UDC 681.3.064

Composing Transformations of Compiled Java Programs with Jabyce

Romain Lenglet, Thierry Coupaye, and Eric Bruneton

France Telecom – R&D Division
28 chemin du Vieux Chêne, 38243 Meylan, France
{romain.lenglet, thierry.coupaye, eric.bruneton}@francetelecom.com

Abstract. This article introduces Jabyce, a software framework for the implementation and composition of transformations of compiled Java programs. Most distinguishing features of Jabyce are 1) its *interaction orientation*, i.e. it represents elements of transformed programs as interactions (method calls), which generally consumes less memory and CPU time than representing programs as graphs of objects; and 2) its *component orientation*, i.e. it allows for the design and composition of transformers as software components based on the Fractal component model. This latter point is strongly connected to infra-structural and architectural issues, and software engineering aspects such as composing, scaling, maintaining and evolving transformers. Jabyce is compared with other existing compiled Java programs transformation systems, using an extension of a previous well-known categorization of program transformation systems.

1. Introduction

The work presented in this article is motivated by the increasing need for *adaptability* in distributed systems. Distributed systems and middleware platforms are deployed in highly heterogeneous and highly evolving environments in terms of computing resources (processing, memory, database connectivity, network resources…). Therefore they need to be easily specialized and configured (assembled) statically and dynamically. Program transformation is one of the most general and efficient technique for the adaptation of complex systems, in a non intrusive way.
Indeed, program transformation deals with the automated modification of programming elements by (other) special executable programming elements called *transformers*. Traditionally, program transformation has been very much connected to the software engineering area. Most program transformation systems [9] have been or are developed by software engineering teams or groups in the context of software maintenance. Most often described uses of program transformation hence include software

evolution, refactoring, change logging, reverse engineering, etc. Software development and programming languages also make use of program transformation for compilation, optimization, partial evaluation, etc. In contrast with those uses, in the context of complex distributed systems, the main targeted uses of program transformation concern load-time weaving of code as a support for runtime adaptability (transparent insertion of so-called *technical services* or *non functional aspects* to components and composition of these technical services). In that respect, motivations for the development of transformation systems, such as the Jabyce transformation system presented in this article, converge with that of aspect weaving in Aspect Oriented Programming (AOP) which is strongly related with program transformation - but with a special interest in dynamic weaving.

Jabyce[1] is a software framework that allows for the implementation and composition of software components (*transformer components*) that transform compiled Java classes. The main goals concerning the design of this framework are the following:

- to be able to *implement any transformations* of compiled classes in contrast to some transformation systems that limit themselves to a subset of all the transformations - for example to ensure some integrity constraints.
- to be able to *separate transformers* as individual software components, that can then be *composed without modifying their code*. This is important to be able to reuse transformers, which are generally complex pieces of software.
- to be able to *compose transformers in an efficient way*. In particular, the system should avoid redundant computations made in separate transformers.

These objectives have guided the design process of Jabyce, which is presented in this article. We also consider them as *criteria* to compare Jabyce to other transformation systems. From a general point of view, Jabyce is the only Java bytecode transformation system that reaches all those three objectives simultaneously.

These objectives are achieved 1) by using well known object oriented design patterns and principles - such as the Interface Segregation Principle -and by using a component-based architecture; and 2) by introducing the representation of transformation program elements as sequences of interactions, instead of the generally used graphs of objects or terms.

The rest of this article is organized as follows. Section 2 introduces and extends the characteristics (or dimensions) that have been introduced in a previous categorization of program transformation systems. Sections 3 to 6

[1] In “Jabyce”, “ja” is pronounced as in “jacket” and “abyce” like “abyss”.

describe Jabyce and compare it with other compiled Java program transformation systems, along the dimensions of that extended categorization. Section 7 provides a more general and synthetic description of Jabyce. Section 8 details an example of a Jabyce transformer, and execution time measurements that are compared with those of two other transformation systems. Finally section 9 shows how the design process used in Jabyce can be generalized, and the conclusion gives some perspectives for future work.
From a general point of view, the purpose of this article is twofold. It is both theoretical and practical, as 1) it considers new theoretical aspects of program transformation systems, in the form of an extension of a well-known categorization of such systems, and 2) it thoroughly describes the Jabyce transformation system, which is implemented and functional, and the process of its design.

2. Categorization of Program Transformation Systems

We define the following terms that will be used in the rest of this article to describe Jabyce and other systems homogeneously. Although no homogeneous terminology is used in the literature, the following terms are a good compromise. We call *transformations* the computations that produce a transformed program from an original program. Transformations are performed by runtime entities called *transformers*, which are software components that may be composed into various *transformer configurations*. A *transformation system*, for instance Jabyce, is an infrastructure, or a *software framework*, used to develop transformer implementations and transformation programs. In Jabyce, we use a second term to refer to a special kind of transformers: transformation *operations*. In Jabyce, the difference between a transformer component and an operation component lies in their architecture, as further explained in section 7.1.
[32] presents a taxonomy and categorization of program transformation systems. The taxonomy deals mainly with the *scope* of transformation systems but the whole survey in itself can be considered as a more general categorization as it introduces additional dimensions, or characteristics: *program representation* and *transformation paradigm*. The rest of this section provides an overview of this categorization.

2.1. Scope

The *scope* dimension concerns the possible uses of a transformation system, i.e. the range of transformer implementations and transformation

programs that can be implemented using it. [32] provides a taxonomy of program transformation systems, according to the possible uses of such systems, that is illustrated as a tree in Fig. 1. Two main categories are identified:

- *rephrasing*: programs are transformed into programs in the same model;
- *translation*: programs in a *source* model are transformed into programs in a different *target* model.

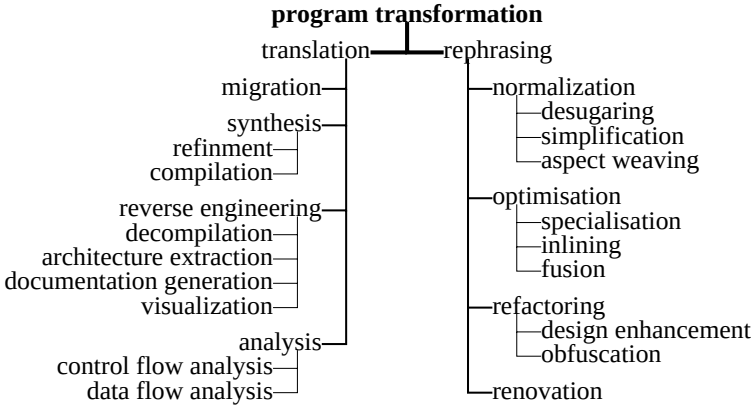

Figure. 1. Possible uses of program transformation

All those uses are relevant in the context of transformation of compiled Java programs. For example, a compiled Java program can be *normalized* by removing its debugging information. As another example, a compiled Java program can be optimized, by removing dead code and unused methods and fields [31]. Also, method calls can be *devirtualized*, i.e. some virtual method calls can be resolved to particular implementations of such methods, and be performed faster at runtime. As a very common use of translation, compiled Java programs must be interpreted or compiled into native binary programs, in order to be executed. Compilation can occur statically, or dynamically during the program execution using a "Just-In-Time" compiler.

2.2. Transformation Paradigm

This dimension deals with the computational and architectural models of transformation systems. We consider that these models are the result of choices that answer to the following questions:

- How to express or implement transformers? This is related to the "transformation language" design activity.
- How to package and deploy transformers? This is related to the programming interfaces design, transformer maintenance and testing activities.
- How to control the computation of transformations, and how to compose transformers? These questions relate to the design of the infra-structural environment required to develop transformers.

The categorization proposed in [32] is part of a survey of the transformation strategies based on *declarative rewrite rules*, also called *schematic rules* or *pattern replacement rules* [27]. There are two general ways to specify transformations as declarative rules: rules are either interpreted or compiled into an imperative programming language, as it is done in Stratego [32]. It is also possible to specify transformations directly in an imperative programming language, such as Java or C. Such transformer implementations are called *procedural rules* [27]. As this article considers procedural rules, in contrast with [32], it also considers more general software architecture issues, such as software component composition.

2.3. Program Representation

The program representation dimension concerns the choice of a representation model for programs to be transformed. In [32] it is asserted that programs are either represented as *trees* or as *graphs* of objects (or *terms*). As an innovative characteristic, Jabyce represents programs as *sequences of interactions* between transformers. This new way of representing program elements is described in details in the rest of this article. As a consequence, we propose an *extension of Visser's categorization*, by splitting the program representation dimension into two related dimensions:

- the *conceptual model* dimension: the choice of the types of the elements of programs that are represented to be transformed.
- the *programmatic model* dimension: the abstractions of the transformer implementation language that are used to make the represented

program elements concrete. This dimension is closely related to the transformation paradigm dimension.

The two categories of programmatic models that we have identified are: 1) graphs of objects (e.g. structs, in C), and 2) sequences of interactions (method calls). These two dimensions are independent: a given conceptual model can be combined with either an object graph representation, or an interaction sequence representation.

2.4. Proposed Extended Categorization

As a conclusion, we propose the following extension of Visser's transformation system categorization proposed in [32], that consists of four dimensions:

- *scope*: the possible uses of a transformation system.
- *transformation paradigm*: the computational and architectural models of a transformation system.
- *conceptual model*: the choice of the model of the types of elements of programs to be manipulated.
- *programmatic model*: the abstractions of the transformer implementation language that are used to make the represented program elements concrete.

This categorization is used as the structure of this article: one section is dedicated to the description of Jabyce and its comparison with other systems, in each dimension.

