

Source-to-Source Transformation In The Large

Thomas Genssler and Volker Kuttruff

Forschungszentrum Informatik Karlsruhe
Haid-und-Neu-Straße 10-14
76131 Karlsruhe, Germany
{genssler|kuttruff}@fzi.de

Abstract. In this paper we present Inject/J¹, a language and a tool for specifying complex source-to-source transformation of Java programs. The focus of Inject/J is on “transformation in the large” that is, modification of large object-oriented software on the design level. We first introduce the meta-model of our transformation language. This meta-model provides a conceptual view on object-oriented software by capturing relevant design entities. It also defines a number of conceptual analysis and transformation operations together with their code-level semantics. The entities of the meta-model together with the respective operations constitute the primitives of our transformation language. We discuss the main features of this transformation language and illustrate how it can be used to perform complex transformation tasks.

Topics: Reflective programming, Software development environments

1 Introduction

There is one constant in the life of software: it changes. It may change because of changing requirements or due to bug-fixing. Changing large-scale software is a difficult task that calls for tool support, especially if the changes have global effects. Automated source-to-source transformation is one possible solution. With source-to-source transformation we denote the purposeful modification of a piece of software by changing the source code directly. The most prominent example is probably refactoring [Opd92], [Fow99].

In this paper we present an approach to the specification of complex source-to-source transformations and their application to large object-oriented software. We focus on design level transformations. Design level transformations modify entities (classes, methods etc.) which contribute to the design of a system as well as the interactions among these entities at runtime (control flow along method invocations or accesses to attributes of a class). An important characteristic of these transformations is that their effects can usually not be limited to one particular design entity. Consider renaming a class: not only the class itself has to be

modified but also all references to this class, which might be scattered throughout the entire system. We distinguish *primary transformations* (i.e., renaming the class) and *secondary transformations* (i.e., adapting the references).

When transforming large software, one of the main problems is the huge amount of information. Therefore, we need *abstraction* and *locality*. We want to be able to focus on only those parts of the system we want to transform (certain classes or specific set of method calls) rather than having to understand the system in its entirety. However, sometimes these parts do not comply to the decomposition mechanism used in the programming language (i.e., classes or methods). Consider the case that you want to replace direct attribute accesses with calls to the respective set- and get-methods. The set of accesses probably crosscuts a huge number of classes. Thus we need a technique to describe, to identify, and to transform these possibly distributed elements in a coherent way.

Another difficulty in automated software transformation is *correctness*. We need a notion of correctness, even if allowing general-purpose transformations. The transformation result must at least be correct according the syntax and the static semantics of the programming language. In addition, we need a mecha-

¹ <http://injectj.sf.net>

nism to impose stricter constraints, e.g., to ensure behavior preservation.

Last but not least, we need *extensibility*. Since there is no complete list of all possible transformations, we need to be able to specify new transformations in terms of existing ones.

Our approach is based on three columns. We first define a two-layer meta-model for large-scale source-to-source transformations. The upper layer introduces a conceptual view on object-oriented languages. It captures common design entities of those languages (e.g., classes, methods) and elements of the statically observable control and data flow among those design entities, such as method invocations or accesses to attributes. This layer also defines a rich set of conceptual analysis and transformation operations. The lower layer defines the mapping between the conceptual view and the actual language semantics. It maps conceptual model entities onto language entities and defines the language-dependent semantics of analysis operations. It also defines semantics of transformation operations in terms of their primary and secondary code-level transformation. Pre- and post-conditions ensure for each transformation that the transformed system remains correct with respect to the syntax and the static semantics of the programming language used. Currently, the lower layer is specified and implemented for Java.

We then define a transformation language based on our meta-model. This language provides a powerful and expressive means to specify complex transformation operations. It provides language constructs to query the model, to match patterns, and to perform transformations. It also provides user pre- and post-conditions to impose stronger constraints than those already defined by the model-to-language mapping.

