RC22612 (W0210-192) October 29, 2002
Computer Science

IBM Research Report

Common Low-Level Support for Composition and Weaving

William H. Harrison, Vincent Kruskal, Harold L. Ossher,
Peri L. Tarr, 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

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. , payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Common Low-Level Support for Composition and Weaving

William Harrison, Vincent Kruskal, Harold Ossher, Peri Tarr and Frank Tip
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
{harrisn, kruskal, ossher, tarr, tip}@watson.ibm.com

1. Introduction

The modularization and composition of concerns that are characteristic of aspect-oriented software development can be specified and implemented in various ways, and can be applied to various representations of software. Yet, despite these differences, there are important similarities. This position paper describes a *composition toolkit* that is intended to capture these similarities, providing a common back end for AOSD tools that involve composition or weaving. We are currently building the next generation of Hyper/J, and related tools, on this support. We believe that it will also be suitable as a basis for tools supporting other AOSD approaches. The toolkit is intended: to supply a base of primitive operations satisfying three goals:

- they provide a common base for weaving that suits a variety of purposes and of AOSD approaches,
- they enable support for applying weaving to different software representations and artifacts, and
- they are capable of representing the weaving required by a variety implementation strategies for AOSD.

For a variety of approaches: Many approaches describe composition or weaving using high-level linguistic constructs that involve pattern matching and sets. For example, Hyper/J [2] uses composition relationships such as mergeByName, which involves name and type matching across entire class hierarchies, and AspectJ [1] uses pointcut designators, which involve patterns that can match constructs across entire systems. Each such high-level specification boils down to (usually many) specific, low-level compositions. Tools that implement these approaches must, in effect, expand the high-level specification into a set of lower-level ones, and then apply those. So the first goal of our toolkit is to provide a common set of low-level composition primitives that are suitable for realizing a wide variety of high-level approaches. They are provided both as Java interfaces and 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 [3]. Making these low-level primitives 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 weaving fragments expressed in some existing language, for example to combine classes, or to insert aspect code or calls to aspect code into base code. This can be done as a separate tool, as a preprocessor, or deep within a compiler, but in the end, some form of weaving is taking place. Even in systems that support runtime manipulation of aspects in standard runtime systems, some form of weaving is needed to prepare objects for dynamic aspect manipulation (e.g., addition of aspect pointers or facet tables). Weaving 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, in the end, performing the detailed manipulations to achieve composition is representation specific. This second goal of our composition toolkit is then to provide a common interface, the low-level composition primitives, which can be implemented by pluggable *composition engines* for a variety of representations, allowing the tools that build upon it to ignore representation details. In developing the primitives, we have been working both

with Java and UML class diagrams. We expect that the primitives will also be applicable to other object-oriented languages, such as C# and Smalltalk.

For a variety of implementations: As suggested above, even ignoring software representation, there are various ways of weaving code to achieve a desired result. For example, two classes can be composed to produce a single class that combines their members (analogous to inheritance), or to produce a new class whose objects contain pointers to instances of each of the original classes and which delegate to them. As another 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 program representation is onerous. The third goal of our composition toolkit, then, is that such different approaches be convenient to specify in terms of the low-level primitives.

We currently have composition engines for Java class files and UML class diagrams represented as XMI files. We also have a "serializer" composition engine that produces the XML form of the composition primitives, and an XML reader that consumes it. This is just a start. A final goal of the composition toolkit is that it be an open framework, permitting new and different composition engines 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, [4,6]). For example, our composition engine for Java class files is built on the JikesBT bytecode manipulation toolkit [5].

2. The Composition Model, Primitives, and Protocol

The composition toolkit augments the basic OO model of field- and method-containing classes with several additional concepts.

2.1. Universes

Composition takes place in a "universe", managed by a single composition engine. Universes contain a collection of named ***input type spaces*** and ***output type spaces***. Input type spaces are bound to some existing source of definitions for the material to be composed, such as a Java classpath or a UML/XMI diagram. Output type spaces are generally bound to an empty space into which the composite result will be put.

In response to calls on composition primitives, the composition engine creates and populates the type spaces to be manipulated in subsequent operations.

2.2. Type Spaces

Each type space is a closed[1] collection of named types: classes, interfaces, and primitives. The elements in type space are treated as though stored in a "flat" name space, although the names of the elements reflect Java's package naming conventions and some composition engines store types in a nested fashion.

There is a composition primitive to create types in an output type space. These types are given attributes, like "public", "interface", "abstract", etc., but are created without any contained members.

