The Best of Both Worlds: Graph Transformations and Java

Sven Konings
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
s.l.konings@student.utwente.nl

ABSTRACT
This paper introduces an approach for combining graph transformations and Java programs. The goal is to add Java based functionality directly to graph transformations and vice-versa. Using this approach, one could enhance their existing Graph transformation models with Java code. Or one could treat Java data as a graph and apply graph transformations to it. This can make the code shorter and more expressive. This approach relies on annotations to identify the intended graph structure of the Java program and uses annotated user defined methods to manipulate the graph structure. The graph structure can be manipulated by applying the graph transformation rules defined in GROOVE. GROOVE is a general purpose graph transformation tool set. This paper researches which different types of graph transformations implemented in GROOVE are applicable to Java programs and how efficient they are. This will be done by creating a GROOVE extension that treats Java programs as graphs.

The approach has complete support for the different types of graph transformations contained in GROOVE. The approach does not scale well compared to GROOVE.

Keywords
Graphs, Graph transformations, GROOVE, Java, Java annotations

1. INTRODUCTION
Graphs are a well-established and powerful tool to represent various data structures. They are used to model pairwise relations (edges) between objects (nodes). Manipulations to these graphs can be regarded as a set of graph transformations. For example, an ADHOC network determines its routes using a gossiping protocol. This network can easily be represented as a graph and the changes to the routes as graph transformations. Tools are available to apply graph transformations to graphs. The tool that will be used for this project is GROOVE [1]. GROOVE is a general purpose graph transformation tool set that contains many different types of graph transformations.

Another well-established and powerful tool is Java. Java is a general purpose programming language and has a broader scope than graph transformations. It allows for programming additional functionality like network communication and graphical interfaces.

When combining both domains one can enhance their existing graph transformation models with functionality written in Java code. This makes these models reusable for programming purposes. One can also use graph transformations to express certain problems in a different domain while using their existing Java code. This can make code shorter and more expressive.

An example where combining graph transformations and Java would be applicable is if one has an existing model of the states of a game using graph transformations. One could enhance their existing models with Java code to create a user interface for this game. The other way around would be when one is implementing a simulation of the gossiping protocol written in Java. One could use graph transformations to express the changes in this network. Graph transformations work really well here because this kind of network is very similar to a graph. Therefore, using graph transformations could make implementing the simulation easier and the resulting code shorter.

This paper will research which types of graph transformations implemented in GROOVE can be applied to Java. To do this, a GROOVE extension that uses Java programs as graphs will be implemented. This means that GROOVE’s existing graph structure, which is built in Java, will be replaced by a structure that uses the Java program itself. This could potentially be faster and more scalable. Therefore, the performance of every applicable type of graph transformation will also be researched and compared to the performance of GROOVE’s existing graph structure.

Some initial design decisions were based on the approach proposed by Mol et al. [5]. This approach also treats Java data as a graph. The details of this approach will be discussed in section 2. This approach mentions that applying graph transformations directly to the Java data itself, instead of converting the data to a graph and then converting the graph back to Java, is more efficient and flexible. Additionally, letting the user manually specify the relation between the Java data and the graph allows for a flexible structure which works well for both domains.

The approach will be implemented in steps. First an initial version of the GROOVE extension will be made containing only nodes and directed edges. Then the implementation will be extended incrementally with extensions like edge labels, node labels, flags, attributes, inheritance, edge multiplicity and multigraphs. At the end the types of graph
transformations that are applicable will be analyzed. If it is not possible to implement an extension the reasons why will be analyzed.

The implementation will be validated by implementing two examples. The first example is to extend the GROOVE model of the river crossing game with a textual user interface (TUI) written in Java. In the river crossing game there is a wolf, a goat and a cabbage that have to be transported from the left bank to the right bank. If the wolf and the goat are left alone, the wolf will eat the goat. The same happens when the goat and the cabbage are left alone. The GROOVE model models the different states of the game using graph transformation rules.

The second example will be to apply graph transformation rules to a Java program. The Java program will be a network of connected nodes that keeps changing over time. This network will be extended with graph transformation rules that can modify the network based on patterns that might occur. These example validate that the approach can be used for existing GROOVE models and for existing Java programs. The second approach also validates that the network can be changed by Java code and by graph transformations simultaneously.

