Graph Transforming Java Data*

Maarten de Mol[1], Arend Rensink[1], and James J. Hunt[2]

[1] Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE, The Netherlands
{M.J.deMol,rensink}@cs.utwente.nl
[2] aicas GmbH, Karlsruhe, Germany
jjh@aicas.com

Abstract. This paper introduces an approach for adding graph transformation-based functionality to existing JAVA programs. The approach relies on a set of annotations to identify the intended graph structure, as well as on user methods to manipulate that structure, within the user's own JAVA class declarations. Other ingredients are a custom transformation language, called CHART, and a compiler from CHART to JAVA. The generated JAVA code runs against the pre-existing, annotated code.

The advantage of the approach is that it allows any JAVA program to be enhanced, non invasively, with declarative graph rules, improving clarity, conciseness and verifiability.

1 Introduction

Proponents of Graph Transformation (GT) as a modeling technique have always claimed as strong points its general applicability and its declarative nature. Many structures can naturally be regarded as graphs and their manipulation as a set of graph operations. For these reasons, GT has been advocated in particular as a vehicle for model transformation [5,14,16], a major component in the Model-Driven Engineering (MDE) paradigm. In this paper we focus on JAVA as application domain, aiming to replace JAVA code that manipulates object oriented data by declarative graph transformations.

Weak points of GT that are often quoted are its lack of efficiency and the need to transform data between the application domain and the graph domain. Though efficiency may to some degree be the price for general applicability, this does not appear to be the dominant factor. Transforming application data structures into a well defined graph format to facilitate sound transformations and then transforming the result back to a form suitable for the application is a bigger problem. These two "transfers" are themselves really model transformations in their own right, and seriously aggravate the complexity of the technique in practice, to the point of making it completely impractical for large graphs, e.g., graphs with hundreds of thousands of nodes.

One solution to the problem is to force an application to use the graph structure of the tool as a basis for its data structures. This has serious drawbacks, as the tool graph

* This work was funded by the Artemis Joint Undertaking in the CHARTER project, grant-nr. 100039. See http://charterproject.ning.com/

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 209–223, 2012.

structure may not be rich enough for the application, and existing code that may already be in place must be rewritten. This makes GT an invasive technique.

In this paper, we propose a radically different approach, aimed at JAVA, which does not share the invasive nature yet preserves the advantages of GT, including its general applicability and its declarative nature. There are three main parts of this approach.

- The graph structure (i.e., the type graph) is specified through JAVA annotations added to existing user code classes. For instance, the framework provides annotation types to specify that a given class represents a node or edge, along with edge properties such as multiplicities and ordering. As no actual code needs to be modified, we consider our method non invasive. Effectively, the JAVA annotation types constitute a type graph specification language.
- Graph manipulation, such as adding or deleting nodes or edges and updating attributes, is achieved by invoking user-provided operations. Again, these operations need to be annotated in order to express their effect in terms of the graph structure.
- Rules are written in a (textual) declarative language (called CHART), and subsequently compiled into JAVA code that runs against the aforementioned user classes, invoking the annotated methods. This obviates the need for transferring data structures to and from the graph domain. Everything is modified in place, using preexisting code.

Our approach allows components of any existing JAVA program to be replaced with declarative graph transformations, while only requiring non invasive additions to the data structures of the program. The approach was developed in the CHARTER project [4], where it is applied within three different tools that in turn make up a tool chain for the development of code for safety critical systems; see Section 4.

1.1 Related Work

There is, of course, a wealth of approaches and tools for model transformation, some of which are in fact based on graph transformation. To begin with, the OMG has published the QVT standard for model transformation [13], which is a reference point for model transformation, even though compliance with the standard is not claimed by many actual tools. A major tool effort is the ATL approach [9]; other successful tool suites are VMTS [11], VIATRA2 [17], HENSHIN [1] and FUJABA [7].