Finally, we provide a tool that constructs the conceptual model from source code, performs transformation specified in our transformation language and regenerates the new code.

The remainder of this paper is organized as follows. In section 2, we introduce our conceptual meta-model and show how it maps to the Java programming language. Section 3 is devoted to our transformation language. We show how our approach can be used to specify and perform practical source-to-source transformations on object-oriented code. We also sketch our tool Inject/J. This tool provides the

necessary infrastructure for source-to-source transformation of Java programs. After a discussion of our approach in section 4, we give an overview on related work in section 5 and conclude in section 6.

2 A Meta-Model for Source-to-Source Transformation in the Large

In this section we introduce our meta-model and show how it maps to Java.

Our meta-model (see Fig. 1) consists of two layers. The upper layer defines a conceptual view for design level transformations. Its goal is to provide an abstraction layer for the specification and application of transformations with non-local effects while shielding the user from the complex syntax tree of the underlying language. The lower layer defines a language-specific mapping of the conceptual entities and operations.

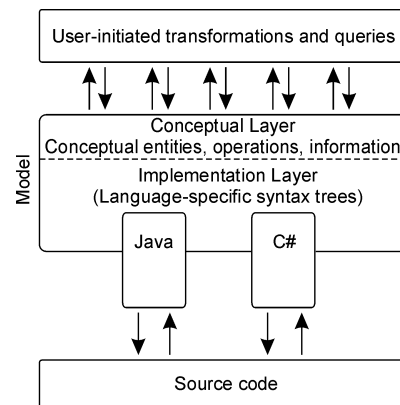


Fig. 1. Inject/J Meta-Model

Typical design level transformations with non-local effects are, among others, method and attribute refactorings (e.g., encapsulate field), class-level refactorings (e.g., replace inheritance with delegation), or modifications of the control- and data-flow among objects (e.g., replace method with method object or code instrumentation along delegation paths). Hence this model captures the following common elements of object-oriented languages that either contribute to the design of a program or represent significant points in the interaction among design entities:

Declaration of structural entities

- Package
- Class, Parameterized Class, Array, Type Parameter
- Attribute
- Local Variable
- Formal Parameter

Declaration of behavioral entities

- Method
- Constructor
- Property
- Exception handler

Entities modelling control or data flow

- Accesses to declared elements, Access paths
- Return
- Exception
- Assignment

A package is a scoping mechanism providing a new namespace, used to group (parameterized) classes. In Java, a package is primarily a grouping mechanism, since it does not provide a new namespace on its own. The different types (class, parameterized class, array, type parameter) are used to reflect a language's type system. For our Java implementation, classes, interfaces, inner classes as well as base types are all mapped to the model entity 'class'. Since Java does not (yet) support type genericity, parameterized classes and type parameters are not used in the Java mapping. Attributes, local variables and formal parameters map in a straightforward way.

Behavioral entities define elements that take part in the system's control flow. All the different types of methods found in Java, like abstract methods, interface methods and executable methods, are mapped to the behavioral entity 'method'. Constructors are modelled as separate entities instead of a special kind of method. This is due to the fact that certain properties and transformations for methods may not be applicable to constructors (e.g., the 'rename' transformation, syntax for accessing a constructor). Java constructors map thus to the model entity constructor. Java's class and instance initializer are also mapped to the 'constructor' model entity. Properties, as found in other object-oriented languages like C#, do not exist in Java and are thus not considered.

In our meta-model, accesses are references to structural or behavioral entities, e.g., method

invocations changing the control flow or attribute/variable accesses describing the data flow. They map directly to the corresponding Java language constructs. The 'return' model element is the point in a method where the control flow leaves the method if no error occurred. The 'exception' entity is the point in the control flow where an exception is raised, thus leaving the method's control flow in a non-regular manner. The 'assignment' entity is important, because it exposes the data flow from one structural entity to another. All these entities have direct correspondence in the Java meta-model.

