Java bytecodes and so do not provide a language-neutral abstraction layer. A recent mailing list posting [11] stated that the next release of AspectJ will contain a library that supports bytecode weaving, built upon BCEL. At time of writing, details are not available, so we cannot compare the abstractions provided, but the fact that it is targeted at Java and built on BCEL suggests that it will also not provide the language- and artifact-neutrality of CAT. We expect that all the transformation tools described above would be excellent bases upon which to build CAT bytecode concern assemblers, and that the required façade would be thinner than in the case of the more primitive toolkits described earlier.

9 Future Work

The Concern Assembly Toolkit is the first step towards our vision of a *Concern Manipulation Environment*: a set of common, reusable components upon which are built a suite of tools that bring the advantages of aspect-oriented software development to a variety of software engineering activities, involving a variety of artifacts at various stages of the software lifecycle. We intend it to be able to support both our and others' approaches to AOSD, and to be a suitable testbed for us and others to experiment with new approaches and to explore the challenging research problems that remain, including behavioral interference.

10 References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition-filters. In Object-Based Distributed Processing, LNCS 791, pages 152{184, 1993

[2] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf, "Role Object." In *Pattern Languages of Program Design 4*. Edited by Neil Harrison, Brian Foote, and Hans Rohnert. Addison-Wesley, 2000.

[3] BCEL web site, http://bcel.sourceforge.net/

[4] Lodewijk Bergmans and Mehmet Aksit, "Composing Crosscutting Concerns Using Composition Filters." CACM 44(10), October 2001, pages 51–57.

[5] Lodewijk Bergmans and Mehmet Aksit, "Composing Multiple Concerns Using Composition Filters." More detailed version of [3] available at http://trese.cs.utwente.nl/publications/papers/CF_superimposition_bergmans_aksit.pdf.

[6] Shigeru Chiba, "Load-time Structral Reflection in Java", In Proceedings of 2000 European Conference on Object Oriented Programming, LNCS 1850, Springer Verlag, 2000

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

[8] Martin Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] W. Harrison, C. Barton, M. Raghavachari, "Mapping UML Designs to Java", In Proceedings of 2000 Conference on Object-Oriented Programming Systems, Languages, and Applications, Minneapolis, 2000

[10] S. Herrmann, M. Mezini, Combining Composition Styles in the Evolvable Language LAC, Workshop on Advanced Separation of Concerns at International Conference on Software Engineering, 2001.

[11] J. Hugunin, "AspectJ-1.1 implementation status report." Posting to the users@aspectj.org mailing list, September 23, 2002.

[12] JikesBT web site, http://www.alphaworks.ibm.com/tech/jikesbt

[13] R. Keller, U. Hölzle, "Binary Component Adaptation," In Proceedings of 1998 European Conference on Object Oriented Programming, LNCS 1445, Springer Verlag, 1998.

[14] G. Kiczales, E.Hilsdale, J. Hugunin, Mik Kersten, J. Palm, W. Griswold, "An Overview of AspectJ." In Proceedings ECOOP'01, Springer-Verlag, June 2001.

[15] D. Kimelman, V. Kruskal, H. Ossher, T. Roth and P. Tarr, "HyperProbe: An aspect-oriented instrumentation tool for troubleshooting large-scale production systems." Demonstration at the First International Conference on Aspect-Oriented Software Development (AOSD 2002), Enschede, The Netherlands, April 2002. http://trese.cs.utwente.nl/aosd2002/index.php?content=hyperprobe.

[16] 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), collocated with International Conference on Software Maintenance (ICSM)

[17] M. Mezini, K. Lieberherr, "Adaptive plug-and-play components for evolutionary software development", In Proceedings of 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications, Vancouver, 1998

[18] H. Ossher and P. Tarr. "Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software." CACM 44(10): 43–50, October 2001.

[19] G. Snelting and F. Tip, Semantics-based composition of class hierarchies, In Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002), (Malaga, Spain, June 10-14, 2002), pp. 562-584.

[20] The AspectJ Team, "The AspectJ Programming Guide." http://aspectj.org/doc/dist/prog-guide/index.html.

[21] Frank Tip, "A survey of program slicing techniques," Journal of Programming Languages 3(3), (1995), 121-189.

