Programming Connectors In an Open Language

Uwe Aßmann, Andreas Ludwig, Daniel Pfeifer
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
Postfach 6980, Zirkel 2, 76128 Karlsruhe, Germany
fax:+49/721/30047, tel:+49/721/608-4763

Abstract
Connectors can be programmed flexibly using an open language with a static meta-object protocol. Illustrated with an example from OpenJava, it is presented how such connectors insert communication code into classes transparently. With this method, connectors become meta-programs in the open language; connecting becomes a program transformation. The method paves the way for connector libraries which are easy to extend.

Category: Technology support, experience paper
Keywords: Software architecture, connectors, meta-programming, program transformations, open language

1 INTRODUCTION

In software construction, it has become popular to separate architectural from application-specific aspects. Software engineers hope that both aspects can be exchanged independently of each other, improving reuse of components in different architectures and reuse of architecture with different components.

For this separation of aspects, *architectural description languages (ADL)* have been developed. Systems based on such languages provide tailored programming environments in which architectures are specified as a hierarchy of *components* [MDK92] [SDK+95] [GAO95] [LKA+95]. Components are sets of classes or modules and provide abstract interfaces expressed in *ports*. Ports describe gates in and out of components through which data items flow. *Connectors* link the ports of different components together describing the communication. From those connector specifications, code can be generated so that applications consist of hand-written component code and generated architectural code. However, often such systems only support fixed communication styles with a restricted set of connectors. Only recently, this begins to change [SDZ96][AG97].

[DR97] extends this approach to *user-programmable connectors*. In his connector language FLO, connectors can be described in a lisp-like syntax enriched with some special programming constructs. From those specifications, the connector compiler can generate connection code. While this approach works well a user has to learn a new programming language for connectors. Instead it would be better to follow a library-based approach: if connectors could were operators in a library for a standard programming language users would quickly understand how to write connectors. Since libraries are extensible per se it would be very easy to add new connectors to the system.

In this paper, we demonstrate how to program such a connector library with an open language that provides a meta-object protocol. In such a language, connectors become simple meta-programs extending classes and methods with architectural code [Aßm98]. As an example, an event-based connector is presented, an *observifier*, which substitutes a procedure call with an event-based communication, changing its connected classes transparently. The connector is implemented with OpenJava [Tat98] and generates ordinary Java code (*static meta-programming*). In this setting, components correspond to sets of Java classes. Ports correspond to ordinary Java method calls, and connectors are meta-operators that replace calls by other communication mechanisms. Hence connector applications are program transformations, generating new versions of the software and specializing the components to specific communication styles.

Before we present the example connector *observifier*, we introduce the open language OpenJava.

2 THE OPEN LANGUAGE OPENJAVA

The OpenJava system is a Java preprocessor and compiler supporting a meta-programming library. Calls to this library are resolved statically. In essence the library offers the abstract syntax tree to the user, and the meta-programs transform the abstract syntax tree of the program. Originally, OpenJava has been designed to develop Java language extensions, and the distribution contains extension examples, e.g. how to extend Java with templates. Language extension in OpenJava works as follows:

- OpenJava allows the user to add new keywords to the language, i.e. to make small extensions to the grammar of the parser (e.g. the keyword *template*).
- If the OpenJava compiler finds these keywords in a program, it calls a meta-program handles the keyword. This meta-program overrides a method of a standard OpenJava compiler class and transforms one or more nodes in the abstract syntax tree to introduce the semantics of the keyword into the program. This process is repeated until the whole program has been investigated.
- Finally, the OpenJava compiler pretty-prints the abstract syntax tree to an ordinary Java file or generates byte code.

In our case, the language extension facilities of OpenJava are not used. Instead we will only exploit the meta-programming interface for implementing the meta-programming connectors.

3 APPLYING A META-PROGRAMMING CONNECTOR

In our approach, connectors are program transformers which are implemented as meta-programming methods. Connector applications are program transformations, generating new versions of the software and specializing the components. This yields a system construction method in which connectors are composition operators composing the final system from the components. In the following, we give an overview how a meta-programming connector works.