Information provided by the model In general, each model entity has a set of attributes providing detailed information. Some of them can be gathered directly from a system's syntax tree, like the name of a structural entity, others need an explicit semantic analysis or more advanced computation.

The information provided by model element attributes can be categorized as follows:

- syntactical information, e.g., class, method and attribute name
- basic semantical information, e.g., static type information, functions to test inheritability of class members
- crossreference information
- basic information for metrics calculation, e.g. lines of code for a class/method, number of nodes/edges in the control flow graph of a method

Information provided by model entities serves to reason about properties of these entities, to select or to filter them. Even more important, specifying and checking preconditions and post-conditions heavily relies on the information provided by the model.

Exact computation of these information, as needed in our model (e.g., for precondition checking), requires a closed world. Additionally, there is also no support for reflective programs (e.g., programs using Java reflection).

Conceptual transformations Conceptual transformations are the basic operations to modify a model instance. However, a single conceptual transformation can result in an extensive reorganization of the source code of the system. Changing a method's signature

	Package	Class	Method	Constructor	Attribute	Property	Local Variable	Class Access	Method Access	Constructor Access	Attribute Access	Property Access	Local Var. Access	Access Path	Assignment	Return	Exception	Exception Handler	Remarks
create	x	x	x	x	x	x													
delete	x	x	x	x	x	x	x												
change signature																			
modifier			x	x	x	x													
name	x	x	x		x	x	x												
type			x		x	x	x												
superclasses		x																	
parameter			x	x															
exceptions			x	x															
The conceptual transformations below allow the insertion of arbitrary code fragments																			
replace body			x	x															x
replace ...								x	x	x	x	x	x						
before							x	x	x	x	x	x	x						before control flow (=cf) reaches entity
after							x	x	x	x	x	x	x						after cf passed entity
before entry			x	x															before cf passes to entity, parameters are already evaluated. Optional 'from' restricts places of invocation
before return			x	x															x before cf leaves entity after normal completion
before failure			x	x															x before cf leaves entity after abnormal completion. Optional 'with' restricts failure reasons
before exit			x	x															x before cf leaves entity (normal or abnormal completion)
after entry			x	x															x after cf passed to entity
after return			x	x															after cf returned to place of invocation (restricted by optional 'to') after normal completion.
after failure			x	x															after cf returned to place of invocation (restricted by optional 'to') after abnormal completion
after exit			x	x															after cf returned to place of invocation (restricted by optional 'to')

Fig. 2. Overview of conceptual transformations supported by the transformation model

is an example for such a conceptual transformation, since all call sites of the method must be adapted as well. In general, a conceptual transformation can be characterized by the fact that in addition to the entity to be changed (*primary transformation*), all structures depending upon this entity are also adapted automatically (*secondary transformation*). This basically means that a conceptual transformation can be used with only local knowledge about its effects. An overview of our conceptual transformations is given in Fig. 2.

In our model, each conceptual transformation is a quadruple $(s, pre, t, post)$ with:

1. *Signature s*: The signature of a conceptual transformation is part of the upper model layer. It defines the name and the parameters of the transformation.
2. *Set of pre-conditions pre*: Each conceptual transformation has a set of pre-conditions, which ensure correctness of the transformation result according to the syntax and the static semantics of the underlying programming language. Since these pre-conditions are language-specific, *pre* is part of the language mapping

3. *Code-level Transformations t*: Code-level Transformations are defined in terms of the necessary primary and secondary transformations that is, all language-specific source code modifications needed to perform a) the actual primary transformation and b) all necessary secondary transformations to make sure that the result remains compilable. Some transformations may require user interaction, e.g., the default value in a method access if adding a new parameter to the method declaration.
4. *Set of post-conditions post*: Post-conditions define properties that the code fulfils after the transformation, given that the transformation succeeded.

All our conceptual transformation have been specified in this manner. An example can be found in Fig. 3.