However, none of the above share the aforementioned characteristics of the CHART approach; in particular, all of them rely on their own data structures for the actual graph representation and manipulation. Although an old implementation of ATL appears to have supported the notion of a "driver" which could be tuned to a metamodelling framework and hence imaginably to our annotations, this has been abandoned in newer versions (see `http://wiki.eclipse.org/ATL/Developer_Guide#Regular_VM`).

Another transformation framework for JAVA is SITRA [12], but in contrast to the declarative rules of CHART it still requires individual rules to be written in JAVA directly.

1.2 Roadmap

In this paper we concentrate on the fundamentals of the CHART approach. In particular, in Section 2 we introduce the formal graph model used, and show how the structure

and manipulation of graphs is specified through JAVA annotations. In Section 3, we introduce the language concepts of CHART and indicate how the formal semantics of the language is defined (details can be found in [6]). Section 4 gives an overview of the use cases of the approach within the CHARTER project.

2 Graphs and Annotations

The basic idea of our approach is that the graph to be transformed is represented externally, in JAVA. In order to be able to transform this graph, its structure must be known, and operations to manipulate it must be available. This information is obtained by an automated analysis of a JAVA program. Not all necessary information can be obtained from the source code alone, however. Therefore, we have defined a language of JAVA annotations, which must be added to the code explicitly to fill the gaps.

In the following sections, we will explain how a graph structure is recognized in a JAVA program. In Section 2.1, we first provide a formal description of the graphs and type graphs that are allowed. In Sections 2.2, 2.3, 2.4 and 2.5, we describe how nodes, edges, attributes and manipulation methods are defined, respectively. In Section 2.6, we investigate the (non) invasiveness of our approach.

2.1 Graphs and Type Graphs

Our approach operates on simple graphs (nodes, binary directed edges, attributes) that are extended with basic model transformation concepts (subtyping and abstractness for nodes; multiplicity and orderedness for edges). This leads to the following formalization of type graphs:

Definition 2.1.1: *(types)*
A type (set $\mathcal{T}$) is either a node type, a set or list over a particular node type, or a basic type. The supported basic types are boolean, character, integer, real and string.

Definition 2.1.2: *(type graphs)*
A type graph is a structure ($\mathcal{T}_n$, $\mathcal{T}_f$, src, $type_f$, abs, $\leq_t$, min, max), in which:
- $\mathcal{T}_n$ and $\mathcal{T}_f$ are the disjoint sets of node and field types, respectively;
- $src : \mathcal{T}_f \to \mathcal{T}_n$ associates each field type with a source node type;
- $type_f : \mathcal{T}_f \to \mathcal{T}$ determines the value type of each field type;
- $abs \subseteq \mathcal{T}_n$ is the subset of node types that are abstract;
- $\leq_t \subseteq \mathcal{T}_n \times \mathcal{T}_n$ is the subtyping relation on nodes, which must be a partial order;
- $min : \mathcal{T}_f \to \mathbb{N}$ and $max : \mathcal{T}_f \to \mathbb{N} \cup \{\mathsf{many}\}$ are the multiplicity functions.

Attributes and edges are collectively called 'fields'. If the maximum multiplicity of a field is greater than one, a single field connects a single source node to multiple targets. In that case, the targets are either stored in a set (unordered), or in a list (ordered).

A graph is straightforward instantiation of a type graph. However, we also require each graph to be rooted:

Definition 2.1.3: *(values)*
A value (set $\mathcal{V}$) is either a node, a basic value, or a set or list of nodes or values.

Definition 2.1.4: *(graphs)*
A graph (set $\mathcal{G}$) is a structure (N, r, F), in which:
- $N \subseteq \mathcal{N}$ is the set of nodes in the graph;
- $r \in N$ is the designated root node of the graph;
- $F : N \times \mathcal{T}_f \hookrightarrow \mathcal{V}$ are the field values in the graph.

Our semantics ensures that a node that gets disconnected from the root becomes invisible for further graph operations.

