The COMPOST, COMPASS, Inject/J and RECODER tool suite for invasive software composition: Invasive composition with COMPASS aspect-oriented connectors

Conference Paper · January 2005

DOI: 10.1007/11877028_14 · Source: DBLP

Uwe Assmann
Technische Universität Dresden
194 PUBLICATIONS **1,235** CITATIONS
SEE PROFILE

Mircea Trifu
Logarix srl
19 PUBLICATIONS **175** CITATIONS
SEE PROFILE

The COMPOST, COMPASS, Inject/J and RECODER Tool Suite for Invasive Software Composition: Invasive Composition with COMPASS Aspect-Oriented Connectors

Dirk Heuzeroth[1] and Uwe Aßmann[2]

[1] `www.dirk-heuzeroth.de`
[2] TU Dresden, Germany

Abstract. Program analyses and transformations are means to support program evolution and bridge architectural mismatches in component composition. The Program Structures Group at the University of Karlsruhe und the FZI Karlsruhe, that we have been members of, have developed a suite of program analysis and transformation tools to attack these problems.
The basic tool Recoder offers sophisticated source code analyses and a library of common transformations in the form of Java meta programs to perform necessary component and interaction adapations. This library can be extended by the Recoder framework that offers means for implementing custom transformations. A transformation can also be a generator to produce glue code, for example.
InjectJ uses Recoder and offers a comfortable scripting language for implementing transformations. The scripting language combines declarative specifications of the program points, where the transformation should be applied, with imperative specifications of the transformation itself.
COMPASS is focused on bridging interaction mismatches among software components. It introduces architectural elements like components, ports and aspect-oriented connectors as source code transformations based on the Recoder framework.
COMPOST defines the general model of invasive software composition, where the ports of the COMPASS model are just one kind of hooks. Hooks are join points, i.e. part of a component that may be extended or replaced.

1 Introduction

Software systems evolve due to changing requirements. This evolution mostly comprises changing the programs' source code, because the new requirements have not been anticipated. Examples are performance tuning and deployment of the system in a new infrastructure requiring other interaction mechanisms. Source code adaptations are also often necessary when composing independently developed components that do not completely fit each other, because of *architectural mismatches* [7], i. e., the components make different assumptions on how to interact with each other.

Indeed, program evolution very often concerns adapting component interactions. In this paper we therefore focus on source code transformations to perform the necessary interaction adaptations automatically. The underlying concept of our program adaptation approach is called *invasive software composition* [2]. It is implemented as the COMPOST (COMPOsition SysTem) framework [3]. Section 3 describes the details. As a basic infrastructure to preform automated program transformations serves our Recoder framework [22, 21]. It offers sophisticated source code analyses, a library of program transformations and means to implement custom program transformations. Section 4 introduces the concepts, design and features of Recoder. Recoder does not provide comfortable means to realize interactive transformations as well as to declaratively specify the program points where to apply transformations. These tasks are facilitated by the scripting language InjectJ [9], introduces in Section 5. All three tools Recoder, InjectJ and COMPOST are quite general program transformation tools none of which is tailored to perform interaction adaptations. This is why we have developed the COMPASS tool [11, 12]. COMPASS offers means to bridge interaction mismatches among software components. It defines architectural elements like components, ports and aspect-oriented connectors as source code transformations based on the Recoder framework and as specialization of the COMPOST framework. Section 6 presents the concepts and features of COMPASS.

Figure 1 illustrates the relations among the four tools Recoder, InjectJ, COMPOST and COMPASS. COMPASS directly builds on Recoder to perform transformations, instead of using InjectJ, since COMPASS needs the comprehensive program model and InjectJ only offers an abstracted program model.

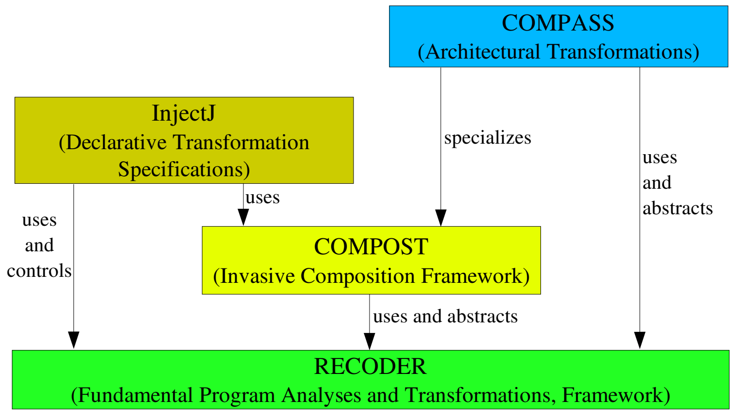
COMPASS
(Architectural Transformations)
InjectJ
(Declarative Transformation Specifications)
specializes
uses and abstracts
uses
uses and controls
COMPOST
(Invasive Composition Framework)
uses and abstracts
RECODER
(Fundamental Program Analyses and Transformations, Framework)

Fig. 1. The relations among the Recoder, InjectJ, COMPOST and COMPASS tools

Before we start explaining the tools, we first introduce a running example (cf. Section 2 that we will use to demonstrate the concepts and features of the program transformation tools.

2 Running Example: Interaction Adaptation

In the remainder of the paper we use the adaptation of the communication in a very simple producer-consumer-system as example. We will transform the producer-consumer-system such that the direct communication of the product from the producer component to the consumer component by a method call is replaced by transferring the product via a buffer component. Such a change may arise from the requirement of increasing the concurrency in the system. Since turning the producer and consumer components into active ones is very easy to achieve in Java just letting both of them inherit from the `Thread` class and adding a `run` method to both of them, we dispense with discussing this step and focus on exchanging the interaction mechanism replacing the direct call by interaction via a buffer.