Some conceptual transformations like *before entry* allow the insertion of arbitrary code fragments. This insertion is also guarded by a set of pre-conditions, since the code fragments must match with their new context. For example, pre-conditions ensure that code fragments are syntactically correct, that they

Signature s :	<code>delete class (c)</code>
Preconditions pre :	$\forall x \in accesses(c) : surroundingClass(x) = c$ $\wedge \neg isLocalClass(c) \wedge \neg isAnonymousClass(c)$
Transformations t :	Remove class c from the system. If c is a Java primary class, also remove the compilation unit, unless another class is declared in this compilation unit.
Postconditions $post$:	$classes^{new} = classes \setminus \{c\}$

Fig. 3. Example for a formal specification of a conceptual transformation

do not create dead code and that they do not redefine used variables. While inserting new code fragments, it is often necessary to adapt the context. For example, if adding a code fragment before an access to an attribute a , a method access $m(x, a)$ using this attribute must be transformed as well. This is due to the fact that the new code fragment can not be inserted inside the parameter list. If necessary, such context adaption is performed automatically by the conceptual transformation. For each context adaption, the conceptual transformation guarantees an unchanged evaluation order, e.g., by using temporary local variables.

For each conceptual transformation, detailed Java-specific pre- and post-conditions have been formalized. These conditions are part of the transformation model and ensure that the transformed sources remain compilable.

3 The Inject/J Software Transformation Language

In the previous section we have sketched our meta-model together with a model mapping to Java. This meta-model provides us with the necessary functionality to reason about software and to transform it on the design level. We are now ready to introduce our software transformation language Inject/J. Inject/J is a mixture of a pattern-based transformation system, i.e., so-called detection patterns are used to identify structures which are to be transformed, and a programmatic one, i.e., the actual transformations are described imperatively. The main goal of the language is to make specifying transformations as simple as possible for a typical software engineer.

3.1 Language Basics

The Inject/J software transformation language is a dynamically typed scripting language that

serves to specify complex transformation operators in terms of model entities and conceptual operations. It provides full access to the upper meta-model layer. It has language constructs to navigate through a model instance, to select model entities according to their properties and to perform model transformations.

The following shows a very simple Inject/J script. This script only consists of a sequence of selection operations and one transformation.

```

1 script HelloWorld {
2   foreach c in classes.filter('mypckg.*') do {
3     foreach m in c.methods do {
4       m.afterEntry( ${
5         System.out.println(
6           "In method <c.name>.<m.signature>!");
7       }$ );
8     }
9   }
10 }
```

The above script introduces the code snippet that prints the name of the method and class as first statement into each method of each class in package `mypckg`.

Our language provides a number of built-in types. These types comprise all entities defined by our model (e.g., class, method, access) as well as a number of general-purpose data types (e.g., strings and lists). The language has quantors (i.e., `foreach` and `exists`) and provides the concept of code blocks².

To declaratively describe code patterns that are to be transformed, the concept of *detection patterns* is provided. Detection patterns serve to detect these code patterns in the system in order to perform the necessary transformations. Detection patterns are basically graph patterns that match – according to certain conditions – sub-graphs in the graph representing the software system. A detection pattern consists of a signature and a body. The signature contains an optional list of parameters to configure

² Code blocks have the form `'[' <optParams> '| ' <exprList> ']'`. They can be bound to variables and passed as parameters.

```

1 script Logger {
2   m = classes.filter('mypackage.MyClass').filter('int myMethod()');
3   m.beforeEntry('sourcePackage.*', ${ Logger.getInstance().print("Calling <m.signature>"); }$);
4 }

```

Java source before transformation

```

...
o.f(anotherMethod(), mc.myMethod())
...

```

after transformation

```

...
<type> tmp1 = anotherMethod();
// begin injected code
Logger.getInstance().print("Calling int myMethod()");
// end injected code
int tmp2 = mc.myMethod();
o.f(tmp1, tmp2);
...