2.2 Definition of Node Types

A node type must be defined by annotating a *class* or *interface* with the custom @Node annotation. The name and supertypes (possible many) of the node type are determined directly by the JAVA representation. The @Node annotation has an additional argument to indicate whether the node type is abstract[1] or not.

Example 2.2.5: *(node type example)*
The following piece of code defines the abstract node type Book and the concrete node type Comic on the left, and the concrete node types Author and Picture on the right. Comic is defined to be a subtype of Book.

```
1  @Node(isAbstract = true)
2  public class Book { ...
3
4  @Node
5  public class Comic extends Book { ...
```

```
1  @Node
2  public class Author { ...
3
4  @Node
5  public class Picture { ...
```

In the JAVA code that will be produced for transformation rules, instance nodes will be represented by objects of the associated JAVA class or interface.

2.3 Definition of Edge Types

An edge type must be defined by annotating an *interface* with the custom @Edge annotation. Only the name of the edge type is determined directly by the JAVA representation. The @Edge annotation has additional arguments to define its target and multiplicity, and to indicate whether it is ordered or not.

The annotated interface does not yet define the source of the edge. Instead, it defines an abstract edge, which can be instantiated with an arbitrary source. Each node type (or more precisely, the JAVA representation of it) that implements the edge interface provides a new source for the edge type.

[1] We allow abstract classes to define concrete node types, and vice versa.

Example 2.3.6: *(edge type example)*
The following piece of code defines the edge types writtenBy, which connects a Book to its Author, and contains, which connects a Comic to its Pictures. The multiplicity indicates the number of targets that a single source may be connected to, which is exactly one for writtenBy, and arbitrarily many for contains. Also, contains is ordered.

```
1 @Edge(target = Author.class, min = 1, max = 1)
2 public interface WrittenBy { ...
3
4 @Edge(target = Picture.class, min = 0, max = Multiplicity.MANY, ordered = true)
5 public interface Contains { ...
```

Note that to make these definitions complete, Book has to implement writtenBy, and Comic has to implement contains.

In the JAVA code that will be produced for transformation rules, instance edges will not be represented on their own, but are instead assumed to be stored by their source nodes.

2.4 Definition of Attributes

An attribute type must be defined by annotating a getter *method* with the custom @AttributeGet annotation. The name, source (the node class/interface in which the method is declared) and type (the return type of the method) of the attribute are all determined directly by the JAVA representation. Our framework does not yet support multiplicity or orderedness of attributes.

Example 2.4.7: *(attribute example)*
The following piece of code defines the text attribute name for Authors on the left, and the integer attribute price for Books on the right. Note that the attribute name is obtained by removing the leading 'get' from the method name, and putting the first character in lower case.

```
1 @Node
2 public class Author {
3     ...
4     @AttributeGet
5     public String getName();
```

```
1 @Node(isAbstract = true)
2 public class Book implements writtenBy {
3     ...
4     @AttributeGet
5     public int getPrice();
```

2.5 Definition of Manipulation Methods

The JAVA code for transformation rules needs to be able to modify the graph. Instead of exposing the actual implementation, our framework defines a number of operations that may be implemented by the JAVA code. Each operation has its own custom annotation, and can only be attached to a method of a certain type and a certain behavior[2].Our framework makes the following operations available:

[2] The type is enforced statically, but the behavior is not; instead, it is currently the responsibility of the user to provide a method with the correct behavior.

- For nodes: *creation*, *visiting* all nodes of specific type.
- For all edges: *visiting* all target nodes.
- For edges (with maximum multiplicity 1): *creation*, *getting* the target node.
- For unordered edges: *addition* of a new target, *removal* of a given target, *clearing* all targets at once, *replacing* a given target with another one, checking if a given node *occurs* as a target, getting the *number* of connected targets.
- For ordered edges: *insertion* of a new target at a given index, *removal* of a given index, *getting* the target at a given index, *replacing* the target at a given index, *clearing* all targets at once, checking if a given node *occurs* as a target, getting the *number* of connected targets.
- For attributes: *getting*, *setting*.