3. Scope

This section lists the Java bytecode transformation systems that are considered in this article, and compares those systems along the scope dimension. The discussion about Jabyce and other transformation systems is restricted to systems that are close to Jabyce, i.e. systems that transform Java programs and that are portable, i.e. that perform transformations without requiring modifying the Java Virtual Machine.

3.1. Considered Transformation Systems

ASM[2] [3, 26] is developed in the same team as Jabyce in France Telecom. It is a Java-based framework for the implementation of very efficient transformations of compiled Java classes. JOIE [6] (Java Object Instrumentation Environment), JMangler [19] and Javassist [5] are frameworks for the transformation of compiled Java classes, that also offer abstractions to implement transformers in Java. Those three systems are very similar. They allow for load-time transformations, i.e. the compiled classes of a transformed program are transformed when the program is loaded. The first academic works on Java bytecode transformation were BIT (Bytecode Instrumentation Toolkit) [20] and BCA (Binary Component Adaptation) [18]. However, these two systems are no more maintained, so we do not consider them in this article, as JOIE, JMangler and Javassist offer similar features.
jclasslib [16], Serp [30] and BCEL (ByteCode Engineering Library) [4] are Java libraries that offer representations of compiled Java classes as graphs of Java objects, and that can be manipulated. Unlike the transformation systems described above, these libraries do not offer abstractions for the implementation of transformers, and therefore are not transformation systems. But such libraries are widely used to implement transformation systems. For instance, BCEL is used in the implementation of JMangler. Serp and BCEL are therefore evaluated in the rest of this article, for the conceptual and programmatic model they offer for the representation of compiled Java programs. jclasslib offers a model that is too basic to be used in practice to implement complex transformers. jclasslib is therefore not considered in the rest of this article.

3.2. Comparison

Since JOIE, JMangler and Javassist are designed for load-time transformations of Java classes, they mainly target only certain kinds of rephrasing transformations, such as optimization, normalization and code weaving. While JOIE allows for any kinds of transformations of Java classes, including removal of methods and fields, JMangler and Javassist restrict the transformations that can be performed, in order to preserve binary compatibility of transformed programs, i.e. they allow only for transformations that do not require transforming also client classes.
Jabyce is intended to support any kinds of transformations of compiled Java programs. Since transformers implemented with Jabyce transform compiled Java programs to produce only compiled Java programs, Jabyce alone does not support translation transformations. However one can

[2] Free software available at http://www.objectweb.org/asm/

combine Jabyce and a similar transformation system to implement translations, as described in section 9. That way, Jabyce can be used to implement any translation and rephrasing transformers of compiled Java programs. ASM has a similar scope as Jabyce. However, unlike Jabyce, the design process of ASM is not easily reproducible, which makes it difficult to design a similar transformation system that can be combined with ASM to implement translations. ASM is therefore limited to rephrasing transformation, like JOIE, JMangler and Javassist.
Since all the considered transformation systems, including Jabyce, manipulate only compiled Java programs, which are generally not directly manipulated by human users, transformations related with program design are not considered because these generally deal with the source code of programs. However, one could still implement transformers that interact with users.
Manipulating compiled Java programs allows performing transformations either statically, i.e. before the transform program is executed, or dynamically, i.e. when the program is loaded just before it is executed. Performing transformations at program load-time is possible in Java, thanks to characteristics of the JVM [21]. The compiled classes forming a Java program are dynamically loaded by the JVM when required, by special objects called *class loaders*. Such objects return to the JVM the sequences of bytes that form the compiled classes, given their name. The JVM then executes the program by interpreting or compiling and executing these classes. It is possible to implement custom class loaders, including class loaders that perform transformations on the loaded classes, for example using Jabyce transformers. All of the considered transformation systems either directly offer such mechanism, or can be used to implement it easily.

4. Transformation Paradigm

The main point in this dimension is the distinction between two categories of paradigms offered to implement transformers: *declarative rewrite rules*, and *procedural rules* that are specified directly in an imperative programming language [27]. The choice between declarative and procedural rules is a trade-off between simplicity and generality. Declarative rule languages make it simple to define transformations, but implicitly restrict the range of the transformations that can be defined. On the contrary, procedural rules offer the widest range of transformations, at the cost of an increase of the complexity of implementation. Since one of our objectives is to allow for the implementation of any transformations, procedural rules are the preferred choice in our context. This latter choice

has been made in ASM, JOIE, JMangler, Javassist and Jabyce: transformers are implemented in the Java programming language.
The rest of this section mainly deals with the abstractions offered by transformation systems to implement transformers as procedural rules, and to compose them.

4.1. Global Architecture

The global architecture of transformer configurations is the same for all transformation systems. This section introduces some concepts and a terminology to describe such global architecture. Let's first consider, for the sake of simplicity, a transformer configuration that does not perform any transformation. We identify two main components in the configuration, as illustrated in Fig. 2:

- a *deserializer* that analyses the programs to transform, in a possibly unstructured form that we call a *serialized form*, such as a stream of characters, and produces representations of the programs that it transmits to the serializer.
- a *serializer* that receives the program representations to produce corresponding serialized forms of the programs.

In the literature, deserializers and serializers are commonly called *front-ends* and *back-ends*, or *parsers* and *pretty-printers* [7, 22]. We call an *intermediate representation* the form of a program representation that is communicated between components. The model of the intermediate representation of programs corresponds to the *conceptual model* and *programmatic model* dimensions, described in sections 5 and 6.

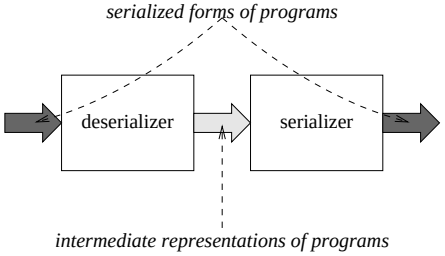

Fig. 2. A deserializer and a serializer

A transformation system should offer a means to support the implementation of any serializer and deserializer, to support any serialized forms of compiled Java programs. This is the case with Jabyce and ASM. In Jabyce, currently one serializer and one deserializer are implemented to support compiled Java classes in the form of sequences of bytes that correspond exactly to the content of .class files. The ASM

deserializer and serializer classes are used internally to implement the deserializer and the serializer implementations provided in Jabyce, that support sequences of bytes as a serialized form. JMangler can also support any serialized form for which a serializer and deserializer can be implemented to serialize and deserialize BCEL object graphs. BCEL provides serialization and deserialization for sequences of bytes. In contrast, JOIE and Javassist support only compiled classes in the form of sequences of bytes.

We propose to describe transformers as *decorators of a serializer*, by referring to the *Decorator* design pattern [10], as illustrated in Fig. 3. Such decorator receives all the program representations, and forwards them to the serializer. When forwarding program representations, it can modify them, e.g. add or remove elements in them, leading to the representation of a transformed program. Several transformers can easily form a chain, one transformer decorating another one. At that level of abstraction, transformers are called *operations* in Jabyce.

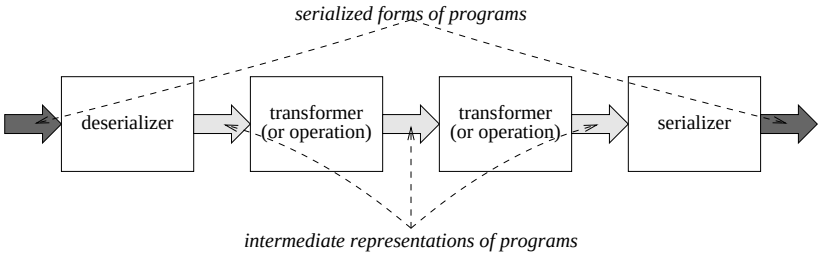

Fig. 3. Transformers as decorators of serializers

All the considered Java bytecode transformation systems considered here match with this general description. However, one limitation of Javassist is that Java classes are deserialized before each transformer is run, and serialized after each transformer is run. Actually, Javassist offers a means to construct chains of "pools" of classes to be loaded by the JVM. A transformer can be associated with each pool, and run when a class is loaded from that pool. However, a Javassist class pool stores the classes as arrays of bytes, that must therefore be (de)serialized to be transformed by the associated transformer. This characteristic of Javassist violates our objective of an efficient composition of transformers.

At this point, it is necessary to make choices about the implementation of transformers and of their infra-structural environment, including the flexibility points of the system (what can be changed in the system?), the identified abstractions, and the implementation model. We consider that all these characteristics form the *transformation paradigm* offered by a transformation system. The next subsections describe the Jabyce transformation paradigm, and compare it to those of the other systems.

4.2. Principles for the Design of Jabyce

The design of transformation systems developed using an object-oriented programming language, such as Java in the case of Jabyce, boils down to object-oriented system design. When designing Jabyce we have applied the following well-known object-oriented design principles to express the desired properties of transformation systems, in order to attain our objectives to make transformer implementations easily reusable and composable.

The Open-Closed Principle [24]

Modules should be both open and closed.

A module is said to be *open* if it is possible to extend it. A module is said to be *closed* if its description is stable and well defined, so that it can be used safely by other modules. Openness is necessary to adapt a system to changes that were not planned initially. Closure is also necessary, because if the description of a module is unstable or not well defined, a change in its implementation or an extension of it may imply modifications in the modules that depend on it. This makes the whole system potentially unstable.
In a transformation system we consider that the most important modules are the transformer implementations, since they are the most complex pieces of code. Since we describe a transformer as a decorator of a serializer, the description common to all transformers based on a given transformation system, is the specification of the program representations that it can receive. This specification is defined precisely, once, and for all the transformer implementations based on the transformation system, making these modules *closed*. Any transformer implementation can be developed as long as it respects this specification, making the system *open* to the use of new transformer implementations.
All the considered transformation systems apply this principle. However, the challenge is to provide the most precise specification of the programs that can be represented, in order to maximize the "closure" of the transformer implementations. An ambiguous specification of the program representation model would make transformers incompatible. An approach for the formal specification and on-line validation of program representations is applied in Jabyce, but it is out of the scope of this article. The other considered systems do not use such formal specification approach.