Given is the following Java source code of a very simple producer-consumer-system:

```
class Trader {
  public static void main(String[] args) {
    Consumer c = new Consumer();
    Producer p = new Producer(c);
    p.produce();
  }
}

class Producer {
  private Consumer consumer;

  public Producer(Consumer c) {
        consumer = c;
  }
  public void produce() {
        Product p = new Product();
        System.out.println(p+" produced.");
        consumer.consume(p);
  }
}
```

```
class Consumer {
  public void consume(Product prod) {
    System.out.println(prod + " consumed.");
  }
}
```

The `Trader` component represents the main program that makes the `Consumer` component instance known to the `Producer` component instance such that the latter can invoke the `consume` service of the `Consumer` component instance. The `Trader` drives the system by calling the `produce` method of the `Producer` object. The communication of the product from the producer to the consumer is initiated by the producer. The producer thus drives the interaction with the consumer.

The target system we aim to achieve as a result is represented by the following source code:

```
class Trader {
  public static void main(String[] args) {
    Buffer b = new Buffer();
    Consumer c = new Consumer();
    Producer p = new Producer();

    p.setProductBuffer(b);
    c.setProductBuffer(b);

    p.produce();
  }
}

class Producer {
  private ProductBuffer productBuffer;

  public Producer() { }

  public void setProductBuffer(
      ProductBuffer productBuffer) {
    this.productBuffer = productBuffer;
  }

  public void produce() {
    Product p = new Product();
    System.out.println(p+"␣produced.");
    productBuffer.put(p);
  }
}
```

```
class Consumer {
  private ProductBuffer productBuffer;

  public Consumer() { }

  public void setProductBuffer(
      ProductBuffer productBuffer) {
    this.productBuffer = productBuffer;
  }

  public void consume() {
    Product prod = productBuffer.get();
    System.out.println(prod + "␣consumed.");
  }
}

class ProductBuffer {
  Vector contents = new Vector();

  public BufferProduct() {}

  public Product get() {
    Object p = contents.firstElement();
    contents.removeElementAt(0);
    return (Product)p;
  }

  public void put(Product p) {
    contents.add(p);
  }
}
```

Obviously, we have to perform many invasive modifications to replace the interaction mechanism. The remaining sections discuss these modifications in detail and show how we support them by a tool. In this special case, the tool can perform the changes fully automatically.

3 Invasive Composition and COMPOST

The idea of *invasive composition* is to compose software components by invasively modifying them using automated program transformations. The underlying model consists of the following three general elements:

box components: represent program units to be composed. A box is a collection of program elements like a package, some compilation units, a single compilation unit, several classes or a single class. a box component maintains a set of hooks as composition interface.

hooks: Technically, a hook is a set of program elements, too. In contrast to a box, a hook always refers to a containing box. The hook's program elements are a subset of the program elements of this box. Conceptually, a hook is a join point, that identifies a part of a component that may be extended or replaced.

composers: A composer is a program transformer which transforms composables, either boxes or hooks.

Thus, invasive composition adapts or extends components at their hooks by program transformations.

3.1 Boxes

Boxes are organized in form of the composite pattern, i.e. they constitute a hierarchy of components. Each composition system (i.e. each project) has one root box. There are

- composite boxes which contain composition programs and other subboxes
- atomic boxes which consist of hand-written Java code.

Each composite box in the box hierarchy carries a composition program by which all subboxes are composed and configured. Atomic boxes do not carry composition programs.

3.2 Hooks

Box components can be configured in complex ways by program transformations. Box writers can indicate where such transformations can be applied, using the concept of *hook*s. A hook is a set of program elements that constitute a join point, identifying a part of a component that may be extended or replaced.

Hooks can be described by

- context-free or context-sensitive patterns which can be matched in the abstract syntax of a box and be replaced. The patterns may be term patterns or graph patterns. Then the hook is a reducible expression (redex) of the pattern in the program's representation.
- selection queries on the program's representations. The query language may be any language that can query graphs, such as Datalog, Prolog, OQL, or a graph database query language. Then the hook is a query result, typically a set of program elements.
- names. A rather simple kind of hooks are named program elements (the appropriate query would just look for a name and result in one program element).

Hooks are further classified into declared and implicit hooks.

Declared (Explicit) hooks denote a certain code position with a name.

Implicit hooks are hooks that are given by the programming language implicitly (here Java). Implicit hooks refer to a syntax element which can be qualified (a compilation unit, a class, a method). Each hook carries its name, and also a qualifier which determines its position in the component. Since implicit hooks need not be declared, the hook name is not a naming scheme, but the name of the hook itself; the hook qualifier plays the role of the naming scheme.

3.3 Composers

A composer is a program transformer which transforms composables, either boxes or hooks. Figure 2 illustrates the effect of a composer.

General classification criteria for composers are

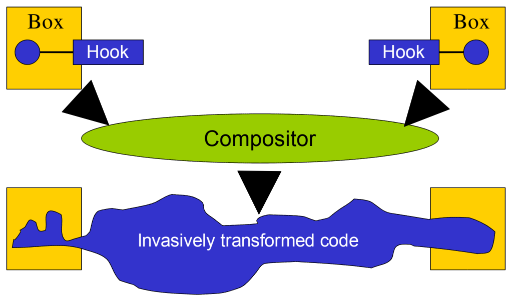

Fig. 2. The effect of a COMPOST composer

- taking composables, i.e. hooks or boxes, as arguments (hook-based or box-based)
- unary, binary, n-ary
- elementary or composed

Composers for Hooks The most important unary composer on hooks is the *bind composer*: It overwrites the hook. If it was a declared hook the hook is eliminated, i.e. cannot be rewritten anymore. Implicit hooks cannot be eliminated from the boxes since they are defined by the programming language semantics.