The user is free to implement as few or as many operations as desired, but if insufficient operations are available, JAVA code cannot be produced for certain rules any more. Some operations are optimizations only, for instance computing the size of an edge is a more efficient version of increasing a counter when visiting all the targets one by one. If both are available, the efficient version will always be used.

Example 2.5.8: *(operations example)*
The following piece of code defines the manipulation methods for the contains edge type. Insertion of an element at a given index (@EdgeAdd), removal of an element at a given index (@EdgeRemove), getting an element at a given index (@EdgeGet), and visiting all elements (@EdgeVisit) are declared. Visiting makes use of the predefined class GraphVisitor, which basically wraps a method that can be applied to an edge target into an interface.

```
1  @Edge(target = Picture.class, min = 0, max = Multiplicity.MANY, ordered = true)
2  public interface Contains {
3      @EdgeAdd
4      public void insertPicture(int index, Picture picture);
5
6      @EdgeRemove
7      public void removePicture(int index);
8
9      @EdgeGet
10     public Picture getPicture(int index);
11
12     @EdgeVisit
13     public GraphVisitor.CONTINUE visit(GraphVisitor<Picture> visitor)
14     throws GraphException;
15 }
```

2.6 Invasiveness

To apply graph transformation rules on an existing JAVA program with our approach, the following modifications have to be made:

- A @Node annotation must be added to each intended node class and interface.
- For each edge type, a dedicated interface must be defined, and an @Edge annotation must be added to it.
- For each required operation, a manipulation method must be made available. This may either involve annotating an existing method with the appropriate annotation, or defining a new method and then annotating it.

These modifications only enrich existing code with meta data, and can safely be applied to any JAVA program. We therefore call our approach *non invasive*. This should not be confused with non modifying, as annotations and edge interfaces still have to be added, and additional manipulation methods need to be implemented as well.

3 Transformation Language

With our approach, we intend to make graph transformation available for JAVA programmers. For this purpose, we have defined a custom *hybrid* transformation language, called CHART. On the one hand, CHART has a textual JAVA like syntax and a sequential control structure. On the other hand, it has declarative matching and only allows graph updating by means of simultaneous assignment.

In the following sections, we will introduce CHART and briefly go into its semantics. In Section 3.1, we first present the rule based structure of CHART. In Sections 3.2, 3.3 and 3.4, we describe the main components of CHART, which are matching, updating and sequencing, respectively. In Section 3.5, we shortly describe the semantics of CHART.

3.1 Rule Structure

A CHART transformation is composed of a number of transformation rules, and can be started by invoking any one of them. Each rule has a signature, which declares its name, its input and its output. Multiple (or no) inputs and outputs are allowed, and each can be an arbitrary (typed) value (see Definition 2.1.3).

Example 3.1.1: *(rule signature)*
The following piece of code declares the rules findRich, addPicture and addPictures. Set types are denotes by trailing '{}', and list types by trailing '[]'. void denotes that a rule has no return type.

```
1 rule Author{} findRich(int price) { ...
2 rule int addPicture(Comic comic, Picture picture) { ...
3 rule void addPictures(Comic comic, Picture[] pictures) { ...
```

The body of a rule consists of a match block, an update block, a sequence block and a return block. The blocks are optional (and no more than one of each type can be used in a rule), and can only appear in the order match-update-sequence-return.

3.2 Match Blocks

A match block searches for all the possible values of a given set of match variables such that a given set of equations is satisfied. The equations are either boolean expressions, or 'foreach' statements that lift equations to all elements of a collection. A match block corresponds to the left-hand-side of a rewrite rule, represented textually.