2.3. 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 attributes and a body. In Java, field bodies are executed when the field is initialized, and method bodies are executed when the method is invoked. In UML, the bodies may specify OCL constraints.

[1] Interpreting a reference not defined within the space is invalid.

There are two kinds of composition primitives for creating members: creation-from-scratch, and creation-by-copy. With creation-from-scratch, the new member is completely specified by the name, characterization, attributes, and other information passed to the construction primitive. With creation-by-copy, this new member specification is augmented with a reference to a similar member in an input space, from which additional attribute and body information is drawn. There are also primitives for exposing and composing join points within method code, but these will be the subject of a future paper.

2.4. Mappings

Declarations and bodies of class members, fields, and methods, 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 types is specified in a mapping, a function associated with the output space in which the output type exists. In some cases, translation from an input type or member referenced in the body of a member to its corresponding output type or member may vary depending on the output type into which the member is copied. To accommodate this need, a local mapping is also associated with each output type.

Composition primitives are employed to add to the mappings the individual translations of input types and members to their corresponding output types and members.

2.5. Relationships

Relationships, such as extends and implements between types and throws between methods and types are specified separately, rather than as part of the definitions of the related entities.

2.6. Protocol

It is important that the construction be clear, unambiguous and well-formed. While it is, strictly speaking, only necessary to assure 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 primitives are therefore constrained to be used in the following sequence:

1. Type-creation primitives
2. Mapping-construction primitives for type translations
3. Member-construction primitives
4. Mapping-construction primitives for member translations
5. Relationships creation primitives
6. Copying and translation of member bodies.

3. Method Combination Graphs

Creation-from-scratch of new methods can either create an abstract method, which has no body, or a method whose body specifies how other method bodies are combined. The specification is called a ***method combination graph***. A method combination graph has:

- a set of declarations for method combination graph variables,
- a set of unique nodes, each of which has an operation (a method call or an exit). Some nodes (e.g. return, throw) are identified as ***exit*** nodes, and
- a set of edges controlling the order and circumstances under which the nodes are interpreted. Each edge connects a predecessor node with a successor node and identifies a condition governing the connection. The presence of an edge has two effects: it indicates that the successor can not be executed while its predecessor is yet to be executed and it indicates that, if the condition is true after executing the predecessor, then the successor should eventually be executed. The edge may also indicate a method combination graph variable in which the result of the execution of the predecessor node's method is recorded.

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.

4. Example

As a concrete illustration of the composition primitives, consider a tiny composition of two classes, `Driver` and `Trace`. `Driver` has a constructor and an `f` method. `Trace` has `before`, `after` and `afterThrow` methods (and a constructor, which is ignored). The intent of the composition is to wrap `Driver.f`, calling `Trace.before` at the start of its execution, an either `Trace.after` or `Trace.afterThrow`, as appropriate, afterwards. Any exceptions thrown by the tracing methods are caught and ignored. Though tiny, this example is representative of real tracing or logging applications; many more methods and classes would be involved, but the details would be repetitive.

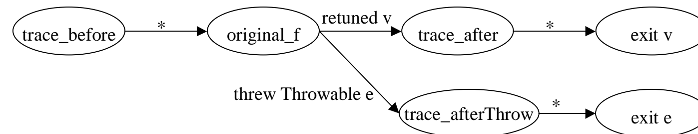

Figure 1. Example Method Combination Graph

Figure 1 shows the method combination graph describing the combination of `f` and the methods that trace its execution. The method names have been changed to have prefixes that indicate their origins. The appendix contains an XML file specifying the entire composition, including these naming issues. Comments in the XML explain some of the details.

References

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

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

[3] 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.