Composers for list hooks are:

extend: Extends a box or a hook with a mixin box or a value.
append: Appends a value to the hook and retains the hook. (Also for point hooks in list hooks)
prepend: Prepends a value to the hook and retains the hook. (Also for point hooks in list hooks)
wrap: List hooks can be wrapped with two values. The first is prepended, the second appended.
extract: Delete an element of a list hook or a point hook completely.

Composers on boxes (box-based Composers) Boxes can be composed themselves without referring to their hooks. Such composers refer to the boxes' hooks implicitly; not all hook operations are applicable.

Composition Programs and Composition Terms COMPOST is a Java library. When composers of this library are applied on boxes and hooks, composition programs in Java result. These composition programs create composition terms from composers and their arguments, as well as forests of composition terms, i.e. composition forests.

Composition terms are one of the central data structures of COMPOST since they describe compositions abstractly. Each object in a composition forest represents a composition operation, i.e. an application of a composer, and represents a Command object in terms of the design pattern Command [6].

3.4 Using COMPOST

Composition programs which use the library should be written as in the following. Composition operations have the calling schema

Unary hooks: `<box>.findHook(HookName).<composer>(Code)`
Binary composers: `<composer>(HookName1, HookName2)`

Currently, only named hooks can be replaced by COMPOST composers. To find a hook, the hook name has to be handed over to the hook finder (`box.findHook(String)`). The hook's name is qualified with the scopes of the program in which it is contained. Qualification is by dot (.).

For our running example the composition program looks like the following.

```
// Prepare the composition by allocating a composition system with
// im- and exporter and all necessary services.
CompositionSystem project = new CompositionSystem(basepath);

// Load the box ProducerConsumer from file ProducerConsumer.coc
CompilationUnitBox pc = new CompilationUnitBox(project,"ProducerConsumer");


// This sets lazy mode with composers as command objects and
// defered transformations
project.setLazyMode();

// Re-compose the Producer and Consumer classes.
try {
  // Add set method for reference to ProductBuffer to Consumer
  pc.findHook("Consumer.members").prepend("public␣void␣setProductBuffer(
␣␣␣␣␣␣ProductBuffer␣productBuffer)␣{
␣␣␣␣␣␣␣␣this.productBuffer␣=␣productBuffer;
␣␣␣␣␣␣}");

  // Add default constructor to Consumer class
  pc.findHook("Consumer.members").prepend("public␣Consumer()␣{␣}");

  // Add private attribute to refer to ProductBuffer to Consumer class
  pc.findHook("Consumer.members").prepend("private␣ProductBuffer␣productBuffer;'');

␣␣//␣Remove␣parameter␣from␣Consumer's␣consume␣method
␣␣pc.findHook("Consumer.consume.parameters").extract("prod");

␣␣//␣Add␣call␣to␣get␣method␣of␣buffer␣component
␣␣pc.findHook("Consumer.consume.statements").prepend("Product prod = productBuffer.get();");

␣␣//␣similar␣actions␣to␣adapt␣the␣Trader␣and␣Producer␣classes␣as␣well␣as␣to
␣␣//␣insert␣the␣ProductBuffer␣class.

␣␣␣//␣In␣lazy␣mode,␣this␣executes␣the␣transformations
␣␣␣pc.execute();

␣␣␣//␣Export␣the␣Producer-Consumer␣Box␣again
␣␣␣//␣(Default␣is␣to␣ProducerConsumer.java)
␣␣␣project.getBoxInfo().default.Export();

}␣catch␣(Exception␣e)␣{
␣␣␣System.out.println("error: composition failed. ");
}
}
```

In our example, `project.setLazyMode()` sets lazy mode. By default, all compositions are directly executed on the abstract syntax tree of the box components (eager mode). In lazy mode, only composer command objects are built. When the method `pc.execute()` is called, the composer command objects are executed, and the transformations are committed in the abstract syntax tree.

3.5 Conclusion

Although we have omitted some composition operations, our composition program has become quite complex. Before we elaborate on the complex transformations to simplify the task of replacing the direct method call by communication via a buffer (Section 6), we first present the details of the underlying framework Recoder (Section 4), that is the common basis for all our transformation tools. We also introduce the the scripting language InjectJ to control Recoder analyses and transformations, first (Section 5).

4 The Recoder Tool and Framework

Recoder is a Java framework for source code meta-programming aimed to deliver a sophisticated infrastructure for many kinds of Java analysis and transformation tools. The system allows to parse and analyze Java programs, transform the sources and write the results back into source code form. Figure 3 illustrates this cycle.

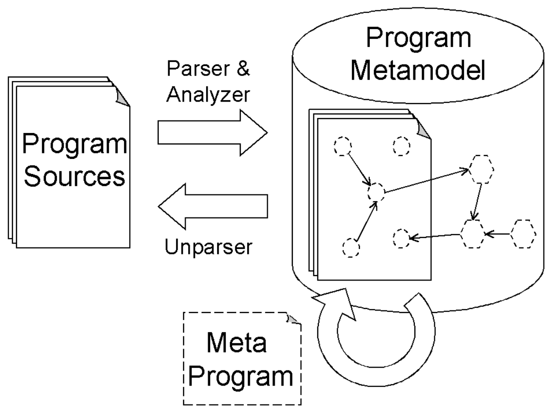

Fig. 3. The Recoder meta programming cycle

4.1 The Recoder Program Model

To enable meta programming, Recoder derives a meta model of the entities encountered in Java source code and class files. This model contains a detailed syntactic program model (including comments) that can be unparsed with only minimal losses. The syntactic model is an attributed syntax tree (in fact a graph), where each element has links to its children as well as to its parent, in order to support efficient upward navigation. All properties of a syntactic element are modeled by corresponding types, i.e., the element has to implement the corresponding interfaces. Figure 4 shows this for a method declaration as an example.