[22] J. Warmer, A. Kleppe, The Object Constraint Language: Precise Modeling With UML , Addison-Wesley, 1998

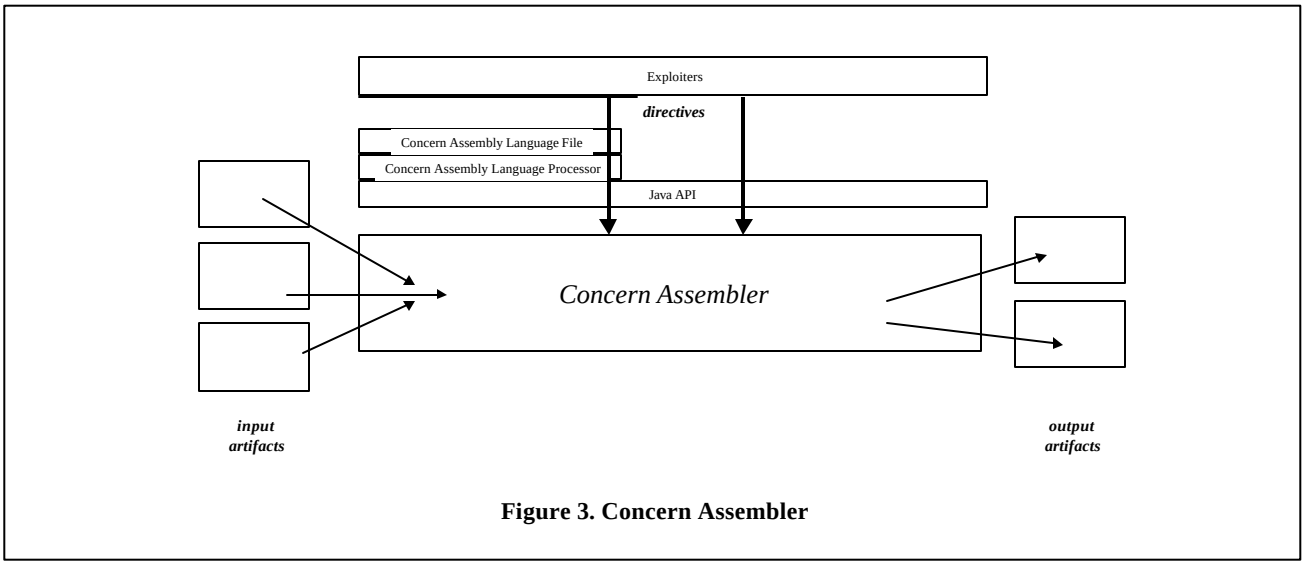

Figure 3. Concern Assembler

```
<types> <!-- List of classes (and interfaces, if any) involved in the assembly -->
  <common classpath="/jre/lib/rt.jar"/>
  <inputs>
    <input name="DOC" classpath="doc"> <!-- Input space listing all input classes used -->
      <type name="ClaimDoc">
        <methoid name="setAddress" attributes="package" tied="no">
          <methoidcharacterization kind="set">
            <characterizationproperty name="name" value="address"/>
            <characterizationproperty name="type" value="java.lang.String"/>
          </methoidcharacterization>
        </methoid>
      </type>
    </input>
    <input name="PROTECT" classpath="protect"> <type name="ProtectedClaimDoc"/> </input>
  </inputs>

  <outputs>
    <output name="OUTPUT" directory="result"> <!-- Output space and classes -->
      <type name="ClaimDoc" attributes="public"/>
    </output>
  </outputs>

  <mapping> <!-- How input classes are mapped to outputs when references are copied -->
    <type> <from name="DOC:ClaimDoc"/> <to name="OUTPUT:ClaimDoc"/></type>
    <type> <from name="PROTECT:ProtectedClaimDoc"/> <to name="OUTPUT:ClaimDoc"/></type>
  </mapping>
</types>

<members> <!-- Details of the members of all assembled classes -->
  <composition> <!-- How composed members are to be constructed -->
    <!-- Copy the name field from input to output -->
    <field within="OUTPUT:ClaimDoc" name="name" type="…String" attributes="private">
      <from within="DOC:ClaimDoc" name="name" type="java.lang.String"/>
    </field>

    <!-- Copy the input "approveClaim", renaming it "originalApproveClaim" -->
    <method within="OUTPUT:ClaimDoc" name="originalApproveClaim">
      <from within="DOC:ClaimDoc" name="approveClaim" types="()" returns="void"/>
    </method>
            <!-- Other copies from the document concern with or without renaming -->
    <!-- Copy input methods from protection concern, with their original names -->
    <method within="OUTPUT:ClaimDoc" name="isClaimAssessor">
      <from within="PROTECT:ProtectedClaimDoc" name="isClaimAssessor" types="()"
                                                                    returns="boolean"/>
    </method>
            <!-- Other copies from the protection concern without renaming -->
    <!-- Create an output method "approveClaim" with body from the graph in Fig. 2 -->
    <method within="OUTPUT:ClaimDoc" name="approveClaim" types="()" returns="void">
      <graph> <!-- See Figure 5; XML form too long to show here --> </graph>
    </method>
            <!-- Create other output methods whose bodies are specified by graphs -->
  </composition>

  <mapping> <!-- How input members are mapped to output when references are copied -->
    <submapping name="OUTPUT:ClaimDoc"> <!-- Member mappings are in context of class -->
      <method>
        <from within="DOC:ClaimDoc" name="approveClaim" types="()" returns="void"/>
        <to within="OUTPUT:ClaimDoc" name="approveClaim" types="()" returns="void"/>
      </method>
            <!-- likewise for the other methods  -->
      <!-- map references to the address field if any were excluded from methoid creation -->
      <field>
        <from within="DOC:ClaimDoc" name="address" type="java.lang.String"/>
        <to within="OUTPUT:ClaimDoc" name="address" type="java.lang.String"/>
      </field>
    </submapping>
  </mapping>
</members>

<relationships> <!-- Extends relationships between types (also implements & throws) -->
  <extends> <type name="OUTPUT:ClaimDoc"/> <type name="java.lang.Object"/></extends>
</relationships>
```