Example 3.2.2: *(match block)*
The following match block finds all authors that have written a book with a price higher than a certain threshold. Line 3 specifies that the block searches for a set of Authors, which will be stored in the variable rich. Line 4 specifies that the equations on lines 5-6 must hold for all[3] these authors. Lines 5-6 specify that for each author there must exist a book (line 5) with a price higher than the threshold (line 6).

```
1  rule Author{} findRich(int price) {
2      Author{} rich;
3      match (rich) {
4          foreach (Author author : rich) {
5              Comic comic;
6              comic.price > price;
7          }
8      }
9      return rich;
10 }
```

A match block can either *fail*, if no valid values for the match variables can be found, or *succeed*, with a single binding for the match variables. If multiple bindings are possible, then one is chosen, and the other possibilities are thrown away. The semantics does not prescribe which binding must be returned.

3.3 Update Blocks

An update block changes the instance graph, and it is the only place where this is possible. It consists of a list of create statements, which are evaluated sequentially, and a list of update statements, which are evaluated *simultaneously*, but after the creations. Each update statement may modify a single field in the graph, and there may not be two update statements that change the same field. An update block corresponds to the differences between the right- and left-hand-side of a rewrite rule.

Example 3.3.3: *(update block)*
The following update block creates a new comic (line 7), which is the same as an existing one, but extended with one picture (line 9). The old comic is decreased in price (line 12), and the old price of the old comic is returned (lines 13 and 15). The match block ensures that the rule can only be applied to comics with a price greater than 1. Note that the right-hand-side of line 13 is evaluated in the graph before the update block, and therefore refers to the old price, instead of the new one.

[3] In our approach, a foreach over a match variable always finds the largest possible set/list only.

```
1  rule int addPicture(Comic comic, Picture picture) {
2      int old_price;
3      match () {
4          comic.price > 1;
5      }
6      update let {
7          Comic new_comic = new Comic();
8      } in {
9          new_comic.contains = comic.contains + [picture];
10         new_comic.price = comic.price;
11         new_comic.writtenBy = comic.writtenBy;
12         comic.price = comic.price − 1;
13         old_price = comic.price;
14     }
15     return old_price;
16 }
```

3.4 Sequence Blocks

A sequence block establishes flow of control, and is the only block in which other rules can be invoked. It contains sequential, JAVA like statements, such as 'if' and 'foreach' (our notation for 'for'), and custom statements for managing rule failure, such as 'try' and 'repeat' (see below).

Example 3.4.4: *(sequence block)*
The following sequence block repeatedly adds pictures to a comic by invoking addPicture (line 4). The 'pictures[2:]' that appears in line 3 is a range selection, which selects all elements starting from index 2. The try statement in line 4 is used to catch a possible failure of addPicture. Because of it, if addPicture fails, execution still continues with the next iteration of the loop.

```
1  rule void addPictures(Comic comic, Picture[] pictures) {
2      sequence {
3          foreach (Picture picture : pictures[2:]) {
4              try { addPicture(comic, picture); }
5          }
6      }
7  }
```

Because a sequence block can contain rule calls, it can also *succeed* or *fail*. If a rule call fails, one of the following things will happen:

- If the rule call is surrounded by a 'try' or 'repeat', then the failure is *caught*, and the remainder of the sequence block is executed normally.
- If the failure is not caught, and the rule in which the sequence block appears has not yet changed the graph, then the sequence block (and consequently the rule itself) *fails*. This is the same kind of failure as in the match block.

– If the failure is not caught, and the graph has already been changed, then the failure is *erroneous*, and the transformation as a whole stops with an exception. This is because our approach does not support roll-back.

3.5 Semantics

A full operational semantics of CHART is available in [6]. Below, we will restrict ourselves to a brief explanation of the top level functions of our semantics, which define the meaning of match blocks, update blocks, sequence blocks, and rule systems.

Notation 3.5.5: *(preliminaries)*
In the following, let $\mathcal{X}$ denote the set of variables, $\ell(B)$ denote the set of *lists* (i.e. ordered sequences) over B, and *Autom* denote the universe of automata.