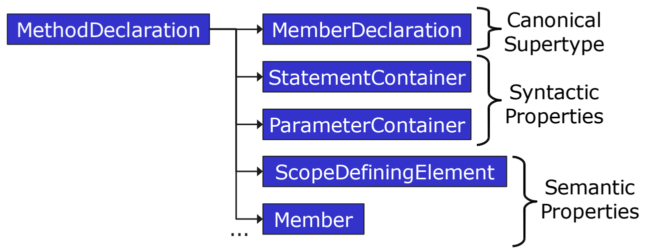

Fig. 4. The Recoder model for properties of a syntactic element

Figure 5 shows the base elements of the Recoder meta model, some sample elements and services that deal with the interface-level abstractions in different representations.

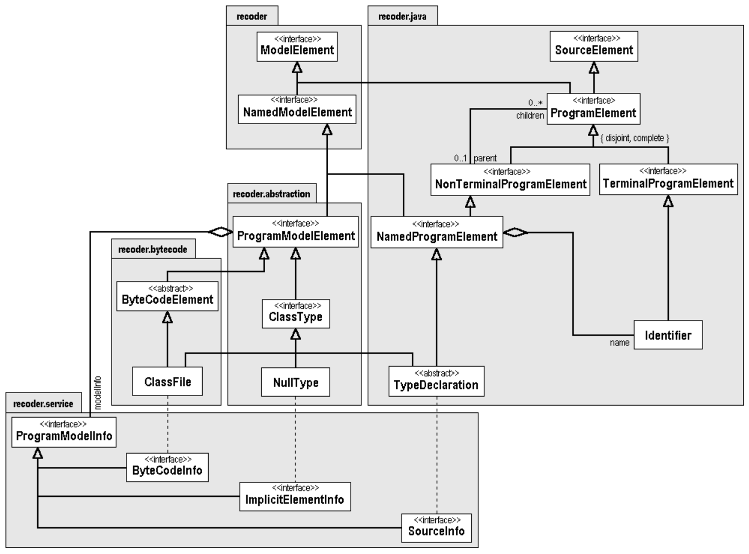
recoder
<<interface>> ModelElement
<<interface>> NamedModelElement
recoder.java
<<interface>> SourceElement
<<interface>> ProgramElement
0..*
children
{ disjoint, complete }
0..1 parent
<<interface>> NonTerminalProgramElement
<<interface>> TerminalProgramElement
recoder.abstraction
<<interface>> ProgramModelElement
<<interface>> NamedProgramElement
recoder.bytecode
<<abstract>> ByteCodeElement
<<interface>> ClassType
Identifier
name
ClassFile
NullType
<<abstract>> TypeDeclaration
modelInfo
recoder.service
<<interface>> ProgramModelInfo
<<interface>> ByteCodeInfo
<<interface>> ImplicitElementInfo
<<interface>> SourceInfo

Fig. 5. The base elements of the Recoder meta model

The topmost type for semantic model data is `recoder.ModelElement` which can represent arbitrary model data such as the majority of syntax nodes, annotations, or syntactic footprints of design patterns. `recoder.NamedModelElements` are model elements that feature a meaningful name for each instance.

The topmost type for syntactic Java source data is `recoder.java.SourceElement` representing any node in a syntax tree such as comments, which do not necessarily carry semantic information. Byte code elements are also syntactic. They are subtypes of `recoder.bytecode.ByteCodeElement`.

`recoder.abstraction.ProgramModelElement`s are part of the abstract model and represent entities visible at the interface level (hence, they are all NamedModelElements). Currently, all ByteCodeElements covered by Recoder are ProgramModelElements, but not all ProgramElements (a Plus operator is not contained in the abstract model).

The core part of the Recoder meta model is located in `recoder.abstraction` and primarily consists of entities that occur in an API documentation: Types, Variables, Methods, Packages, with some additional abstractions such as Member or ClassTypeContainer. These entities inherit from `ProgramModelElement`.

While many ProgramModelElements have a syntactic representation, the `recoder.abstraction` package also contains entities that have no syntactic representation at all, but are implicitly defined. Examples are ArrayType, DefaultConstructor, or the aforementioned Package.

Figure 6 shows the elements of this abstract model and their associations.

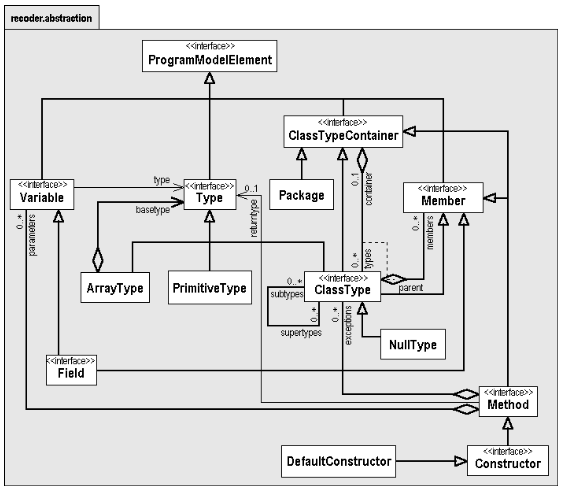

Fig. 6. The elements of the Recoder abstract model

While the syntactic model provides only the containment relation between elements, the complete model adds further relations, such as type-of, or refers-to, as well as some implicitly defined elements, such as packages or primitive types.

In order to derive this semantic information, Recoder runs a type and name analysis which resolves references to logical entities. The refers-to relation can be made bidirectional for full cross referencing which is necessary for efficient global transformations.