The Common Closure Principle [22]

The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

If we abstract the concepts of class and package in this definition, this principle means that the implementation parts of a program that are modified together should be packaged and reused together.
In the context of program transformation, there can exist *causal dependencies* between transformations, that are due to *semantic relations between the elements* of a program. For example, the renaming of a method in a Java class implies to transform all the method call instructions that refer to that method. According to the Common Closure Principle, the corresponding two transformer implementations must be packaged and reused together.
No existing transformation system enforces such principle. However, Jabyce makes it easy to do this by ensuring *structurally* (through subtyping relations between the Java classes and interfaces of the Jabyce framework, cf. section 9) that a transformer that transforms program elements of type "A" also transforms program elements of type "B" that depend semantically on type "A" elements. These semantic dependencies are expressed between element types. For example, Jabyce ensures that a transformer implementation that transforms methods also implements transformations of method call instructions. Therefore a transformer implementation in Jabyce is well encapsulated and can be reused easily. We claim that this is a prominent feature of Jabyce.

The Interface Segregation Principle [23]

Clients should not be forced to depend upon interfaces that they do not use.

In the context of transformer implementation design, we interpret this principle as follows: transformation implementations should not be forced to implement computations that are not related to the program elements that they transform. For instance, a transformer that only renames the methods in a Java program should not need to manipulate the fields in that program. In declarative rewrite rules systems, this is achieved in a simple way because a rule is declared to rewrite elements of only certain types. For instance, in Stratego [32], a declarative rewrite rule that transforms methods does not deal with fields.
As described in sections 5 and 7.1, a prominent feature of Jabyce is a representation of elements of programs as interactions which, in the case of Jabyce, allows transformers to manipulate only the representations of

the program elements that it transforms. The other considered transformation systems do not apply this principle.

4.3. Jabyce Transformers as Fractal Components

To exhibit these desired properties of an object-oriented transformation system, and to apply the design principles discussed above, it is necessary to choose an appropriate platform to implement this system. In all the considered transformation systems except Jabyce, that platform is limited to the Java programming language and the JVM. In Jabyce, we have chosen a more powerful component model, to ease the application of the design principles and the implementation and composition of transformers.

The Fractal Component Model

To implement transformers, deserializers and serializers in Jabyce, we choose the *Fractal component model* [1, 2, 27]. It is an extension of object models such as RM-ODP [14] or Java, that exhibits several properties that are interesting here:

- *dynamism*: components are runtime entities: they can be manipulated and (re)configured at runtime.
- *encapsulation*: components interact, only through well defined access points called *interfaces*. An explicit *binding* can link two interfaces. Bindings are arbitrary communication paths.
- *nested composition*: components can contain components, recursively. Recursion ends up with *primitive* components which have an empty content and directly encapsulate plain objects. Components that do have a content, i.e. that contain sub-components, are called *composite* components.
- *control*: components transparently provide *introspective* and *intercessive* capabilities (i.e. to access and modify metadata, respectively) to exercise arbitrary reflexive control over their execution. Fractal provides standard controls: adding or removing of sub-components in composite components, starting and stopping components, bindings reconfiguration, etc.

Fractal components expose interfaces to interact with other components. *Server interfaces* specify the functionalities offered to other components, while *client interfaces* specify the functionalities required for the components to run. In the projection of Fractal into the Java language, component interfaces are specified by Java interfaces that define method

signatures, i.e. that specify the possible *interactions* between components. A *binding* is a directed link from a client interface to a server interface, so that the components can interact. Only interfaces with compatible types can be bound. The minimum requirement of the Fractal type system is that the Java interface that specifies the server interface must be or extend the Java interface that specifies the client interface. Components and bindings are created and manipulated at runtime. Special control interfaces allow for the control of the components lifecycle, bindings and content (subcomponents of a composite component).

Application to the Design of Transformers in Jabyce

In Jabyce, we specify that transformer configurations are built exclusively by instantiating and composing components, dynamically. By components, we mean deserializers, serializers, and transformers. These components are bound using Fractal bindings, to allow for interactions between these components that communicate intermediate representations of programs to transform. The details of the architecture of such components, including the interfaces they offer, are described in section 7.

4.4. Transformation Strategy

Generally, we consider that transformation systems, including Jabyce, apply the Hollywood Principle [33] ("don't call us; we'll call you"), also called the Inversion of Control Principle [15]. In these systems, additional components, distinct from the transformers and (de)serializers in a transformer configuration, *control* the components and control the interactions between them. This separation makes the system more flexible, because the way the transformers are controlled can be modified without modifying the transformer implementations. Adding information into transformer implementations about how to apply them would reduce their reusability.
The main control is the application of a *strategy*, defined in [32] as an algorithm for choosing a path in the triggering of transformations in the transformed programs. For example, applying a bottom-up strategy means transforming first the instructions, then the method signatures, then the class signatures. Applying a top-down strategy means transforming first the class signatures, then elements in them, recursively. In the Stratego declarative rewrite rule language [32], users can define their own strategies, that are either dependent on specific rules, or independent such as the standard strategies provided by Stratego, but rules are never dependent on strategies.

In Jabyce, operation components in a chain are composed together using user-specified component bindings. The manipulation of such bindings, outside of the components, is allowed by the standard control interfaces provided by the implementations of the Fractal model. The transformation strategy is therefore expressed in Jabyce as a set of component bindings. The triggering of the transformers is determined by the order by which they "decorate" each other in a chain. The flexibility offered by such mechanism lets users specify arbitrary chains of transformers, without modifying the implementation of transformers. It also allows for more complex configurations with multiple deserializers and serializers. This is an advantage of Jabyce over most other systems: it does not only allow chain configurations, but offers immediately almost arbitrary strategies.

4.5. Comparison

In the other Java bytecode transformation systems, transformers are Java objects that implement Java interfaces that are specific to the transformation system. The transformation paradigm is therefore the Java object model, and abstractions such as Java interfaces.
JMangler defines the Java interface CodeTransformerComponent, to be implemented by method implementations transformers, and the Java interface InterfaceTransformerComponent, to be implemented by transformers of signatures of Java interfaces and classes and of members defined in them. Those interfaces define methods that accept intermediate representations of programs to transform. JMangler limits the range of transformers that can be implemented, in order to maintain the *binary compatibility* of the transformed Java classes. Any method implementation transformer can be implemented, but transformations of class and interface signatures are limited to adding fields, methods and inheritance relations, etc. For instance, it is impossible to rename a method in JMangler. This is opposed to our objective to offer the widest range of implementable transformers. A similar pattern is used in JOIE (ClassTransformer interface), Javassist (Translator interface) and ASM (ClassVisitor and CodeVisitor interfaces). However, these systems do not limit the range of implementable transformers like JMangler. The advantage of the use of the Fractal component model in Jabyce, is to allow for a better design and maintenance of complex transformer implementations, compared to these systems.
It must be noted that the Decorator design pattern, described in the beginning of this section, is applied literally only in Javassist, ASM and Jabyce. More precisely, in these systems, a transformation strategy is directly expressed as links between transformers in a configuration. The links are Fractal component bindings in the case of Jabyce, direct Java object references between transformers in the case of ASM, and direct

Java object references between compiled class pools in Javassist. In JMangler and JOIE, a strategy is implemented in a separate object that schedules the executions of transformers. In JMangler two strategies are combined. The program interface transformations are applied in a non-deterministic order until the transform program reaches a fixed point, thus the transformations are applied always with the same strategy, and any order produces the same transformed program. The code transformers are run in a user-specified sequential order, i.e. with a user-specified strategy. In JOIE, the transformers are run in a user-specified sequential order, on one whole class at a time.
Jabyce already allows for transformer configurations that are more complex than simple chains, for instance multiple transformer chains that are bound to a single serializer component at their end. In other systems, arbitrary configurations could be implemented on top of the systems, by implementing transformers that control other transformers. For example, in JMangler, one can use an object which class implements CodeTransformerComponent and that interacts with other transformers, to build a complex configuration. However, there is no generally defined mechanism to do so, and such class must be implemented "by hand". It is therefore more difficult than in Jabyce.

5. Transformed Programs Programmatic Model

This dimension concerns the abstractions offered to implement manipulations of elements of programs to be transformed. In [32] it is assumed implicitly that programs are either represented as *graphs* of terms, i.e. as graphs of Java objects in our context. This is the case for all considered transformation systems, except Jabyce and ASM. In Jabyce and ASM, we propose to represent program elements as *interactions* between parts of transformers, and not as Java objects.

5.1. Interaction Sequences vs. Object Graphs

Our proposal is based on an analogy between programs and semi-structured documents. As described in [28], there are two ways to represent XML and SGML documents in Java programs that manipulate them:

- using a graph-based API such as DOM (Document Object Model) [12], which represents a document as a graph of Java objects;
- using an interaction-based API such as SAX (Simple API for XML) [29], which represents a document as a sequence of method calls that notify

for instance the beginning or the end of the parsing of an XML element in the document.

In a DOM object-graph approach it is possible to transform a document by interacting with the object graph, to add new node objects, remove sub-graphs, modify node objects, etc. The edges in such graphs are represented as direct references between Java objects. These edges model not only the composition relations between XML elements, but may also model XML element references.