Initialization phase First, all classes which should be connected must be determined (*parameter classes*). Either they are looked up in an existing abstract syntax tree or read from file.

Selection phase Then the places have to be selected where the connector connects classes, i.e. ports have to be identified. Since in our case ports are calls, calls in Java methods have to be identified which should be substituted. The calls can be selected manually by specifying abstract syntax tree nodes or by an interactive wizard using a class browser.

Transformation phase This phase applies the selected connectors, transforming the classes and resulting in a modified abstract syntax tree. This is repeated until all connections are performed.

Emitting phase Finally, the abstract syntax tree is traversed by the standard OpenJava pretty printer to emit the final Java source code. Since the modified classes are derived from their originals, they have to be put under version control.

3.1 The example: Towers of Hanoi

The following presents a simple example, attaching an event-based connection to a procedure call port. The program solves Towers of Hanoi, recursively calling itself for solutions of Towers of Hanoi with smaller problem size. The program prints a message about each move with the method `display`. In terms of software architecture, `Hanoi.compute` can be considered as a component which is connected to `Hanoi.display` by a procedure call connector (Figure 1). Our example shows how the connector *observifier* replaces this connection by an event-based connection.

```
/**
 * Towers of Hanoi; File with procedure call as connector
 */
import java.io.*;
```

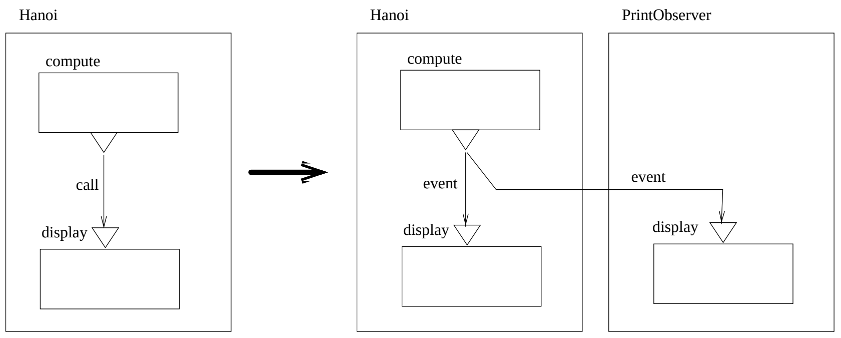

Figure 1 Exchanging the method call to an event-based communication. A second observer listens to the events.

```
public class Hanoi {
    public Hanoi() {}
    protected void compute(int n, String s, String t, String h) {
        if (n > 1)
            compute(n - 1, s, h, t);
        display(s, t);
        if (n > 1)
            compute(n - 1, h, t, s);
    }
    public void display(String s, String t) {
        System.out.println(s + " --> " + t);
    }
    public static void main(String[] a) {
        Hanoi h = new Hanoi();
        h.compute(n, "Source", "Target", "Help");
    }
}
```

After applying the connector *observifier*, the program contains the following modifications (marked by `>` in the next listing).

- In procedure `compute`, the call to `display` is exchanged to a notifications of all currently listening observers.
- These observers are attached to the `Hanoi` class in the constructor of the `Hanoi` class; one of them is `Hanoi` itself, another one is an object `PrintObserver` that prints some messages.
- `compute` activates `display` with an event object containing all parameters of the old call. To this end, there is a new procedure `update` which acts as observer: it receives the event, unpacks the parameters from the event object, and calls `display` via a procedure call.

```
/*
 * This code was generated by OpenJava System.
 */
import java.io.*;

```

```
public class Hanoi extends java.util.Observable implements java.util.Observer
{
  public Hanoi() {
>   addObserver( new PrintObserver() );
>   addObserver( this );
  }
  protected void compute( int n, String s, String t, String h ) {
    if(n > 1){
      compute( n - 1, s, h, t );
    }
>   setChanged();
>   notifyObservers( new displayPack( s, t ) );
    if(n > 1){
      compute( n - 1, h, t, s );
    }
  }
  public void display( String s, String t ) {
    System.out.println( s + " --> " + t );
  }
  public static void main( String[] a ) {
    Hanoi h = new Hanoi();
    h.compute( n, "Source", "Target", "Help" );
  }
> public void update( java.util.Observable o, java.lang.Object arg ) {
>   this.display( ((displayPack) arg).s, ((displayPack) arg).t );
>   return;
> }
}
```

The output of the modified program is similar to the output of the original program. In addition, the listener procedure from `PrintObserver` prints a message for each move.