4.2 Recoder Program Transformations

Recoder program transformations operate on syntax trees. Basic transformations are `attach` that attaches a node or subtree to an existing node or tree, and `detach` that detaches a node or subtree from an existing tree. The central `ChangeHistory` service collects all change objects (performed attach and detach operations) and notifies the registered services that are responsible for keeping the program model consistent. The model is not rebuilt for every change operation, instead the services only update the model when a new model query arrives. This minimizes the number of costly model updates. Unless committed a transformation can also be rolled back. Figure 7 illustrates the interconnection of change reporting services (`SourceFileRepository` service and Transformation class) and model maintaining services (`SourceFileRepository`, `NameInfo`, `SourceInfo`, `CrossReferencer`) with the change history service.

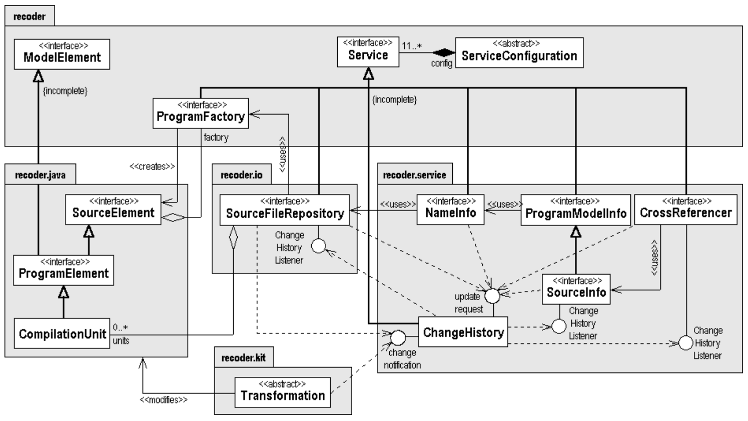

Fig. 7. Connection of Recoder services to ensure model consistency

Before a transformation is performed, its applicability has to be checked. This is done during an analysis phase that also collects the information necessary to perform the transformation. If the analysis phase indicates that the transformation can be performed without problems, the consecutive transformation phase performs the changes. This two pass transformation protocol ensures that a transformation is always based on valid information. This is especially required for composed transformations. Otherwise, results obtained by a model query may have been invalidated by a transformation step as a side effect, so that another transformation step of the composed transformation uses invalid information and will therefore corrupt the program.

4.3 Using Recoder

Recoder meta programs are written as follows, using our running example.

```
// Create a service configuration, first:
ServiceConfiguration serviceConfig = new CrossReferenceServiceConfiguration();

// Get the file management service:
SourceFileRepository sfr = serviceConfig.getSourceFileRepository();

// Get the program factory service:
ProgramFactory pf = serviceConfig.getProgramFactory();

try { // parse file
    CompilationUnit cu = sfr.getCompilationUnitFromFile("ProducerConsumer.java");
} catch (ParserException pe) {
    // do something
}

// Create a new syntax tree for the ProductBuffer class:
TypeDeclaration td = pf.parseTypeDeclaration("class␣ProductBuffer␣{␣...␣}");

// Create a custom transformation object:
Transformation t = new Transformation(serviceConfig) { };

// Add the declaration of the ProductBuffer class to the compilation unit:
t.attach(td, cu);

// Similar actions for the remaining adaptation steps...

// Write the changes back to file:
PrettyPrinter pp =
    pf.getPrettyPrinter(new PrintWriter(
      new FileWriter("ProducerConsumerTransformed.java")));
cu.accept(pp);
```

4.4 Conclusion

A pure Recoder meta program to perform the transformation task of our running example is quite too complex to demonstrate here. Instead, we use the Recoder infrastructure to construct special complex transformations to solve this task. These transformations are part of the COMPASS tool and framework (cf. Section 6).

5 The InjectJ Tool

InjectJ is an operational scripting language for invasive composition. It offers means to control the analyses and transformations of Recoder and therefore provides the following operations:

Navigation over a simplified structure graph (simplified with respect to the program model of Recoder):

```
foreach c in classes(**) {
  foreach m in c.methods {
    foreach a in m.accessesToSubtype('SomeClass') {
    }
  }
}
```

Transformations like injecting code before a syntax element:

```
a.before(${ <einzufgender Code> }$);
```

Control structures like if-statements:

```
if (introduceThrows) {
  m.addToThrows(class('ResourceAccessException'));
}
```

User interaction:

```
ask("Please␣choose␣a␣name␣for␣a␣temporary␣variable", name);
```

Navigation paths and hooks can be defined by the user, the analyses and transformation operations cannot, they are built-in.

5.1 Using InjectJ

InjectJ scripts are written as follows, using our running example.

```
script IntroduceBuffer {

  // search Producer/Consumer classes
  producerClass = class('Producer').get(0);
  consumerClass = class('Consumer').get(0);

  // search produce/consume methods
  produceMethod = producerClass.getMethod("produce()");
  consumeMethod = consumerClass.getMethod("consume(java.lang.Object)");

  // Introduce new members to Producer class, e.g. reference to the
  // new buffer
  producerClass.addToMembers (${
    ProductBuffer productBuffer;

    public void setProductBuffer(ProductBuffer productBuffer) {
      this.productBuffer = productBuffer;
    }
  }$);

  // remove references to Consumer class
  foreach att in producerClass.getAttributes() do {
    if (att.staticType==consumerClass){
      // remove the assignment in the constructor and
      // the call to the consume method
      foreach ref in att.getReferences() do {
        ref.delete;
      }
      att.delete;
    }
  }
  foreach par in produceMethod.getParameters() do {
    if (par.staticType==consumerClass) { par.delete; }
  }

  // add call to buffer to body of produce method:
  produceMethod.getBody().append("productBuffer.put(p)");
  // introduce new members to Consumer Class
  consumerClass.addToMembers( ${

    ProductBuffer productBuffer;

    public void setProductBuffer(ProductBuffer productBuffer) {
      this.productBuffer = productBuffer;
    }
  }$);

  // remove parameter from consume method:
  productClass = class('Product').get(0);
  foreach par in consumeMethod.getParameters() do {
    if (par.staticType==productClass) { par.delete; }
  }

  // add call to buffer to body of consume method:
  consumeMethod.getBody().prepend("Product p = productBuffer.get(p);");
} // end of script
```

5.2 Conclusion

Our InjectJ script to replace the direct method call by communication via a buffer has become quite complex. In the next section we present the COMPASS tool tailored to perform such a program transformation.