In a SAX interaction-sequence approach, transformations would be performed in a transformer by producing a new document based on the received sequence of interactions and by performing *decorations* of those interactions. The new document is produced by calling the same methods as those implemented by the transformer. For example, it is possible to remove an XML element simply by ignoring the interactions that notify the start and end of the parsing of this XML element. The other elements are kept back by reproducing them when receiving the corresponding interactions. For example, Fig. 4 shows the source code of a SAX content handler class that transforms parsed XML documents to add an XML element <someNewElement/> into each parsed XML element.

```
public class XMLLogicalModelParser
      implements ContentHandler {

   public ContentHandler delegate;

   public void startElement(String namespaceURI,
         String localName, String qualifiedName,
         Attributes attributes)
         throws SAXException {

      delegate.startElement(namespaceURI,
         localName, qualifiedName, attributes);

      delegate.startElement(…, "someNewElement",…);
      delegate.endElement(…, "someNewElement",…);

   }
   public void endElement(String namespaceURI,
         String localName, String qualifiedName)
         throws SAXException {
      delegate.startElement(namespaceURI,
         localName, qualifiedName);
   }
   public void characters(char[] chars, int offset,
         int length) throws SAXException {
```

```
        delegate.characters(chars, offset, length);
    }
    /*...*/
}
```

Fig. 4. Source code of a SAX content handler that transforms XML documents

Any of these two programmatic models can be used to transform programs. Both models fit well with the general architecture proposed in section 4.1, and based on the Decorator design pattern. In the example above, which illustrates the interaction-sequence representation of XML documents, the transformer object is a decorator of another SAX ContentHandler object, which reference is given in the delegate field. The links between several such transformer objects, i.e. the values of their delegate fields, forms an XML document transformation strategy.

5.2. Performance vs. Simplicity

The choice of a programmatic model for the representation of program elements is motivated by two contradictory characteristics of transformation systems: transformation performance and implementation simplicity. Globally transformers consume two resources: memory and CPU time. High performance is achieved by minimizing this consumption. Using an object graph representation, memory is consumed to represent the objects that represent program elements, and time is consumed to create these objects, proportionally to the size of the programs to transform. Therefore the size of the programs that can be transformed is limited by the available memory size. Another drawback is that when using a transformation system one generally needs not to manipulate all the objects in the graph, leading to excessive memory and time consumption. It is possible to implement an incremental construction of the graph of objects into memory, like in BCEL and JOIE, but at the cost of an increase of the complexity of the system. Using an interaction sequence representation, it is not necessary to store a complete representation of programs into memory. Memory is used only for the interactions stack and to store the minimal state necessary to perform the transformations. CPU time is consumed only to perform transformations and the interactions that represent program elements; no CPU time is consumed to create unused objects. An interaction sequence representation is therefore the preferred programmatic model when considering resource consumption.
On the other hand, complex transformers are easier to implement using an object graph representation of programs. For example, when a transformation consists in adding a method in a Java class, it is necessary

to make sure that no method with the same name and signature is already defined in that class. Such verification is easy to perform when it is possible to visit at that time the whole representation of the transformed Java class. Using an interaction sequence representation, it would be necessary to maintain "by hand" (i.e. apart from the transformation system in use) as a state the list of all the method names and signatures represented by the interactions, and to wait until all the interactions for the class have happened, to perform the verification. An object graph representation is therefore the preferred programmatic model when implementing transformers that have a state.
As a conclusion, the choice of a programmatic model for the representation of program elements is the result of a *trade-off* between transformation performance and implementation simplicity. This choice is made at transformation system design time.

5.3. Comparison

The purpose of Jabyce is primarily to implement transformers that weave code into programs, possibly at program load-time. Transformation performance is a major concern in this case, in order to perform transformations efficiently dynamically on large programs or in constrained environments. Therefore we choose to represent programs as interactions in Jabyce. The range of transformers that we consider implementing can be implemented simply using Jabyce, as demonstrated by the example transformer implementation presented in section 8.1. Their low complexity does not require using an object graph representation of programs. However, when a transformer needs to keep a state, it must be explicitly implemented by the developer. The same choice has been made in ASM [3].
The other systems (JOIE, JMangler and Javassist) use a Java object graph representation of the manipulated program elements. To our knowledge, no other program transformation system, for other languages, uses an interaction representation of programs. This is a prominent feature of our systems Jabyce and ASM.

6. Transformed Programs Conceptual Model

This dimension deals with the identification of the types of elements that can be distinguished in a program representation manipulated by a transformer. In the context of compiled Java programs transformation, one must identify the abstractions in a Java class that correspond to the transformed elements, i.e. one must define the *granules of transformation*,

for instance “class”, “method”, “formal argument”, etc. Using a Java object graph representation, these abstractions correspond to the possible types of node Java objects. Using an interaction sequence representation, these element types correspond to the possible kinds of interactions. For instance, a “formal argument” element can be represented by a Java object of type FormalArgument using an object graph representation, or by a call to a method formalArgument() using an interaction representation. The choice of a conceptual model concerns the two following aspects:

- *granularity of transformations*: one must make a trade-off between the size and complexity of the representations of program elements, and the total number of transformations necessary to transform a complete program.
- *abstraction of transformed elements*: which details of the format of the serialized forms of programs does the system hide?

The granularity has an impact on the performance and simplicity of implementation of transformers. For example, a coarse-grained model of a Java class may define only three complex abstractions: “class”, “method” and “field”. A method representation may contain the instructions of that method as an array of bytes. Therefore each transformer that transforms instructions must decode these bytes itself, which costs a lot and is complex to implement. In very fine-grained models, such as BCEL, there are a lot of fine-grain abstractions, e.g. an object type is defined for each instruction type. The decoding of the instructions is performed by the system, making the implementation of instruction transformers simple, but memory and time is consumed to create an object for each instruction even if not all instructions are manipulated. The Javassist framework has been designed to implement only structural transformations. Its model of a Java class is therefore very coarse-grained, and offers only a few element types, similar to that of a class source file, including fields, methods, constructors, inheritance relations, access modifiers, formal arguments, “new” instructions and field access instructions. All other details are hidden. Therefore we claim that the choice of a granularity is motivated by the desired transformations, i.e. it is motivated by the scope of the system, and the model is chosen to optimize transformations in that scope.

6.1. General Issues About Modeling Java Classes

In Jabyce, Java programs are transformed one class at a time, because we want to be able to transform programs at load time, and classes are loaded one at a time by the virtual machine. The same choice has been made in JOIE, Javassist and ASM for the same reason. In JMangler, transformers can manipulate several classes at a time, i.e. they can manipulate whole programs. JOIE is the only transformation system which transformers

cannot produce several classes when transforming a class, i.e. the transformation of a class must result in exactly one transformed class.
The format of a compiled Java class, as specified in the JVM specification [21], defines several abstractions that are difficult to deal with when performing transformations. Some of these abstractions are presented thereafter, to compare how Serp, BCEL (and therefore JMangler), JOIE, Javassist, ASM and Jabyce deal with them. Javassist offers two models for manipulating compiled Java classes: a high-level model that offers abstractions similar to Java source-level element, and a low-level model. However, the low-level model does not hide enough details of the format of compiled Java classes to be used in practice in complex transformers. For instance, the decoding of the bytecode instructions must be performed by each transformer. In the rest of this section we therefore consider only the high-level model offered by Javassist.

Constant Pool

The constant pool contains and indexes all the constants that are used in the compiled class: string constants, names of accessed classes, methods and fields, etc. For example, in an instruction that accesses a field, the name and type of the field, and the name of the class that defines it, are indicated in the instruction by the index of a constant that contains the name, in the pool. Therefore changing the name of a field requires adding a string constant containing the new name, into the constant pool of the defining class and of all the classes that access it, and changing the field definition and all the corresponding field instructions, to use the index of the new constant in the constant pool. **JOIE**, **BCEL** and **Serp** make these constant indexes explicit, but provide utility methods to insert new constants into the constant pool and to modify constants. In **Javassist**, **ASM** and **Jabyce**, the constant indexes are hidden, and the constants are directly manipulated by transformers. In Jabyce, for example, a string constant, such as a string pushed by an ldc instruction, is directly manipulated by transformers as a string object. The constant pool of the transformed class is constructed while the class is progressively represented. As a secondary effect, the constant pool is therefore automatically optimized to contain only the constants that are actually used in the transformed class, which is difficult to achieve with BCEL for instance.

Branch Addresses

A method can contain control transfer instructions, that branch to other instructions in the method. The target instructions are specified as offsets,

in number of bytes, to the address of the transfer instructions. Each instruction has a unique address, which is a number of bytes from the start of the method. Therefore, inserting or removing instructions from a method potentially requires to modify the target offsets of control transfer instructions. In **Javassist**, instruction addresses are never manipulated, since transformers can only manipulate source-level abstractions. In **JOIE**, **BCEL** and **Serp**, transfer addresses are represented as direct references between Java objects representing bytecode instructions. In **ASM** and **Jabyce**, instruction addresses are abstracted, by the use of abstract *instruction label* objects. Instruction labels are inserted in a method to "tag" instructions, so that they can be referred to by branch instructions. In all systems, the offsets are automatically calculated.

Local Variables

In a method, each local variable is identified by its integer index and its size (32- or 64-bit). This index is specified in each instruction that loads or stores a local variable. The this pointer and the formal arguments are represented like normal local variables initialized with the passed values, whose indexes are 0 for this, and 1 and higher for the arguments in the same order as in the signature. The normal local variables, that are uninitialized at the beginning of the method, have the next higher indexes. Therefore, adding a formal argument, such as when *adding a hidden software capability* [11] to an argument, requires incrementing the index of all the uninitialized local variables, in the instructions that access them. In **Javassist**, it is not possible de transform method signatures, nor to manipulate local variables. In **JOIE**, **BCEL**, **ASM** and **Serp**, the local variable indexes and sizes are manipulated explicitly. In **Jabyce**, local variables indexes and sizes are abstracted, by the use of FrameSlotIdentifier objects that uniquely identify local variables when representing local variable access instructions. Therefore transformers never directly manipulate local variable indexes. When representing a formal argument, an identifier is given to identify its corresponding local variable, its index is automatically calculated, and the indexes of the already represented normal local variables are automatically modified in the instructions. When representing a normal local variable, an identifier is given to identify it, and it is assigned automatically the next unused index. This design choice makes it very easy to implement transformers that add or remove local variables or formal arguments, using Jabyce.