Match blocks. A match block consists of a list of match statements, which are assumed to be defined by the *MatchStat* set. Its meaning is determined by the

$$match : \wp(\mathcal{X} \hookrightarrow \mathcal{V}) \times \ell(MatchStat) \times \mathcal{G} \to \wp(\mathcal{X} \hookrightarrow \mathcal{V})$$

function, which computes the set of *all* valid matches (=variable bindings) relative to a given input graph. An implementation only has to return one of these matches, but the semantics takes all of them into account. Later, we will enforce that all choices converge to a single output graph.

The *match* function is initialized with a single match, which is the input variable binding of the rule in which it appears. It then processes each match statement iteratively. If the statement is a match variable, then each input match is extended will all possible values of that variable. If the statement is an equation (or 'foreach'), then the input matches are filtered, and only those that satisfy the equation are kept.

Update blocks. An update block consists of a list of create statements and a list of update statements, which are assumed to be defined by the *CreateStat* and *UpdateStat* sets, respectively. Its meaning is determined by the

$$update : (\mathcal{X} \hookrightarrow \mathcal{V}) \times \ell(CreateStat) \times \ell(UpdateStat) \times \mathcal{G} \to (\mathcal{X} \hookrightarrow \mathcal{V}) \times \mathcal{G}$$

function, which modifies a variable binding and a graph. It first processes the create statements sequentially. Then the update statements are merged into one atomic update action, which is applied on the intermediate state in one go.

Sequence blocks. A sequence block consists of a list of sequence statements, which are assumed to be defined by the *SequenceStat* set. Its meaning is determined by the

$$sequence : \ell(SequenceStat) \to Autom$$

function, which builds an automaton that statically models the sequence block. It uses simplified sequence statements as alphabet, and a basic numbering for states only. The purpose of the automaton is to distinguish between success with or without changing the graph, and failure. For this purpose, it has three distinctive final states, and it models the conditions under which these states are reached. The automaton does not model any dynamic behavior, however, as its states do not include graphs or variable bindings.

Rule systems (1) A rule system consists of a set of rules, which are assumed to be defined by the *Rule* set. The meaning of a rule system is determined by the

$$\mathit{automs} : \wp(\mathit{Rule}) \times \mathit{Rule} \times \mathcal{G} \to \mathit{Autom} \times \mathit{Autom}$$

function, which computes two automata. The first is the *applier* automaton, which has tuples of graphs and variable bindings as states, and all possible applications of all available match and update blocks as (separate) transitions. The second is the *control* automaton, which is basically the combination of the sequence automata of all rules. The initial state of the applier automaton is the empty variable binding with the input graph, and the initial state of the control automaton is the initial state of the start rule.

Rule systems (2) The final purpose of a rule system is to transform an input graph into a single output graph. This is modeled by the semantic function:

$$[\![\cdot]\!] : \wp(\mathit{Rule}) \times \mathit{Rule} \times \mathcal{G} \hookrightarrow \mathcal{G}$$

First, the *product* automaton of the applier and control automata is built, which synchronizes on the rule calls and adds local state to the control automaton. In our approach, the final transformation has to terminate and be deterministic. If the product automaton is not confluent, acyclic and finite, then the transformation is therefore considered to be erroneous, and the semantic function does not yield a result. Otherwise, the semantics is given by the graph in the unique final state of the product automaton.

4 Experience and Evaluation

A major part of the effort has gone into the CHART compiler that generates the corresponding JAVA code. The compiler is called RDT, which stands for Rule Driven Transformer, and supports all features that have been described in this paper. We have used the RDT successfully on several smaller test cases, and more importantly, it is currently being used by three of our partners in the CHARTER project [4]. The tool will be made publicly available by the final project deliverable planned for April 2012.

Concretely, the following transformations have been built with the RDT:

- For testing purposes, we have implemented an interpreter for finite state automata, and an interpreter for a simplified lazy functional programming language. Some metrics are collected in Table 1.
- A collection of CHART rules have been produced by ATEGO to replace the JAVA code generator within ARTISAN STUDIO [2].
- A collection of CHART rules are being developed by AICAS for the purpose of optimisations and machine code generation in the JAMAICAVM byte code compiler [15]. This application is discussed in some more detail below.
- Part of the code simplification from JAVA to Static Single Assignment form within the KEY tool [3] is scheduled to be replaced by CHART rules.