6 The COMPASS Tool and Framework

COMPASS (COMPosition with AspectS) is designed to consistently exchange and adapt interactions among software units. It is based on an architectural model consisting of the three levels depicted in Figure 8.

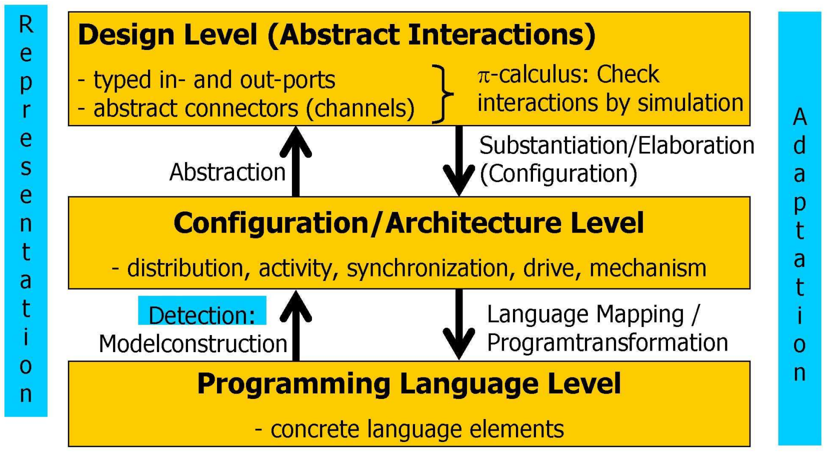
Representation
Design Level (Abstract Interactions)
- typed in- and out-ports
- abstract connectors (channels)
π-calculus: Check interactions by simulation
Abstraction
Substantiation/Elaboration (Configuration)
Configuration/Architecture Level
- distribution, activity, synchronization, drive, mechanism
Detection: Modelconstruction
Language Mapping / Programtransformation
Programming Language Level
- concrete language elements
Adaptation

Fig. 8. Interaction Configuration Levels.

– The *programming language level* contains the concrete program to adapt. So the first step of COMPASS transformations is the *detection phase (model construction)* that analyzes the source code to identify components, their interactions and interaction patterns. Interaction patterns (especially design patterns [6]) are identified combining static and dynamic analyses as described in our earlier papers [15, 14, 16, 17].
– The *configuration level* results from the detection phase. It is a program representation suited to manually specify interaction configurations. This level abstracts from the concrete programming language and represents all interactions explicitly as first-class entities combining the ideas of architecture systems [8] and aspect-oriented programming [19] as well as hyperdimensional separation of concerns [24]. Figure 9 illustrates the elements of the configuration level and their links to corresponding source code elements.

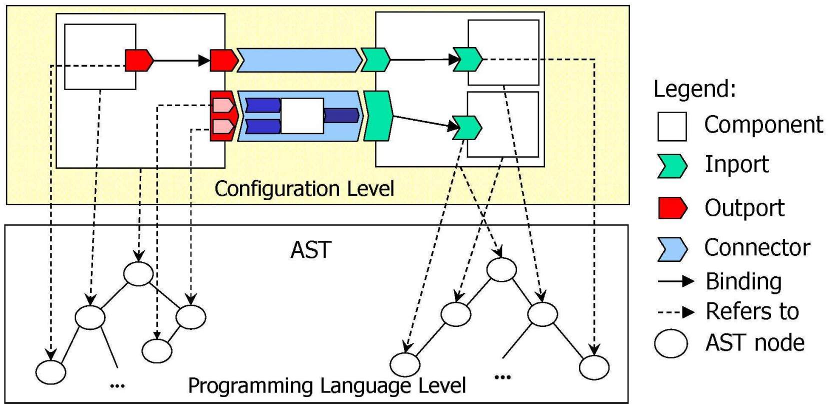

Fig. 9. Configuration Level Elements, their Structure and their Links to Source Code.

The elements of the configuration level are:

- *components* that are the basic building blocks of every system. A COMPASS component is a box in the sense of COMPOST. Table 1 shows the mapping of Java syntax elements to COMPASS component model elements. Whenever the iterator of the detection and model construction phase encounters such a syntax node, it creates an instance of the corresponding COMPASS component model element. Of course, components may be hierarchically composed of further components including subsystems.
- *aspect-oriented ports* that encapsulate the interaction points of components in an aspect- or hyperslice-like fashion and represent the interaction properties at these points. Interaction properties are for example the direction of the interaction (in or out), the type of the interaction (control or data), synchronization, drive (initiation of control flow), and

Java element	**COMPASS element**
compilation unit	ModuleComponent
class declaration	ModuleComponent
method declaration	ProcedureComponent
field	FieldComponent
variable	VariableComponent

Table 1. Mapping of Java Syntax Elements to COMPASS Component Model Elements.

interaction mechanism. A COMPASS port is a hook in the sense of COMPOST.
The model construction phase constructs a COMPASS port model element instance for every syntax node denoting data or control flow, like a method call for example.
Some ports of components are implicitly defined by the language semantics without an explicit syntactical representation in the programming language. Since our model aims at making all interactions explicit, we also create explicit port model element instances for those and associate them with their owning components. An example is the implicit first parameter (`this`) of a Java method, identifying the object to which the method belongs, i. e., the object which state has to be considered when referring to fields.