Instructions

Several instruction types have similar semantics, and therefore require similar transformations. For example, all getstatic, putstatic, getfield and putfield instructions access fields. When transforming fields, it is also necessary to manipulate instructions of these four types. A transformation system should therefore provide information to easily identify the element types that require similar transformations. **Javassist** does not allow for the direct manipulation of instructions. In **BCEL**, a Java class is defined for each instruction type specified in the JVM specification, and abstract classes are inherited from these classes to identify similar instructions. For example, the FieldInstruction abstract class is subclassed by the GETSTATIC, PUTSTATIC, GETFIELD and PUTFIELD classes. BCEL defines more than 180 Java classes to represent instruction nodes in graphs. This design choice makes BCEL the most complex of the considered systems. **Serp** follows a similar approach, by defining an Instruction class that is inherited by 31 classes to represent bytecode instructions, including abstract classes to group semantically similar instruction types like in BCEL. **JOIE** defines an Instruction class, that is inherited by 17 classes to represent instructions. However, not all instructions have a dedicated class to represent them. For instance, the bytecode instruction monitorenter and monitorexit are represented as Instruction objects. The JOIE conceptual model is therefore not homogeneous, which makes it difficult to implement certain transformations of instructions. In **ASM**, the CodeVisitor interface defines 12 "visit" methods for instructions, that correspond to groups of instruction types that have the same structure. In some cases, this structural grouping of instruction types also corresponds to a semantic grouping. For example, the visitFieldInsn method is used to represent instructions that access fields, i.e. instructions that have a similar semantics. These instructions have also the same structure, i.e. they contain an *opcode* (the identifier of the instruction type), the name of the class that defines the field, the name of the field and the type of the field. But in some other cases, a "visit" method concerns instructions that have a different semantics. For example, the visitInsn method is called to represent instructions that contain only an opcode, such as pop instructions that pop values from the stack, and such as i2l instructions that convert integer values into long values. Therefore implementations of this method must deal with semantically dissimilar instructions, and must analyze the given *opcode* to identify the semantics of an instruction. This contradicts the Interface Segregation Principle, as presented it in section 4.2. In **Jabyce**, we choose a model similar to the ASM model, but we refine it to group only semantically similar instruction types. For example, instructions that access fields are FieldAccessInstructions, instructions that manipulate the stack like pop are StackInstructions, and instructions

that perform arithmetic conversions like i2l are ArithmeticTypeConversionInstructions. In addition, instruction *opcodes* are hidden to transformers, so that the decoding of *opcodes* is performed once and for all transformers in a chain, and transformers manipulate only high-level information about instructions, making them easier to implement than with ASM. Jabyce defines 30 instruction kinds, which makes it only slightly more complex than ASM.

Optimized Instruction Forms

Some instruction types have optimized forms. For example, an iload instruction loads an integer value from a local variable, which index is specified in the instruction after its *opcode*. An iload instruction is therefore encoded as two bytes, in a compiled class file. But four optimized forms of this instruction type exist (iload_0, iload_1, iload_2 and iload_3), with distinct *opcodes* and no index in the instructions, that can be used when the local variable index is 0, 1, 2 or 3. These instructions are encoded as only one byte containing the opcode. A transformation system should automatically use the most optimized form of instruction according to the instruction parameters. All systems, including Jabyce, do not offer specific abstractions for the optimized instruction forms, and implicitly produce the most optimized forms of instructions.

Conclusion

As a conclusion, Jabyce offers the most abstract model of a Java class, after Javassist. As opposed to Javassist, it still allows for the implementation of any transformer of compiled Java classes, and the additional abstractions are introduced only to automate the optimization of the generated code, and to simplify the implementation of transformers.

6.2. The Model of a Java Class in Jabyce

In Jabyce the model of a Java class is a graph, which nodes are *element types*, and edges are composition relations. Element types which elements can contain other elements are called *composite* element types. The other types are *primitive*. A top-level composite element is a Class. A Class can contain InterfaceInheritanceDeclaration and InnerClassDeclaration elements, for the declaration of inherited interfaces and inner classes. A Class can contain composite Field elements, which can contain a FieldConstantValue. A Class can contain composite Method elements. A Method can contain ThrownExceptionDeclaration and

MethodFormalArgument elements for the method signature declaration. It can also contain LocalVariable, MethodFormalArgumentLocalVariable and ThisLocalVariable elements for the declaration of local variables, InstructionLabel elements to identify instructions, and instruction elements. Jabyce defines 30 instruction element types, such as ReturnInstruction, LocalVariableAccessInstruction, and FieldAccessInstruction. As a conclusion, Jabyce defines 50 element types to construct Java classes, including three composite element types.
Next section describes how the element types defined above are used to define in Jabyce the possible interactions between transformer components which represent program elements.

7. Putting It All Together

This section describes more thoroughly Jabyce, by combining the two main characteristics of Jabyce introduced in the preceding sections: 1) the use of the Fractal component model, and the expression of transformation strategies as component bindings, and 2) the representation of program elements as interactions between components.

7.1. Transformation Operation Internal Architecture

In Jabyce, transformer configurations are built exclusively by instantiating and composing components dynamically, which are deserializers, serializers, and transformers. The interactions (method calls) between these components represent elements of transformed programs. Between two components, we define that there is one binding for each type of element that makes up a compiled Java class. For instance, one binding is used for interactions that represent fields (element type Field), one is used for interactions that represent method formal arguments (element type MethodFormalArgument), etc. According to the Jabyce conceptual model defined in section 6, each component must therefore offer 50 interfaces, one for the representation of entities of one type. Those bindings are illustrated in Fig. 5. The interfaces define methods, which calls represent each a program element.[3]

[3] Although all the interfaces in Jabyce are called "factory" interfaces, this architecture is not an orthodox application of the Abstract Factory design pattern [10]. That terminology is motivated by the need for a prefix for method names to avoid name clashes. We have arbitrarily chosen the "`create`" prefix, hence "`Factory`" interfaces.

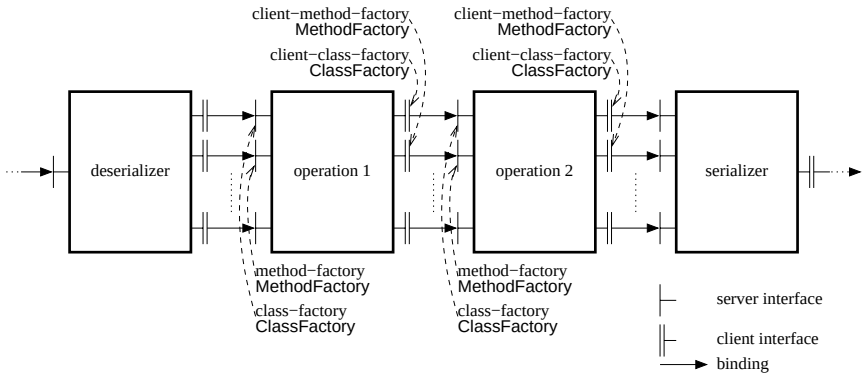

Fig. 5. Operations bound to form a chain

The most basic operation does not perform any transformation, which is achieved simply by not intercepting the interactions that represent program elements through bindings. We design such an operation in Jabyce as a composite component whose server and client interfaces are directly bound, as illustrated in Fig. 6. If the reference implementation of Fractal in Java (named Julia[4]) is used, such an architecture does not add any overhead to the interactions on the bindings, and the binding of such an empty operation in a configuration has no cost, thanks to optimization mechanisms.

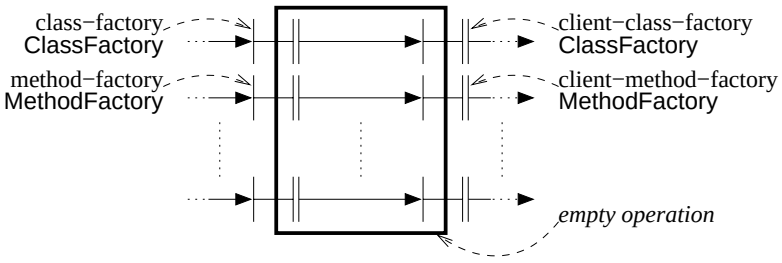

Fig. 6. An operation that performs no transformation

One way to construct operations that actually perform transformations is to add transformers, i.e. sub-components, into such composite component, whose interfaces are bound to the composite component interfaces. Fig. 7 illustrates an operation that contains one transformer that decorates the interactions representing program elements received on one interface, and that uses two client interfaces to represent program elements of two kinds.

[4] Free software available at http://www.objectweb.org/fractal/

The other interfaces are directly bound, i.e. the program elements represented by calling methods on them are not transformed. Such transformer has a small granularity, therefore it would naturally be implemented as a primitive component. From a general point of view, a configuration of transformers therefore concerns two abstraction levels: between deserializers, operations and serializers, and inside operations.