```

Fig. 4. Inject/J script and its code level effects

the pattern and a mandatory selector. The selector serves to distinguish different matches. The body of a detection pattern consist of an optional initializer, an optional list of pattern variables, an optional list of additional functions and a mandatory list of conditions. The list of conditions specify the properties an instance of a particular pattern needs to have. The following shows, how one could express our introductory example using a detection pattern. We use two pre-defined sets `classes` and `methods` that contain all classes or methods of the system, to instantiate the pattern.

```

1 script HelloWorld {
2   pattern MyPattern():(c, m) {
3     conditions {
4       (c.package == 'mypckg'), (m in c.methods);
5     }
6   }
7   foreach p in
8     MyPattern()(classes, methods) do {
9     // [...]

```

The selector of the used pattern consists of a class and a method. The elements of the selector are bound to the elements of the cross-product of sets used in the pattern instantiation (e.g., `classes` and `methods`). Two structures matched by this pattern are different from each other, iff the binding of at least one of the parameters of the selector is different, i.e., ('MyClass', 'myMethod') differs from ('MyClass', 'myOtherMethod'). We call structures that match with a particular detection pattern *pattern instances*. The order in which pattern instances are processed is not defined.

After identifying pattern instances (or navigating to a certain location in the model instance), we can apply transformations. Inject/J provides the same conceptual transformations as defined in our model (see Fig. 2 for a summary). Transformations can easily be

composed to construct more complex operations. Each conceptual Inject/J transformation automatically takes care of language-specific code-level effects (such as flattening of expressions). Each conceptual transformation guarantees that the code remains compilable. It also guarantees that the execution or evaluation order will not be changed (i.e., in case expressions have to be split, the evaluation order is guaranteed to remain the same). Fig. 4 gives an example of a transformation of distributed elements (i.e., methods call-sites) that potentially requires a large number of secondary transformations. The examples also illustrates code-level effects of the transformation.

Matching detection patterns (basically graph patterns) and transforming them (basically graph rewriting) may lead to *non-termination*. This is the case, when a transformation applied to a pattern instance constantly creates new pattern instances. To avoid this, we have chosen the following approach. The set of all pattern instances (of possibly different patterns used in one script) is constructed, before any transformation the pattern is possibly involved in, takes place. New pattern instances that are created during the transformation process are not considered.

The case that two pattern instances overlap each other also has to be considered. The transformation of one of these instances could "destroy" another one, i.e., change its properties in a way that it is no longer a pattern instance. To deal with this, each pattern instance is analyzed before it is processed, whether parts of it have been already modified. If so, the pattern conditions are re-checked for this instance. If they still hold, the pattern instance can be transformed. Otherwise, the instance is ignored.

3.2 Examples

In this section we show, how Inject/J can be used to specify and to perform source-to-source transformations of Java programs.

Example 1 In the first simple example we show, how Inject/J can be used to deal with problems that often arise when switching from one library or language version to another. With Java 1.4, the new keyword `assert` has been added to the language. However, many people wrote their own assertion class with the respective `assert` method. In order to comply to the Java 1.4 language, the method `assert` has to be renamed (refactoring *Rename Method*). The following script illustrates how this can be done with Inject/J.

```
1 script FixAssert {
2   meth = classes.map(
3     [cur|cur.methods.filter('* assert(*)')
4     ].flatten();
5   foreach m in meth do {
6     sig = '* require' + m.paramList.toString();
7     pre:
8     ! m.surroundingClass.hasMethod(sig);
9     m.rename('require');
10  }
11 }
```

In the above script, the pre-condition has been stated explicitly. In this particular case it would, however, not have been necessary, since this condition is already checked implicitly as part of the transformation `rename`.

Example 2 The second example shows how Inject/J can be used to fix a common design problem, e.g, classes with different responsibilities. Each class that implements more than one key abstraction (responsibility) is considered a problem (e.g., in [Rie96]). We show how the description of the problem can be formalized using detection patterns. We also specify the necessary transformations to correct the problem.