- *aspect-oriented connectors* that represent interactions by connecting ports. A connector also is a representation of a program transformation. Via the ports it connects, a connector has access to the interaction points buried in the components' code. It can thus substantiate or replace this code by the configured interaction code. Even connections established by private fields are considered, but the corresponding ports are marked as internal component ports. A COMPASS connector is a composer in the sense of COMPOST.

These component, port and connector entities constitute an architectural model.
In the *(re-)configuration phase*, the developer reconfigures and adapts interactions by exchanging the port and connector entities on the configuration level. This triggers corresponding source code transformations (*transformation phase*) implemented as meta programs using the Recoder framework. Since connectors are architecture and design level instances by nature, our source code transformations eliminate them in the final code mapping them to potentially several constructs of the target language. Nevertheless, we retain the configuration level representation of the system as an architectural representation.
A program transformation consists of the two passes *analysis* and *transformation*. The *analysis pass* first collects the information necessary to perform the transformation by transitively identifying the ports and connectors affected by the transformation. These provide the relevant information already collected during the model construction phase. This especially comprises the

source code elements to modify. Second, the analysis pass checks if the transformation can be executed, i. e., if the required information are available and the transformation is applicable in the given context. It might happen for example that a Java class is only available as byte code, so that we cannot transform its source code. Moreover, the configuration may define to connect incompatible ports using the wrong type of connector. The *transformation pass* carries out the transformation without performing any further analyses. This is important, since a sub-transformation may have changed the underlying system already, so that the analysis now produces different or misleading results, although the whole complex transformation should be regarded as atomic. The whole transformation is encoded as a Java program, that modifies the abstract syntax forest of the sample program. The final target code is then produced using Recoder to pretty print the modified syntax forest.
  Base transformations of COMPASS are to append or remove ports and/or components, as well as to insert or remove connectors.
– The *abstract level* abstracts from the concrete interaction properties represented on the configuration level. On the abstract level interaction consists of input and output actions on typed channels. This allows to bridge architectural mismatches or mismatches caused by certain implementation techniques.
  An *abstraction phase* maps configuration level elements to abstract model elements, a *substantiation phase* conversely maps abstract model elements to configuration level elements by adding implementation specific interaction properties like synchronization, drive and mechanism.

The semantics of these levels is defined formally using the π-calculus, i. e., every interaction is resolved to one or more input ($c(v)$) or output actions ($\bar{c}v$) on channels (c). We have presented the details of this model in [13].

6.1 Using COMPASS

The transformation task of our running example amounts to exchanging the procedure call connector by a buffered data transfer connector as depicted in Figure 10. The developer performs this reconfiguration on the configuration level.

This reconfiguration is implemented as the following single COMPASS transformation

```
public class ReplaceProcedureCallConnectorByBuffer
    extends CompassTransformation {
  ...
  public ReplaceProcedureCallConnectorByBuffer(CompassConfiguration config,
                                               ProcedureCallConnector procCallConnector) {
      ...
  }
```

that consists of 348 lines of Java code (including blank lines and comments), using the base transformations of COMPASS and the Recoder framework.

This transformation can simply be invoked by

```
ProblemReport pr = new ReplaceProcedureCallConnectorByBuffer(config, pc).execute();
```

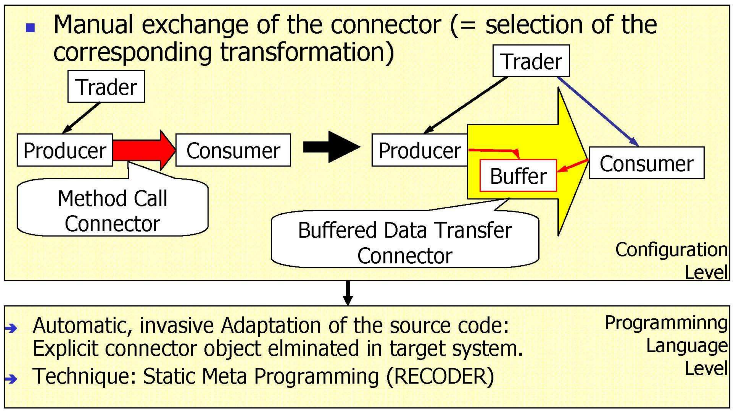
Manual exchange of the connector (= selection of the corresponding transformation)
Trader
Producer
Consumer
Method Call Connector
Trader
Producer
Buffer
Consumer
Buffered Data Transfer Connector
Configuration Level
Automatic, invasive Adaptation of the source code: Explicit connector object elminated in target system.
Technique: Static Meta Programming (RECODER)
Programminng Language Level

Fig. 10. Reconfiguration of Producer-Consumer-System Method Call.

where

– `config` is a concrete COMPASS configuration that consistently bundles the source code parser, the mapping to model elements and the source code transformations including the pretty printer for a concrete programming language (Java in the current implementation of COMPASS), and
– `pc` is the instance of the procedure call connector to replace by a buffered communication connector.

6.2 Conclusion

COMPASS offers transformations tailored to reconfigure component interactions. To perform the transformation task in our running example we therefore just need to reconfigure the system by calling the corresponding transformation.

7 Related Work

Most of the work in this area is concerned with general approaches to software design, decomposition, composition and evolution [19, 24, 2], program understanding and refactoring. Only few works specifically deal with adapting interactions.