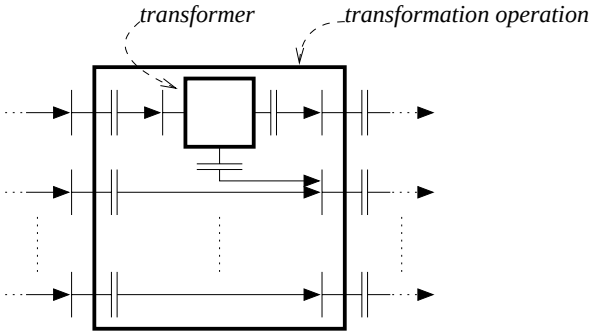

Fig. 7. An operation composed of one transformer

An operation is a kind of transformer that always has the same set of client and server interfaces, i.e. all operations have externally the same form, while an ordinary transformer has only the minimum necessary interfaces. As illustrated in Fig. 5, 6 and 7, operations have several interfaces, each one is used to represent program elements of one kind. For example, one interface is used to represent field access instructions, another one is used to represent constant value push instructions, etc. The list of the element types of the Jabyce conceptual model is presented in section 6. All the operations have the same set of interfaces, on the server and the client side, like deserializers and serializers. Inside operations, transformers offer only the minimum set of interfaces that are needed. Their only server interfaces are for receiving the interactions for the representation of the elements to transform; their only client interfaces are used to represent elements in the resulting program. In transformers, the only computations are therefore performed to transform elements. This is an application of the Interface Segregation Principle, and is an improvement over ASM, which was not designed with interface separation. ASM decorators must implement the "visit" methods for all the element types of a Java class even if some are not transformed. There is therefore always an overhead on the representation of each program element.

Components interact only through interfaces specified by the Jabyce Java interfaces, that are general, abstract and stable, since they do not depend on specific transformers. Therefore, as stated in the Open-Closed Principle, the Jabyce transformation system is both *open*, i.e. a

transformer can be implemented to perform any transformation, and *closed*, i.e. its specification consists of the Jabyce interfaces, which are closed. Therefore a transformer implementation, i.e. a Java class that implements a primitive component and uses the Java interfaces defined in Jabyce, can be easily reused, so we define it as the *granule of reuse*.

7.2. Specification of the Interfaces

For each element type we define a server interface, on each serializer component and each operation. Therefore the Jabyce framework defines 50 Java interfaces: ClassFactory, FieldFactory, LocalVariableFactory, FieldAccessInstructionFactory, etc. Each Java interface defines one method, to represent elements of the corresponding type. For example, the source code of the FieldAccessInstructionFactory interface is partially given in Fig. 8. The arguments specified in the signature are used to parameterize the representation of the instruction. The ObjectType and FieldDescriptor classes are defined in Jabyce to communicate Java types when representing elements. In this method, all the field and type names are specified as objects, not as indexes in the constant pool. This illustrates our choice to hide such details. The given MethodBuildingContext object identifies the method in which the instruction is represented.

```
public interface FieldAccessInstructionFactory {
  /**
   * Creates the instruction that accesses a field
   * with the specified fields.
   *
   * @param compositeMethodBuildingContext
   *   the context for the building of the composite
   *   method that will contain the created
   *   instruction that accesses a field.
   * @param type the type of the target object that
   *   defines the accessed field.
   * @param fieldName the name of the accessed field.
   * @param fieldIsStatic the flag that indicates
   *   that if true, the field is static.
   * @param fieldType the accessed field type.
   * @param loadOrStore the flag that indicates that
   *   if true, the field is loaded; otherwise it is
   *   stored.
   * @pre compositeMethodBuildingContext
   *   ! = null
   * @pre type ! = null
```

```
    * @pre fieldName ! = null
    * @pre fieldType ! = null
    */
   void createFieldAccessInstruction(
      MethodBuildingContext
         compositeMethodBuildingContext,
      ObjectType type, String fieldName,
      boolean fieldIsStatic,
      FieldDescriptor fieldType,
      boolean loadOrStore);
}
```

Figure 8. Source code of the FieldAccessInstructionFactory interface

A *building context object* is used to identify a composite element when representing elements inside it, i.e. Jabyce defines the ClassBuildingContext, FieldBuildingContext and MethodBuildingContext classes. Such objects are returned by the methods which calls represent composite element types. For example, the source code of the MethodFactory interface is partially given in Fig. 9.

```
public interface MethodFactory {

   MethodBuildingContext createMethod(
   ClassBuildingContext compositeClassBuildingContext,
   ...);

 void finishMethodBuilding(
    MethodBuildingContext buildingContext);
}
```

Figure 9. Source code of the MethodFactory interface

Each building context must be explicitly "closed" after it has been used, by calling the corresponding "finish" method in the interface, when all the elements have been represented in the corresponding composite element. For example, when all the elements (instructions, etc.) have been represented inside a method, the method finishMethodBuilding must be called with the building context object.
From a general point of view, Jabyce relies on the equality of object references between the context object returned when representing a composite element such as a method, and the context object passed when representing an element inside it, as a general mechanism to *represent composition relations* between program elements. More generally, Jabyce uses that mechanism to represent the edges of the conceptual graph that models a Java class.

In Fig. 10 is given the source code that initiates the interactions with a serializer, to represent one class that contains one method. These interactions are equivalent to the interactions initiated by a deserializer that analyzes the corresponding existing class.

```
// create a class:
ClassBuildingContext classBC =
   classFactory.createClass(...);

// create a method:
MethodBuildingContext methodBC =
   methodFactory.createMethod(classBC, ...);

// create a formal argument:
formalArgumentFactory.createFormalArgument(
   methodBC, ...);

// create instructions:
// ...
fieldAccessInstructionFactory
   .createFieldAccessInstruction(methodBC, ...);

// end the building:
methodFactory.finishMethodBuilding(methodBC);
classFactory.finishClassBuilding(classBC);
```

Figure 10. Source code for the creation of a class

7.3. Jabyce: a Transformation Framework

The Jabyce transformation system is designed as a transformation framework. A framework is defined as "a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact" [17], and is described to "bind certain choices about state partitioning and control flow; the user completes or extends the framework to produce an actual application" [8]. The abstract reusable parts that make up Jabyce are the Java interfaces that are the specifications of component interfaces, and abstract classes that help implementing transformers. It comes as well with a set of classes that are predefined general-purpose transformers. It is also a specification of how the components should be bound together, and how they should interact. The transformer configurations are defined by users so that they can express transformation strategies according to their context. The Jabyce

framework can be extended, by implementing new transformers, new deserializer and serializer components, and by specifying new strategies as transformer configurations.

8. Experiments

8.1. An Example Transformer

This section presents the design of a Jabyce transformer that inserts code at the beginning of the implementation of each method of each class, to print a trace message into the standard output. This transformer is similar to the example XML document transformer presented in section 5.1. For example, the result of the transformation of the compiled class which source code is given in Fig. 11 is a compiled class which is equivalent to the compiled class whose source code is given in Fig. 12. The inserted trace printing code is the sequence of bytecode instructions given in Fig. 13.

```
package some.package;
public class HelloWorld {
  public void printHello() {
    System.out.println("hello");
  }
  public void printWorld() {
    System.out.println("world");
  }
  public static void main(String argv[]) {
    HelloWorld obj = new HelloWorld();
    obj.printHello();
    obj.printWorld();
  }
}
```

Figure 11. Source code of the original example class to transform

```
package some.package;
public class HelloWorld {
  public void printHello() {
    java.lang.System.out.println(
"called printHello in class some.package.HelloWorld");
    System.out.println("hello");
  }
  public void printWorld() {
    java.lang.System.out.println(
"called printWorld in class some.package.HelloWorld");
    System.out.println("world");
  }
  public static void main(String argv[]) {
    java.lang.System.out.println(
"called main in class some.package.HelloWorld");
    HelloWorld obj = new HelloWorld();
    obj.printHello();
    obj.printWorld();
  }
}
```

Figure 12. Equivalent source code of the transformed example class

1. getstatic: push the object reference stored in the static field out defined in class java.lang.System, of type java.io.PrintStream;
2. ldc: push the message string to print, identified by an index of the string in the constant pool;
3. invokevirtual: call the println(String) method defined in java.io.PrintStream, with the target object and argument that have been pushed.

Figure 13. Bytecode instructions inserted for trace printing

The corresponding transformer transforms methods. Therefore it exposes one server interface of type MethodFactory. It must be able to represent methods (to reproduce the representation of the transformed methods), field access instructions (getstatic), string constant push instructions (ldc) and method call instructions (invokevirtual). Therefore it exposes four client interfaces of type MethodFactory, FieldAccessInstructionFactory, StringOrNullConstantPushInstructionFactory and MethodCallInstructionFactory. It is illustrated in Fig. 14. It can be integrated into an operation, as illustrated in Fig. 15, by binding its interfaces to the interfaces of the operation. Only the server MethodFactory interface is bound to the transformer component, to transform the method representations, while the others are directly

bound. The client interfaces of the transformer are bound to the corresponding client interfaces of the operation. To perform transformations, a simple configuration is a pipe-line composed of one deserializer, the operation, and a serializer bound together.