On the first sight, it seems awkward to substitute a call in a sequential program with an event-based communication. However, event communication allows that multiple observer procedures can be attached dynamically to the call, performing additional actions. For instance, a visualization algorithm may be attached that starts an incremental display of a data structure. Since this is transparent to the code, systems can be extended and adapted flexibly.

3.2 How the observification works

Our implementation of the observifier is a method which replaces a call to a event-signaling call sequence. Its interface is as follows. It can be called as standard procedure from a main program in order to perform the connection on arbitrary classes.

```
package compost.connectors;
import openjava.ptree.*;
import openjava.util.*;
import openjava.*;

public static ClassDeclaration observify(
   ClassDeclaration subject,         /* subject of communication */
   StatementList statementList,      /* subject's statement list */
   int position,                     /* the place in statement list to mix-in call */
   ClassDeclaration observer)        /* object of communication */
   throws openjava.ptree.PtreeException
```

On top of the meta-model and meta-object protocol of OpenJava, the implementation of the observifier is straightforward. It contains the following steps, namely deleting, allocating, and modifying meta-objects of type `ClassDeclaration`, `MethodDeclaration`, and `Statement`.

- The interface of `Hanoi` is extended to implements the interface `Observable` as well as the interface `Observer` from JDK-1.2.
- The call is removed from its including statement list.
- A call to a `setChanged` is inserted, signaling that an event has occured in the subject.
- Since event communication in Java transfers single objects, the parameters of a call to method `display` have to be tupled into an event object. Hence another meta-operator *packify* is assumed which creates a new class `displayPack` whose objects carry the attributes of such a call.
- A call to `notifyObservers` from `java.util.Observable` is inserted. This call transfers the event object and control to all observing objects. Its parameter is a new object of the pack class.
- The packifier supports another method `unpackify` that creates statements to unpack the arguments from the event object. This operator is used to insert the unpacking of parameters into the update method of the observer.

More information and the listing of the observifier are found in the full paper which can be obtained from the authors.

4 CONCLUSION

This paper presented a method how to program connectors in an open language. Driven by an example, a connector meta-program was explained that introduces event-based observation into a standard object-oriented program. Since connectors are programmed in a standard programming language, new connectors can easily be developed and collected in libraries.

REFERENCES

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, July 1997.

[Aßm98] Uwe Aßmann. Meta-programming composers in second-generation component systems. In J. Bishop and N. Horspool, editors, *Systems Implementation 2000 - Working Conference IFIP WG 2.4*, Berlin, February 1998. Chapman and Hall.

[BW97] Martin Büchi and Wolfgang Weck. A plea for grey-box components. Technical Report TUCS-TR-122, Turku Centre for Computer Science, Finland, September 5, 1997.

[DR97] S. Ducasse and T. Richner. Executable Connectors: Towards Reusable Design Elements. In M. Jazayeri and H. Schauer, editors, *Proc. 6th European Software Eng. Conf. (ESEC 97)*, volume 1301 of *Lecture Notes in Computer Science*, pages 483–499. Springer-Verlag, Berlin, 1997.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, D. Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[MDK92] Jeff Magee, Naranker Dulay, and Jeffrey Kramer. Structuring parallel and distributed programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, London, March 1992.

[SDK+95] Mary Shaw, Robert DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, pages 314–335, April 1995.

[SDZ96] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and implementations for architectural connections. In *3rd International Conference on Configurable Distribute systems.* IEEE Press, May 1996.

[Tat98] Michiaki Tatsubori. OpenJava language manual, version 0.2.3, January 1998. http://www.softlab.is.tsukuba.ac.jp/~mich/openjava/.