Architecture systems [8] like Darwin [5], UniCon [27], Rapide [20] and Wright [1] also introduce *port* entities to represent interaction interfaces of components and *connector* entities to represent interactions among components. But these architecture systems are unable to adapt the interactions of already existing code, because they do not implement architecture reconstruction algorithms.

Carriere et al. [4] generate program transformations to exchange communication primitives in C source code. First, they identify communication primitives using regular expressions and naming conventions. Then, they use the mapping defined in [18] to specify pre- and post-conditions to generate corresponding program transformations to be performed by the tool Refine/C [26, 10]. These transformations then transform the abstract syntax tree of the program under consideration, automatically. This approach has the following shortcomings: the mapping of communication mechanisms is neither unique nor 1:1, thus leading to dead or superfluous code in the target system. Moreover, this includes the problem of deciding which target mechanism to use. In COMPASS this is done interactively or by providing a transformation strategy. Carriere et al. do not solve this problem. Furthermore, their approach does not detect complex interaction patterns. This is due to the fact that their analysis is based on naming conventions and regular expressions that are not powerful enough.

Pulvermüller et al. [25] encapsulate CORBA-specific communication code in aspects implemented using AspectJ and present a few highly specialized transformations. But they do not define an interaction model, do neither provide general support to adapt interactions nor to adapt interaction patterns. An analysis phase to detect interactions and provide a representation suitable to adapt them is completely missing.

Altogether, we do not know any approach that deals with the complete procedure of adapting interactions, as we did. The stepwise refinement of architectures defined by the COMPASS model has now become popular using the term *model driven architecture (MDA)* [23].

8 Conclusion

We have presented a suite of program transformation tools for invasive software composition and applied them to a representative interaction adaptation task. The COMPASS tool is tailored to tackle this kind of tasks. It combines the ideas of architecture systems, aspect-oriented programming and hyperdimensional-separation of concerns to define *ports* that provide invasive access to interaction points of *components*, and *aspect-oriented connectors* as program transformations to consistently adapt interactions according to specified configurations.

Our future work comprises to implement further transformations thus expanding our transformation library. Moreover, we need to deal with the problem of how to find the interactions to adapt when a problem specification or a catalog of known problems is given. The detection of anti-patterns and the application of metrics to apply refactorings are promising in this domain, but need to be extended to non-meaning-preserving transformations, in the sense that desired new observable properties can be added to the target system.

References

1. Robert Allen and David Garlan. Beyond definition/use: Architectural interconnection. In *ACM IDL Workshop*, volume 29(8). SIGPLAN Notices, 1994.
2. Uwe Aßmann. *Invasive Software Composition*. Springer, 2002.
3. Uwe Aßmann, Andreas Ludwig, Rainer Neumann, and Dirk Heuzeroth. The COMPOST project main page. `http://www.the-compost-system.org`, 1999 – 2005.

4. S. J. Carriere, S. G. Woods, and R. Kazman. Software Architectural Transformation. In *WCRE 99*, October 1999.
5. Darwin. `http://www.doc.ac.ic.uk`, 2000.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
7. David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 1995.
8. David Garlan and Mary Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing, 1993.
9. Thomas Genßler. Inject/j. `http://injectj.fzi.de/`, 1999 – 2005.
10. David R. Harris, Alexander S. Yeh, and Howard B. Reubenstein. Extracting Architectural Features From Source Code. *ASE*, 3:109–138, 1996.
11. Dirk Heuzeroth. The COMPASS project main page. `http://www.info.uni-karlsruhe.de/~heuzer/projects/compass`, 2003.
12. Dirk Heuzeroth. COMPASS: Tool-supported Adaptation of Interactions. In *Automated Software Engineering 2004*. IEEE, 2004.
13. Dirk Heuzeroth. A Model for an Executable Software Architecture to deal with Evolution and Architectural Mismatches. Technical report, Universität Karlsruhe, 2004.
14. Dirk Heuzeroth, Gustav Högström, Thomas Holl, and Welf Löwe. Automatic Design Pattern Detection. In *IWPC*, May 2003.
15. Dirk Heuzeroth, Thomas Holl, and Welf Löwe. Combining Static and Dynamic Analyses to Detect Interaction Patterns. In *IDPT*, June 2002.
16. Dirk Heuzeroth and Welf Löwe. *Software-Visualization - From Theory to Practice, Edited by Kang Zhang*, chapter Understanding Architecture Through Structure and Behavior Visualization. Kluwer, 2003.
17. Dirk Heuzeroth, Welf Löwe, and Stefan Mandel. Generating Design Pattern Detectors from Pattern Specifications. In *18th ASE*. IEEE, 2003.
18. Rick Kazman, Paul Clements, and Len Bass. Classifying Architectural Elements as a Foundation for Mechanism Matching. In *COMPSAC 97*, August 1997.
19. Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented Programming. In *ECOOP'97*, pages 220–242. Springer, 1997.
20. David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE ToSE*, 21(4), 1995.
21. Andreas Ludwig and Dirk Heuzeroth. Metaprogramming in the Large. In *GCSE, LNCS 2177*, pages 443–452, October 2000.
22. Andreas Ludwig, Rainer Neumann, Dirk Heuzeroth, and Mircea Trifu. The RECODER project main page. `http://recoder.sourceforge.net`, 1999 – 2005.
23. OMG. MDA Guide Version 1.0.1. Technical report, OMG, 2003.
24. Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns in Hyperspace. Technical report, IBM T. J. Watson Research Center, April 1999.
25. E. Pulvermüller, H. Klaeren, and A. Speck. Aspects in distributed environments. In *GCSE'99*, September 1999.
26. `http://www.reasoning.com`, 2003.
27. Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE ToSE*, 21(4), 1995.