Classes with different responsibilities are usually complex and their attributes and their methods can be clustered in disjoint, highly cohesive sets with pair-wise low cohesion. To detect such classes, complexity metrics such as Weighted Method Count (WMC) [CK94] and cohesion metrics, such as Tight Class Cohesion (TCC) [BK95] can be used. [Fow99] proposes to use the refactoring "Extract class" to split classes with multiple responsibilities.

The detection pattern we use to detect instances of our problem is shown in Fig. 5. It detects complex classes with cohesive clusters of

methods according to usage of class variables. For reasons of clarity we use a rather naive approach to formalize our problem³. We refer the reader to work on coupling and cohesion metrics for an extensive discussion on how to formalize this problem.

Note that this detection pattern identifies only problem *candidates*. This is due to the fact that the problem is described in a fuzzy way (using percentages, thresholds etc.). We need human interaction to decide, whether a problem candidate is an actual problem structure. Our language provides support for that. The respective code has, however, been omitted from Fig. 6.

Once having confirmed that the candidate is a real problem instance, we can apply the necessary transformations. In Fig. 6, we only show the use of the complex refactoring `Extract Class` for reasons of clarity (line 8).

Other application areas Besides the correction of design problems, our approach can also be used for different purposes. One examples is code instrumentation, possibly as a pre-step of compilation, another is code generation (e.g., in the context of Java EJB systems). We have also been working on the integration of techniques from Adaptive Programming (AP) [Lie95]. In [SGS02] we show, how our approach can be extended to support introduction of new functionality along inheritance and delegations paths. Another area is the automated application of design patterns [SGMZ98], [Ó00].

3.3 The Inject/J tool

Together with our transformation language, we developed the tool Inject/J⁴ to apply transformations to a software system. Inject/J is based on the meta-programming library Recoder for Java ([Rec02], [LH02]). Recoder provides the necessary infrastructure to build an abstract syntax tree together with semantic information (e.g., cross-reference and type information). Recoder also allows to modify this syntax tree and to re-generate code from it. Recoder takes

³ We actually only simulate TCC. However, all information necessary to compute TCC is available in our model.

⁴ [GK03], The current version still works with an old transformation language. The new transformation language is currently being integrated.

```

1 pattern CWMR( methodThreshold, similarityPercentage, clusterSize) : (c) {
2   vars clusters;
3   init {
4     threshold = c.attributes().size() * similarityPercentage;
5     tmpMeths = c.methods().clone();
6     cluster = []; clusters = [];
7     // Find similar methods. Similarity is defined by the amount of common attributes uses
8     foreach m_i in c.methods do {
9       tmpMeths.removeElement(m_i);
10      cluster.addElement(m_i);
11      foreach m_j in tmpMeths do {
12        tmpAtts = m_i.accessedAttributes().intersect(
13          m_j.accessedAttributes().intersect(c.attributes()) );
14        // attributes used by both methods
15        if( tmpAtts.size() > threshold )
16          cluster.addElement(m_j);
17      }
18      if(cluster.size() > clusterSize)
19        clusters.addElement(cluster.clone());
20      cluster.clear();
21    }
22  }
23  conditions { (c.WMC() > methodThreshold), (clusters.size() >= 2); }
24  public getClusters() { return clusters; }
25 }

```

Fig. 5. Detection pattern for ‘Classes with multiple responsibilities’

```

1 use library cwmr;
2 script FixCWMR {
3   foreach p in CWMR(...)(classes) do {
4     foreach cluster in p.clusters do {
5       name = "";
6       ask("Qualified name of the new class:", name);
7       clusterAtts = cwmr.getAttributesForCluster(cluster);
8       extractClass(p.c, cluster, clusterAtts, name);
9     } // foreach cluster
10  } //foreach p
11 } // end of script

```

Fig. 6. Transformations for ‘Classes with multiple responsibilities’

care of layout preservation of source files (i.e., indentation and comments are maintained). In order to compute cross-references, Recoder also analyzes Java byte-code files. However, modification is only allowed for software available in source code. Inject/J interfaces with the Recoder syntax tree and constructs its own model. Inject/J scripts have full access to this model that is, they can access model information and perform model transformations. Inject/J provides support for projects consisting of a set of source files and class files (read-only), a set of transformation scripts and optional libraries containing pre-defined detection patterns or transformations.