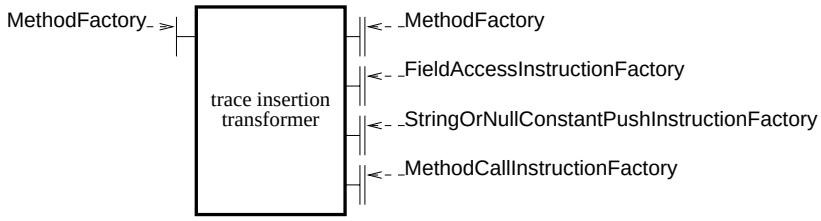

Figure 14. Trace insertion transformer

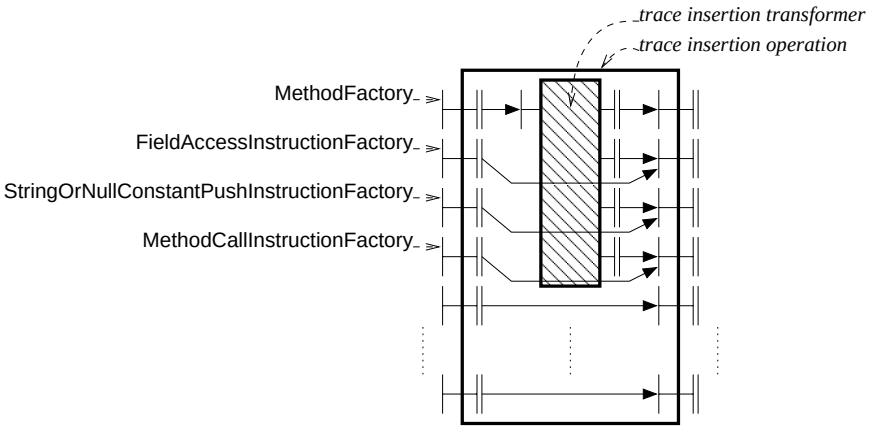

Figure 15. Trace insertion operation

In this example, the transformer is a primitive component. Therefore its implementation is a Java class, as specified in the Fractal framework. The source code of this class is partially given in Fig. 16. The BindingController interface is defined in the Fractal component framework, to define callback methods used by the framework to give to a component the references to the components bound to its client interfaces. Implementing this interface is the only constraint imposed by the Fractal framework.

```
public class TraceInterceptor implements
   MethodFactory, BindingController {

 /* references to components bound
   to client interfaces */
 protected MethodFactory boundMethodFactory;
 protected FieldAccessInstructionFactory
   boundFieldAccessInstructionFactory;
 protected StringOrNullConstantPushInstructionFactory
   boundStringOrNullConstantPushInstructionFactory;
 protected MethodCallInstructionFactory
   boundMethodCallInstructionFactory;

 /* defined inUserBindingController, to query,
   set and unset the client component references */
 public String[] listFc() {/*…*/}
 public Object lookupFc(String clientItfName) {/*…*/}
 public void bindFc(String clientItfName,
   Object serverItf) {/*…*/}
 public void unbindFc(String clientItfName) {/*…*/}

 /* defined in MethodFactory, to represent methods */
 public MethodBuildingContext createMethod(
   ClassBuildingContext compositeClassBuildingContext,
   String name, MethodReturnDescriptor retType,
   boolean isPublic, boolean isPrivate,
   boolean isProtected, boolean isStatic,
   boolean isFinal, boolean isSynchronized,
   boolean isNative, boolean isAbstract,
   boolean isStrict, boolean isSynthetic,
   boolean isDeprecated) {

   /* re-represent the method using
     the bound factory component */
   MethodBuildingContext methodBuildingContext =
     boundMethodFactory.createMethod(
       compositeClassBuildingContext, name, retType,
       isPublic, isPrivate, isProtected, isStatic,
       isFinal, isSynchronized, isNative, isAbstract,
       isStrict, isSynthetic, isDeprecated);

   /* do not transform initializers */
```

```
    if (name.equals("<init>")) {
      return methodBuildingContext;
    }
    String className =
      boundClassFactory.getBuiltClassClassType(
        compositeClassBuildingContext).toString();

    /* compose statically the trace message */
    String messageToPrint = "called " + name
      + " in class " + className;

    /* represent the instructions for
      System.out.println("called…");
      just at the beginning of the method */
    boundFieldAccessInstructionFactory
      .createFieldAccessInstruction(
        methodBuildingContext,
        new ObjectType("java.lang.System", false),
        "out", true,
        new ObjectType("java.io.PrintStream", false),
        true);
    boundStringOrNullConstantPushInstructionFactory
      .createStringOrNullConstantPushInstruction(
        methodBuildingContext, messageToPrint);
    boundMethodCallInstructionFactory
      .createMethodCallInstruction(
        methodBuildingContext,
        new ObjectType("java.io.PrintStream", false),
        false, "println", false, false, VoidType.VOID,
        new FieldDescriptor[] {
          new ObjectType("java.lang.String", false)});
    return methodBuildingContext;
  }
  public void finishMethodBuilding(
    MethodBuildingContext buildingContext) {
    /* finish the method represented
      with createMethod(...), when all instructions
      have been created in its context */
    boundMethodFactory
      .finishMethodBuilding(buildingContext);
  }
}
```

Figure 16. Source code of the transformer Java class

This example implementation of a transformer illustrates our choice to prefer high performance, by representing program elements as sequences of interactions, over implementation simplicity. This trade-off is discussed in section 5.2. This example shows that simple transformations are easier to implement when representing program elements as graphs of objects. However, our other experiments have also demonstrated that complex transformations are equally difficult to implement using either object graphs or interaction sequences. Next section gives some execution time measurements for the example transformer above.

8.2. Measurements

This section compares the time consumption of the execution of the Jabyce transformer described above, with the same transformer implemented using ASM, and another implemented directly using BCEL. The measured execution times comprise the transformation time and the .class files deserialization and serialization time, and exclude the file access time and the configuration initialization time. Transformations are performed for all the 8251 classes of the Blackdown Java 2 SDK 1.4. The transformation program is executed using that JVM on an unloaded Pentium III 1GHz PC running GNU/Linux. The measured execution real (wall clock) times are indicated in Table 1, as the means of 10 measures.

Table 1. Execution time measurements

	ASM	BCEL	Jabyce
Total time	31380 ms	92444 ms	**45301 ms**
Time / class	**3.8 ms**	**11.2 ms**	**5.49 ms**

According to our measures, the Jabyce transformations are performed in only 1.44 times the time of the ASM transformations. This additional execution time is mainly due to our choice of a more abstract model of a Java class, as described in section 6, which implies for example more computations for the local variables. However, we have measured no overhead due to the Fractal component model, thanks to optimizations implemented in the Julia reference implementation of Fractal.

The Jabyce transformations are performed twice as fast as the BCEL transformations. This confirms our choice to represent programs as interaction sequences, in ASM and Jabyce, instead of representing programs as graphs of objects as in BCEL, to reach our objective of high performance program transformations. This choice is discussed in section 5.2.

As a conclusion, from the point of view of transformation performance, these measures validate the two main decisions that we make in the

design of Jabyce, i.e. the use of a general component model that offers a high level of control without reducing performance, and the representation of program elements as interactions.
A Jabyce operation is typically a composite Fractal component that has a sparse internal architecture, i.e. in which most server interfaces are directly bound to client interfaces. We therefore predict that the composition of a high number of operations in a chain is more efficient using Jabyce than using ASM, where each transformer must intercept all interactions that represent program elements. A perspective is to perform measurements using such transformer configurations to confirm that prediction.

9. Generalization of the Approach

The design process of a transformation framework like Jabyce is reproducible. For instance, one may need to design and implement a similar transformation framework, to transform LaTeX documents, which has the same characteristics, i.e. 1) that uses the Fractal component model and 2) that represents documents as interactions between components. The only difference between that framework and the Jabyce framework is the conceptual model of the elements to transform, which is expressed as the set of component interfaces as described in section 7.2. In order to automate the design and implementation of such framework, we provide the Jaidee[5] tool. Jaidee takes as input a formal description of the elements to be transformed, in the form of an XML file, and generates the source code of most of the Java classes and interfaces of a transformation framework. These are: the component interfaces for the representation of program entities, similar to the interfaces presented in section 7.2, the building context interfaces and base classes, and abstract classes that help implement transformer components. Jabyce is the first transformation framework generated by Jaidee: 165 source files are completely automatically generated with Jaidee, out of the 226 source files that make up Jabyce. The 61 human-coded classes are the classes used as formal arguments in the signatures of methods which calls represent program elements, and the classes that implement deserializers and serializers. This demonstrates that Jaidee minimizes the effort of design and implementation of such transformation framework.
The XML file expressing a conceptual model specifies:

- the name and textual description of element types;
- the composition relations between element types;

[5] In the Thai language, "jaidee" is an adjective that means "nice". "ja" is pronounced as in "jacket", and "idee" like "ID" with a long final "ee".

- the dependencies between element types;
- the information necessary to represent elements, as "fields" that make up an element type;
- constraints on the element type fields, in the form of assertions.

For example, the description of the FieldAccessInstruction element type, in the Jabyce model is expressed as the XML element partially given in Fig. 17. The complete Jabyce model is an XML file that contains one such element for each one of the 50 element types described in section 6. The text elements are used to generate the comments in the source files, as illustrated in Fig. 8. The fields are used to define the formal arguments of the "create" method. The validity test expression elements are used to generate the pre-conditions, i.e. @pre tags in the comments of the "create" method, that are interpreted and compiled using the iContract [13] Design by Contract [24] tool.

```
<entityType name='FieldAccessInstruction'>

<singularTextName>instruction that accesses a field</…>

<compositeEntityType name='Method'/>
<depTargetEntityType name='Field'/>

<physicoLogicalField index='0'>
  <name name='type'/>
  <javaType name='…ObjectType'/>
  <text>type of the target object
    that defines the accessed field</text>
</physicoLogicalField>

<physicoLogicalField index='1'>
  <name name='fieldName'/>
  <javaType name='java.lang.String'/>
  <text>name of the accessed field</text>
</physicoLogicalField>

<physicoLogicalField index='2'>
  <name name='fieldIsStatic'/>
  <javaType name='boolean'/>
  <text>flag that indicates that if true,
    the field is static</text>
</physicoLogicalField>

<physicoLogicalField index='3'>
  <name name='fieldType'/>
```

```
  <javaType name='...FieldDescriptor'/>
  <text>accessed field type</text>
</physicoLogicalField>

<physicoLogicalField index='4'>
  <name name='loadOrStore'/>
  <javaType name='boolean'/>
  <text>flag that indicates that if true, the field
    is loaded; otherwise it is stored</text>
</physicoLogicalField>

<validityTestExpression>{0}!=null</...>
<validityTestExpression>{1}!=null</...>
<validityTestExpression>{3}!=null</...>

</entityType>
```

Fig. 17. Model of the FieldAccessInstruction element type

In the generated source code, the dependency relations are expressed as inheritance relations between the abstract component classes and the interfaces. For instance, the generated AbstractFieldTransformer abstract class inherits from the FieldFactory interface, but also from the FieldAccessInstructionFactory interface because of the declaration of a dependency relation between FieldAccessInstruction and Field. Such semantic dependencies are expressed using <depTargetEntityType> XML elements.