These examples show that the technology can already be applied in practice. In all cases, the CHART rules are more concise than the JAVA code. The JAMAICAVM and KEY examples also show that the RDT can successfully be connected to an existing JAVA program. In the other examples, a custom JAVA program was built explicitly.

Table 1. metrics for the functional interpreter case

JAVA data (represent functional program)	21 classes, 11 interfaces, 1448 lines of code
added annotations	19 nodes, 11 edges, 49 methods, 3 auxiliary
CHART rules	53 rules, 1024 lines of code
produced JAVA code	53 classes, 7667 lines of code
analysis and compilation time	approx. 4,5 seconds (2GHZ, 4GB laptop)
execution time (50 primes computed with sieve)	approx. 1,5 seconds (2GHZ, 4GB laptop)

4.1 Using the RDT in JAMAICAVM

A complex application of the RDT is its application in the JAMAICAVM byte code compiler. This application demonstrates the strength of the synthesis of a strongly typed object-oriented programming language with a domain specific graph transformation language. The compiler implementation takes advantage of JAVA for implementing the basic graph operations, and uses the RDT for deciding what transforms should be applied (match clause), when a transform should be applied (sequence clause), and what change a transform should make (update clause).

The most general CHART rules are used in the optimization of the intermediate representation. These rules implement standard compiler optimizations such as:

- unnecessary node removal,
- expression simplification,
- duplicate check removal,
- common subexpression elimination,
- method inlining,
- loop inversion, and
- loop expression hoisting.

There are not that many optimizations, but each optimization takes in general several rules to implement it. Each rule has about 10 to 20 lines of RDT code. The generated code is approximately ten times as long. Hand coding might bring a factor of two improvement, but that would still be five times large than the CHART code.

There are many more CHART rules for translating the intermediate representation to the low-level representation: each intermediate instruction requires its own rule. These rules are simpler, so they tend to be shorter than the optimization ones. Still there is a similar ten fold expansion of code when these rules are compiled.

CHART rules are also used in the optimization of the low-level representation. These rules tend to be more instruction dependent. Some of the simpler intermediate optimizations are applicable on the low-level too, because they do not depend on the actual instructions and operate on an abstract representation of the graph. This works because both graphs share a common parametrized subclass structure using JAVA generics. Again, a ten fold expansion is typical.

Performance measurements have not yet been made, but there is no noticeable slowdown for the couple of optimizations that have been converted to rules. In fact, the new compiler is faster than the previous one. However, this is probably due to improvements in the graph structure. Certainly, the performance is within acceptable bounds.

In general, CHART rules are easier to reason about than JAVA code for two reasons. Firstly, the code is much shorter. Secondly, the language itself is declarative instead of imperative. The next challenge will be to provide theory and methods for reasoning about the correctness of rules written in CHART.

4.2 Evaluation

The cases reported above provide first evidence of the advantages and disadvantages of the CHART approach. We can make the following observations:

- The approach is flexible enough to be applicable in several, quite different contexts: model-to-text generation for ARTISAN, code compilation and optimisation for JAMAICAVM and text-to-text transformation in the case of KEY. The latter is done on the basis of its JAVA syntax tree structure.
- The non invasive nature of the approach enables a partial or stepwise adoption of CHART. Indeed, the fact that all data stay within the user domain was the reason to adopt CHART for KEY, where it was initially not foreseen in the project proposal.
- CHART enhances productivity by a factor of ten, measured in lines of code. This metric should be taken with a grain of salt as the generated code is less compact than hand-written code for the same purpose would have been; however, as argued in Section 4.1, even taking this into account the ratio in lines of code is a factor 5.
- The generated code been applied to very large graphs (in the order of 10^5 – 10^6 nodes) with a performance comparable to the replaced handwritten JAVA code.

All in all, we feel that these are very encouraging results.