4 Discussion And Open Issues

The main goal of our approach was to provide a software engineer with a simple yet practical tool for source-to-source transformation

of large object-oriented systems. To be able to deal with large systems, we provide a conceptual model that shields the user from the complexity of the underlying language model and provides significant abstraction. For structure-rich Java systems, our approach usually reduces the number of model elements – compared to the complete syntax tree – by about 60%⁵. For algorithmic-centric systems, the reduction is even better. Besides this information reduction, our model together with the Inject/J language allow for the specification of complex transformations with local knowledge about the system. The software engineer can focus on those entities he actually want to transform while all necessary secondary transformations are taken care of automatically by our tool. With detection patterns one can describe, match and transform structures in the

⁵ e.g., Recoder for Java [Rec02], 85kLOC, 500 classes; reduction: ~65%

code that are potentially distributed throughout the entire system, in a coherent way. Inject/J also supports pre- and post-conditions to impose stronger constraints than those defined as part of the conceptual transformations. For this purpose, the language provides full access to analysis information provided by the transformation model. We have specified behavior-preserving pre-conditions for 26 design-level refactorings presented in [Fow99], for the automated application of a number of design patterns and a number of code instrumentations.

There are, however, some open issues. Our approach does not consider reflective features of programming languages, such as Java reflection. This is an intrinsic limitation of tool-supported software transformation. A possible solution is to use heuristics or conservative approximations to deal with this problem. Another intrinsic property of software transformation is that there is no support for incomplete code (i.e., code that is not compilable). Currently, we do also not support the automatic inference of pre-conditions for composite transformations. This means that a long-running transformation may fail “in the middle” and must thus be rolled back completely. A possible solution to this problem is described in [Rob99]. Another problem is that design entities may not only be referenced from other design entities but also from different artifacts such as Javadoc comments or design documents. Currently we do not support secondary transformations to keep artifacts other than the code itself consistent (e.g., by renaming a Javadoc reference to a class when the class is renamed).

5 Related Work

Several approaches have been proposed in literature to support software transformation in the context of object-oriented software evolution – the most visible among them is without doubt refactoring [Opd92], [Fow99] [BR02]. Although refactorings define a set of conceptual transformations together with pre- and post-conditions, their code-level effects as well as the needed analysis information is only semi-formalized if at all. There is no notion of extensibility (apart from sequential execution of several, otherwise independent refactorings). Pattern detection support, i.e., identifying structures where to actually apply refactorings,

other than the rather informal “bad smells” in [Bec99] and [Fow99], is not provided. There exist a number of efforts to improve refactorings. [Tic01] proposes a language-independent model for refactoring. This model provides conceptual analysis and transformation operations together with a formalization of their language-specific code-level effects. [Rob99] proposes a technique to derive pre- and post-conditions of complex refactorings from the conditions of their constituent refactorings. However, neither extensibility nor pattern detection support are considered.

There are a number of general-purpose transformation systems that address source-to-source transformation. Most of them are so-called pattern-based transformation systems, e.g., Design Maintenance System (DMS) [Des03], ASF+SDF [ASF03], JATS [CB], to name a few. A source pattern, described in a special pattern language, is searched in the program representation and replaced by a target pattern. Both, the source pattern and the target pattern are declaratively specified in so-called rewrite rules. The following shows an example of a rewrite rule to replace `while`-statements with `if-goto` combinations.

```
while(cond) stmt =>
  L: if (cond) {stmt; goto L;}
```

Pattern-based transformations theoretically offer – due to their declarative nature – an expressive, simple and easy to understand means to specify complex transformations. In practice, however, with pattern languages it often gets very inconvenient to specify transformations which rely on a lot of contextual information. Our detection patterns compare to source patterns in rewrite rules. We use, however, a programmatic approach to specify the actual transformations.