[4] BCEL,http://bcel.sourceforge.net/

[5] JikesBT, http://www.alphaworks.ibm.com/tech/jikesbt

[6] JMangler, http://javalab.cs.uni-bonn.de/research/jmangler/

Appendix

```
<JikesCT version="2.0">
  <types> <!-- List of all classes (and interfaces, none in this e.g.) involved in the composition -->
    <inputs>
      <input name="INPUT" classpath="."> <!-- Input space listing all input classes used -->
        <type name="Eg.Driver"/>
        <type name="Eg.Trace"/>
      </input>
    </inputs>

    <outputs>
      <!-- Output space, listing all composed classes to be produced, and where to put them -->
      <output name="OUTPUT" directory="zresults/Eg">
        <type name="Eg.Driver" attributes="public"/>
      </output>
    </outputs>
```

```
  <mapping> <!-- How input classes are to be mapped to output classes when references are copied -->
    <type> <from name="INPUT:Eg.Driver"/> <to name="OUTPUT:Eg.Driver"/></type>
    <submapping name="OUTPUT:Eg.Driver">
      <!-- The mapping in this submapping applies only in the context of producing Eg.Driver.
           If tracing were applied to multiple classes, INPUT:Eg.Trace would be mapped to each
             of them, in separate submappings. -->
      <type> <from name="INPUT:Eg.Trace"/> <to name="OUTPUT:Eg.Driver"/></type>
    </submapping>
  </mapping>
</types>

<members> <!-- Details of the members of all composed classes -->
  <composition> <!-- How composed members are to be constructed -->
    <!-- Copy the constructor from input to output -->
    <method within="OUTPUT:Eg.Driver" name="Driver">
      <from within="INPUT:Eg.Driver" name="Driver" types="()"/>
    </method>

    <!-- Copy the input "f" method to the output, renamed as "original_f" -->
    <method within="OUTPUT:Eg.Driver" name="original_f">
      <from within="INPUT:Eg.Driver" name="f" types="(java.lang.String)" returns="void"/>
    </method>

    <!-- Other copies with renaming -->
    <method within="OUTPUT:Eg.Driver" name="trace_before">
      <from within="INPUT:Eg.Trace" name="before" types="(java.lang.Object)" returns="void"/>
    </method>
    <method within="OUTPUT:Eg.Driver" name="trace_after">
      <from within="INPUT:Eg.Trace" name="after" types="(java.lang.Object)" returns="void"/>
    </method>
    <method within="OUTPUT:Eg.Driver" name="trace_afterThrow">
      <from within="INPUT:Eg.Trace" name="afterThrow"
             types="(java.lang.Object, java.lang.Throwable)" returns="void"/>
    </method>

    <!-- Creat an output method "f" whose body is specified by the graph in Fig. 1 -->
    <method within="OUTPUT:Eg.Driver" name="f" types="(java.lang.String)" returns="void">
      <graph>
        <variable name="v" type="void" />
        <variable name="e" type="void" />
        <node name="before" start="yes" kind="call" target="this" arguments="($1)">
           <method name="trace_before" within="OUTPUT:Eg.Driver"
                   types="(java.lang.Object)" returns="void"/>
        </node>
        <node name="method" kind="call" target="this" arguments="($1)">
           <method name="original_f" within="OUTPUT:Eg.Driver"
                   types="(java.lang.String)" returns="void"/>
        </node>
        <node name="after"  kind="call" target="this" arguments="($1)">
           <method name="trace_after" within="OUTPUT:Eg.Driver"
                   types="(java.lang.Object)" returns="void"/>
        </node>
        <node name="afterT"  kind="call" target="this" arguments="($1, e)">
           <method name="trace_afterThrow" within="OUTPUT:Eg.Driver"
                   types="(java.lang.Object, java.lang.Throwable)" returns="void"/>
        </node>
        <node name="normalExit" kind="exit" result="v" />
        <node name="throwExit"  kind="exit" result="e" />
        <edge from="before" to="method"     condition="*" />
        <edge from="method" to="after"      condition="returned" variable="v" />
        <edge from="method" to="afterT"     condition="threw"
                                            exceptiontype="java.lang.Throwable" variable="e" />
        <edge from="after"  to="normalExit" condition="*" />
        <edge from="afterT" to="throwExit"  condition="*" />
      </graph>
    </method>
  </composition>
```

```
    <mapping> <!-- How input members are mapped to composed members when references are copied -->
      <submapping name="OUTPUT:Eg.Driver"> <!-- Member mappings are in the context of the class -->
        <method>
          <from within="INPUT:Eg.Driver" name="Driver" types="()"/>
          <to within="OUTPUT:Eg.Driver" name="Driver" types="()"/>
        </method>
        <method>
          <!-- A call to "f" copied from the input should become a call to "OUTPUT:Eg.Driver.f" in
               the output, the method built from the graph in Fig. 1, implementing composed behavior.
               This method will, in turn, call "original_f," as well as the tracing methods -->
          <from within="INPUT:Eg.Driver" name="f" types="(java.lang.String)" returns="void"/>
          <to within="OUTPUT:Eg.Driver" name="f" types="(java.lang.String)" returns="void"/>
        </method>
      </submapping>
    </mapping>
  </members>

  <relationships> <!-- Extends relationships between types (also implements and throws, if any) -->
    <extends> <type name="OUTPUT:Eg.Driver"/> <type name="java.lang.Object"/></extends>
  </relationships>
</JikesCT>
```