It is possible to compose distinct transformation frameworks by implementing *translators*, i.e. components that offer all the server interfaces of the *source framework*, and all the client interfaces of the *target framework*. It initiates interactions that represent elements in the target framework, according to the elements represented in the source framework. For instance, we have developed a prototype of a Java bytecode to C compiler, that is designed as a translator component that relies on Jabyce as the source framework.

10. Conclusion and Perspectives

This article presents Jabyce, a software framework for the implementation and composition of transformers of compiled Java classes. Jabyce is mainly targeted at the implementation of transformers that weave code into programs, possibly at program load-time. Therefore, for the sake of transformation performance, we choose to represent programs as interaction sequences, which is an optimal representation that fits well

with such transformations. To allow an easy reuse and a flexible composition of transformer implementations, Jabyce is based on Fractal, a general component model. This allows users to compose transformers using user-specified component bindings, to express arbitrarily complex transformation strategies. Jabyce is compared with other systems that transform Java classes, by comparing characteristics such as their possible uses, the model that is used to represent programs, and the computational and architectural model they use to implement transformers. Jabyce appears as the most general and flexible of these systems. In addition, performance measurements show that the choice of an interaction sequence representation of programs offers good transformation performance, and that the use of a general component model such as Fractal does not add any overhead. The design process of transformation frameworks like Jabyce is reproducible, using the Jaidee tool, which we have developed, and that generates automatically the source code of most part of a transformation framework like Jabyce, given a formal conceptual model of the elements to transform.

The Jabyce framework is operational and will be used to extract structural information from programs to produce skeletons of transparent object persistence mapping specifications. It is also used to weave transparent persistence code into Java programs, with respect to the Java Data Objects (JDO) standard. We will also use it to implement extensible and adaptable component containers, that provide services such as transparent security and transaction demarcation.

Some transformations, such as optimizations or refactorings, are difficult to implement with an interaction sequence representation of programs, and would require manipulating graphs of objects. Therefore we plan to support object graph manipulating transformers in Jabyce. Such object graphs would be automatically "deserialized" out of interaction sequences that represent programs, and "serialized" into corresponding interaction sequences after the graphs have been transformed.

We are investigating several architectures and formalisms for the specification and validation of contracts, at transformation time, to ensure the correctness of represented programs. We plan to extend the Jaidee transformation framework generator, to generate automatically such formal contract specifications for each generated framework.

Currently, transformers are directly implemented as Java classes that specify components. It would be interesting to investigate the design of higher-level languages, such as declarative rewrite rule languages, that could be compiled into such transformer implementation Java classes. In particular, Domain Specific Languages could ease the specification of a transformation for a limited range of transformations.

Acknowledgements

We would like to thank the ComSIS reviewers, and the editor and reviewers of the Science of Computer Programming 2003 Special Issue on Program Transformation to whom we submitted an early version of this article. Their numerous remarks were very valuable for enhancing this article.

References

1. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: An open component model and its support in Java. In Proceedings of the 7th Component-Based Software Engineering International Symposium (ICSE2004-CBSE7), volume 3054 of Lecture Notes in Computer Science, 7-22. Springer. (May 2004)
2. Bruneton, E., Coupaye, T., Stefani, J.-B.: The Fractal Component Model Specification 2.0, http://fractal.objectweb.org/specification/index.html (Feb. 2004)
3. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a code manipulation tool to implement adaptable systems. In Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems), Grenoble, France. (Nov. 2002)
4. Byte Code Engineering Library (BCEL). http://jakarta.apache.org/bcel/
5. Chiba, S.: Load-time structural reflection in Java. In Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000), Lecture Notes in Computer Science, volume 1850, 313–336, Sophia Antipolis and Cannes, France, Springer–Verlag. (Jun. 2000)
6. Cohen, G. A., Chase, J. S., Kaminsky, D. L.: Automatic program transformation with JOIE. In Proceedings of the 1998 USENIX Annual Technical Symposium, 167–178. (1998)
7. Jonge, M. de, Visser, E., Visser, J.: XT: a bundle of program transformation tools. Electronic Notes in Theoretical Computer Science, 44. (2001)
8. Deutsch, L. P.: Design reuse and frameworks in the smalltalk-80 systems. In Biggerstaff, T. J., Perlis, A. J. (eds): Software Reusability, volume II, 57–71. ACM Press, New York. (1989)
9. Visser, E. et al.: The online survey of program transformation – http://www.program-transformation.org/survey.html
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object–Oriented Software. Addison Wesley. (1994)
11. Hagimont, D., Mossière, J., Rousset de Pina, X., Saunier, F.: Hidden software capabilities. In Proceedings of the International Conference on Distributed Computing Systems, 282–289. (1996)
12. Le Hors, A., Le Hégaret, P., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S.: Document object model (dom) level 2 core specification – http://www.w3.org/tr/dom-level-2-core/ (Nov. 2000)
13. iContract. http://www.reliable-systems.com/tools/iContract/iContract.htm.

14. ITU/ISO Reference Model of Open Distributed Processing – Part 3: Architecture, International Standard ISO/IEC 10746–3, ITU–T Recommendation X.903. (1995)
15. Jakarta Avalon Framework. Inversion of control – http://avalon.apache.org/framework/cop/guide-patterns-ioc.html
16. jclasslib. http://www.ej-technologies.com/products/jclasslib/overview.html
17. Johnson, R. E.: Frameworks = (components + patterns): How frameworks compare to other object-oriented reuse techniques. Communications of the ACM, 40(10):39–42. (Oct. 1997)
18. Keller, R., Hölzle, U.: Binary Component Adaptation. In Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98), Lecture Notes in Computer Science, volume 1445, 307–329, Brussels, Belgium. Springer–Verlag. (Jul. 1998)
19. Kniesel, G., Costanza, P., Austermann, M.: JMangler – a framework for load-time transformation of Java class files. In Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001). (Nov. 2001)
20. Bok Lee, H., Zorn, B. C.: BIT: A tool for instrumenting Java bytecodes. In Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'97), Monterey, California, USA. (Dec. 1997)
21. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley. (1999)
22. Martin, R. C.: Granularity. C++ Report, 8(10):57–62. (Nov.-Dec. 1996)
23. Martin, R. C.: The Interface Segregation Principle. C++ Report. (Aug. 1996)
24. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall. (1997)
25. ObjectWeb ASM. http://asm.objectweb.org/
26. ObjectWeb Fractal. http://fractal.objectweb.org/
27. Partsch, H., Steinbrüggen, R.: Program transformation systems. ACM Computing Surveys (CSUR), 15(3):199–236. (1983)
28. SAX. Events vs. trees – http://www.saxproject.org/?selected=event
29. SAX. http://www.saxproject.org/
30. Serp. http://serp.sourceforge.net/
31. Tip, F., Laffra, C., Sweeney, P. F., Streeter, D.: Practical experience with an application extractor for Java. In Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99), 292–305. ACM Press. (Nov. 1999)
32. Visser, E.: A survey of rewriting strategies in program transformation systems. In Gramlich, B., Lucas, S. (eds): Proceedings of the 1st Workshop on Reduction Strategies in Rewriting and Programming (WRS'2001), volume 57 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers. (May 2001)
33. Vlissides, J.: Protection. I. the Hollywood Principle. C++ Report, 8(2):14, 16–19. (Feb. 1996)

Romain Lenglet received the ENSIMAG Computer Science Engineer degree and the Research M.Sc. degree from the INPG Technological University, Grenoble, France. Since 2001 he is working on his Ph.D., in a

context of collaboration between the France Telecom R&D Division and the French National Institute for Research in Computer Science and Control (INRIA). His research areas include program transformation, distributed systems, software architecture and dependability. He is the developer of several software packages related with program transformation, including Jabyce and Jaidee.

Thierry Coupaye completed his Ph.D. in Computer Science from the UJF Grenoble University, France, in 1996 in the area of active databases and worked afterwards as a teaching assistant at INPG Technological University. Then he worked as a researcher at the European Bioinformatics Institute (EMBL-EBI) in Cambridge, U.K., in the area of semi-structured data management, and then in the Dassault Systems and University of Grenoble Joint Laboratory where he worked on large scale software deployment. He joigned France Telecom in 2000. He leaded several R&D projects and took lead of the Distributed Software Architectures & Infrastructures Research Pole in 2003. His research interests include software architecture, component-based systems, AOP, reflexive systems and autonomic computing.

Eric Bruneton completed his Ph.D in Computer Science from the INPG Technological University, Grenoble, France, in 2001, in the area of distributed systems and reflexive middleware platforms. He entered the France Telecom R&D Division afterwards, as a research engineer. His research areas include reflexive systems, middleware platforms, component-based systems and program transformation. He is the developer of several software packages, including the ASM Java bytecode transformation system, and the Fractal Julia platform.