4.3 Future Work

Although, as shown above, CHART has already proved its worth in practice, there is obviously still a lot of work that can be done to strengthen both the formal underpinnings and the practical usability.

- The formal semantics presented in this paper enables reasoning on the level of the CHART rules. As a next step, we intend to develop this into a theory that allows the CHART programmer to deduce confluence and termination of his rule system. A more ambitious goal is to be able to prove semantic preservation of model transformations in CHART (see, e.g., [8]).
- Based on the formal semantics, we plan to formally verify that the RDT actually produces correct code. Code correctness can be addressed on different levels: firstly, by requiring it to run without errors; and secondly, to actually implement the transformation specified in CHART. The second especially is a major effort, analogous to the Verified Compiler research in, e.g., [10], and will be addressed in a separate follow-up project.
- On the pragmatic side, the RDT needs further experimentation with an eye towards efficiency. This is likely to lead to improvements in performance of the generated code.
- The CHART language can be extended with additional functionality, as well as syntactic sugar for the existing features.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010), http://www.eclipse.org/modeling/emft/henshin/
2. Artisan studio (2011), http://www.atego.com/
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007), http://www.key-project.org
4. Charter: Critical and hight assurance requirements transformed through engineering rigour (2010), http://charterproject.ning.com/page/charter-project
5. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–646 (2006)
6. de Mol, M., Rensink, A.: Formal semantics of the CHART transformation language. CTIT technical report, University of Twente (2011), http://www.cs.utwente.nl/~rensink/papers/chart.pdf
7. The FUJABA Toolsuite (2006), http://www.fujaba.de
8. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 183–198. Springer, Heidelberg (2010)
9. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
10. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
11. Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H.: A systematic approach to metamodeling environments and model transformation systems in VMTS. Electr. Notes Theor. Comput. Sci. 127(1), 65–75 (2005), http://www.aut.bme.hu/Portal/Vmts.aspx?lang=en
12. Akehurst, D.H., Bordbar, B., Evans, M.J., Howells, W.G.J., McDonald-Maier, K.D.: SiTra: Simple Transformations in Java. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 351–364. Springer, Heidelberg (2006)
13. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (2011), http://www.omg.org/spec/QVT/1.1/
14. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
15. Siebert, F.: Realtime garbage collection in the JamaicaVM 3.0. In: Bollella, G. (ed.) JTRES. ACM International Conference Proceeding Series, pp. 94–103. ACM (2007), http://www.aicas.com
16. Taentzer, G.: What algebraic graph transformations can do for model transformations. ECEASST 30 (2010)
17. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming 68(3), 187–207 (2007), http://www.eclipse.org/gmt/VIATRA2/

```
/**
 * ============================================================
 * CLASS generated for RULE 'examples.comic.generated.addPicture'.
 * ============================================================
 */
public class addPicture extends RDTRule.RDTRule1<Integer> {
    ...
    /** Finds a single match for the rule. */
    private boolean match() throws GraphException {
        // check (V1_comic.price > (1));
        if (!(V1_comic.getPrice() > 1)) {
            return false;
        }
        // Report success.
        return true;
    }

    /** Applies the update block of the rule on an earlier found match. */
    private void update() throws GraphException {
        // Store for postponed graph updates.
        final List<Closure> postponed = new ArrayList<Closure>(25);
        // Comic V4_new_comic = new Comic();
        final Comic V4_new_comic = Comic.createComic(this.context.getSupport());
        // V4_new_comic.contains = V1_comic.contains + [V2_picture];
        final GraphVisitor<Picture> t1 = new GraphVisitor<Picture>() {
            @Override
            public CONTINUE apply(final Picture node) throws GraphException {
                postponed.add(new Closure() {
                    @Override
                    public void apply() throws GraphException {
                        V4_new_comic.insertPicture(-1, node);
                    }
                });
                return CONTINUE.YES;
            }
        };
        ...
    }
    ...
}
```

Part of the generated code for the addPicture rule.