Detection patterns also compare to join-point descriptions, used for aspect composition in aspect-oriented programming ([KLM⁺97]). In aspect languages (e.g., AspectJ [Asp03]), one can specify so-called point-cuts – sets of points in a program (method entry/exit, method call, object access etc.) where aspect code is woven. Aspect languages do, however, not support general design transformations (e.g., there is no support for behavior-preserving restructuring).

6 Summary

In this paper we have presented the transformation language Inject/J. The basis of our approach is a meta-model for source-to-source transformation of large object-oriented software. This meta-model provides an abstraction layer that shields the user from the complexity of code-level transformation, i.e, transforming the syntax tree directly. The model provides conceptual analysis and transformations together with their language-specific pre- and post-conditions. Transformation operations are formally defined in terms of their language-dependent primary and secondary code-level transformations. Using conceptual transformations one can specify transformations that effect large parts of the system, with only local knowledge about the system (e.g., knowledge about certain classes). With the help of detection patterns, one can describe, match, and transform possibly distributed structures in a coherent way. Examples of transformations have been given to illustrate, how this all fits together.

Our future work includes further improvement of the expressiveness of the transformation language. One improvement is a wrap operation that consistently adds code before and after a certain location (e.g., conditional method invocation). Other fields of current work are the evaluation of our tool in industrial case studies and the support for C# ([Rec03]).

References

- ASF03. ASF+SDF MetaEnvironment. <http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment>, 2003.
- Asp03. AspectJ Team. AspectJ WWW Page. <http://www.eclipse.org/aspectj/>, 2003.
- Bec99. K. Beck. *Extreme Programming Explained – Embrace Change*. Addison Wesley, 1999.
- BK95. J. M. Bieman and B. K. Kang. Cohesion and reuse in an object-oriented system. *Proceedings of the ACM Symposium on Software Reusability*, April 1995.
- BR02. J. Brant and D. Roberts. The Refactoring Browser. <http://st-www.cs.uiuc.edu/users/brant/Refactory/>, 2002.
- CB. F. Castor and P. Borba. A language for specifying Java transformations.
- CK94. S. R. Chidamber and C. F. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
- Des03. Semantic Designs. DMS software reengineering toolkit. <http://www.semdesigns.com/products/DMS/DMSToolkit.html>, 2003.
- Fow99. M. Fowler. *Refactoring – Improving The Design Of Existing Code*. Addison Wesley, 1999.
- GK03. T. Genssler and V. Kutruff. Inject/J WWW Page. <http://injectj.sf.net/>, 2003.
- KLM⁺97. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 1997.
- LH02. A. Ludwig and D. Heuzeroth. Meta-programming in the large. In *Proceedings of Conference on Generative Component-based Software Engineering (GCSE)*, 2002.
- Lie95. K. J. Lieberherr. *Adaptive Object-Oriented Software – The Demeter Method*. PWS Publishing Company, 1995.
- Ó00. Mel ÓCinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. PhD thesis, University of Dublin, Trinity College, October 2000.
- Opd92. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.
- Rec02. The RECODER/Java homepage. <http://recoder.sf.net>, 2002.
- Rec03. The RECODER/C# homepage. <http://recoder-cs.sf.net>, 2003.
- Rie96. Arthur J. Riel. *Object-oriented Design Heuristics*. Addison-Wesley, 1996.
- Rob99. Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
- SGMZ98. B. Schulz, T. Genßler, B. Mohr, and W. Zimmer. On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems. In *Proceedings of the 27th TOOLS conference*, 1998.
- SGS02. O. Seng, T. Genßler, and B. Schulz. Adaptive Extensions of Object-Oriented Systems. In *Proceedings of the IFIP TC2 Working Conference on Generic Programming*. Kluwer, 2002.
- Tic01. Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, Switzerland, December 2001.