Figure 4: Example Assembly Directives

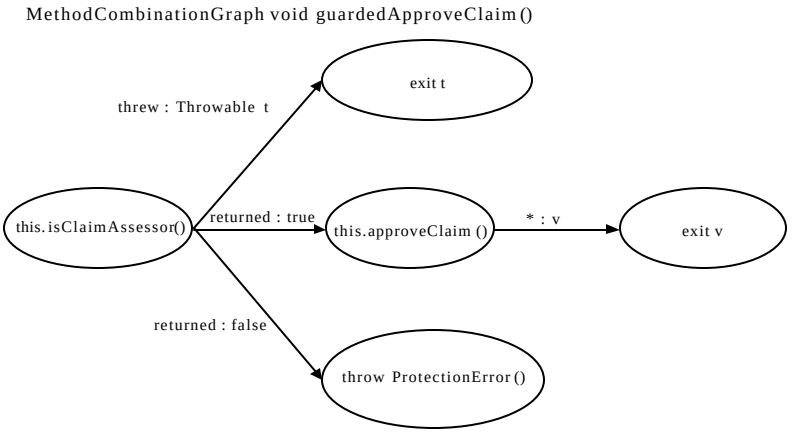

Figure 5. Example Method Combination Graph

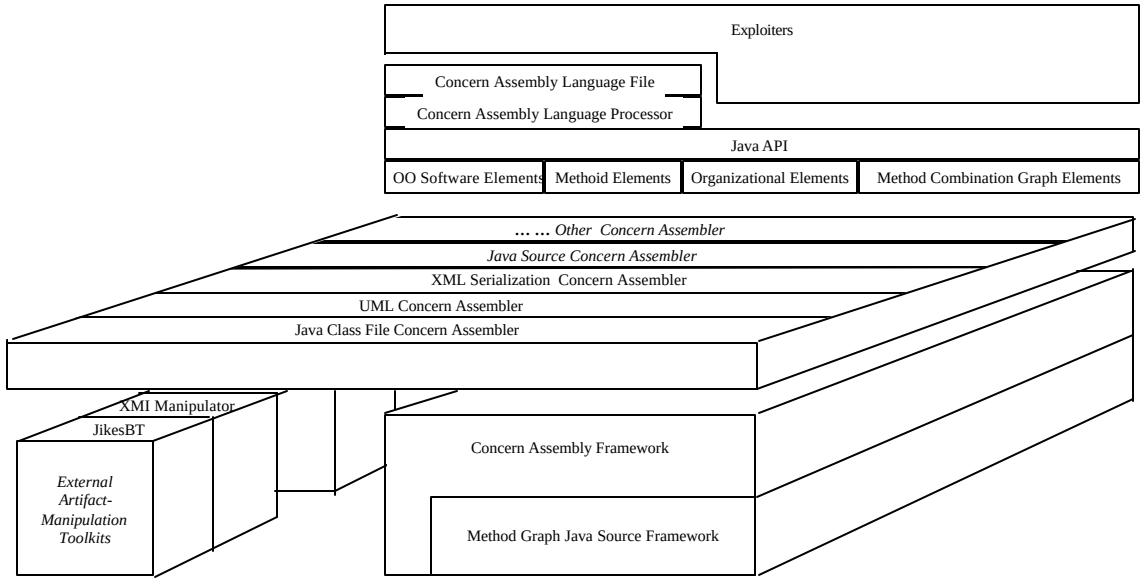

Figure 6. The Concern Assembly Toolkit