The performance of every applicable type of graph transformation will also be analyzed. This will be done by creating a large example that specifically uses that type of graph transformation and measuring the time it takes to apply the graph transformation rules. The time of the implementation will be compared to the time of GROOVE’s existing graph structure.

1.1 Paper outline
In section 2 the basic concepts of Java and GROOVE will be introduced. In section 3 the related works will be discussed. Section 4 introduces the approach. The validation of the approach will be discussed in section 5 and the conclusion in section 6.

2. BACKGROUND
2.1 Java
Java is an imperative general-purpose programming language that is concurrent, class-based and object-oriented. It is one of the most commonly used programming languages.

2.1.1 Annotations
In Java, an annotation is a form of metadata that can be added to Java source code. Classes, methods, variables, parameters and packages can be annotated. Java annotations can be processed at compile time to generate classes or files based on the annotated source code.

2.1.2 Reflection
Reflection is a term used for code which is able to inspect other code in the same system (or itself). This can be used in combination with annotations to call the annotated methods.

2.2 GROOVE
GROOVE is a general purpose graph transformation tool set that uses simple labeled graphs and single push-out (SPO) transformation rules. The core functionality of GROOVE is to recursively apply all rules from a predefined set to a given start graph, and to all graphs generated by such applications. This results in a state space consisting of the generated graphs.

2.2.1 Graphs
An example of a simple labeled graph would be:

Figure 1. Example simple labeled graph

This graph G consists of a set of nodes N = {A, B, C}, a set of edges from source node to target node E = {A \rightarrow B, B \rightarrow C} and a set of labels L = \{x, z\}. In this graph each edge is associated with a label. This is a simple graph because there can only be a single edge with the same source, label and target. GROOVE also has support for special edges called flags, but for all intended purposes these can be treated as edges with the same source and target node. The exact graph definition used in this paper will be explained in section 4.

2.2.2 Type graphs
Type graphs specify which types of nodes, edges and labels are allowed in a graph and where. An example type graph would be:

Figure 2. Example type graph

Each graph instance of this type graph can only have nodes of type A, B, C or D. Each node of type A can only have an edge with label x to nodes of type B and an edge with label y to nodes of type C. This also applies to all the other node types. The exact type graph definition used in this paper will be explained in section 4.

2.2.3 Graph transformation rules
GROOVE modifies graphs using graph transformations. The transformations to apply are specified using graph transformation rules. A GROOVE graph transformation rule consists of nodes and edges, labeled as one of these five types:

Readers These are nodes and edges that must be present to make the rule applicable. They are colored black.

Erasers These are nodes and edges that must be present to make the rule applicable and will be deleted when...
the rule is applied. They are colored blue with a dashed outline.

Creators: There are nodes and edges that will be created when the rule is applied. They are colored green with a thick outline.

Embarques: These are nodes and edges that must not be present to make the rule applicable. They are colored red with a thick, dashed outline.

Conditional creators: These nodes and edges that must not be present to make the rule applicable and are created when the rule is applied. They are colored green with a green-and-red outline.

Rules can also be typed or untyped. Untyped rules can also contain nodes and edges without a type. Typed rules must always specify a type and adhere to the type graph. An example of a (typed) rule would be:

![Figure 3. Example rule](image)

This rule would be applicable when nodes of type A and B exist, there is an edge labeled with x from A to B and there is not an edge labeled with y from B to a node with type D. If the rule is applied B and the edge from A to B will be deleted and a new node of type C will be created with an edge labeled with y from A to C.

3. RELATED WORK

Mol et al. [3] describes the CHART Transformation Language. This is a graph transformation language for Java. CHART has a textual Java like syntax and a sequential control structure combined with declarative matching and graph updating by the means of simultaneous assignment. CHART uses rules with match, update and sequence blocks to express changes to the graph. Some of the initial design decisions were based on this approach. In contrast to the GROOVE rules, the CHART rules are compiled to Java code before they are applied.

AGG [6] is a rule-based graph transformation language. The graphs used in AGG have support for Java objects and types as attributes and the rules have support for Java expressions. The Java methods associated with the rules can be evaluated while exploring the graph and the results can be used for the attributes of the nodes. However, the communication can not be modified directly by the Java code so the communication is one-way only.

There are multiple other model transformation frameworks which use graph transformations, some of which allow the transformations to be compiled to code so they can be combined with existing code. However these model transformation frameworks use their own data structures for the actual graph representation. Examples include FUJABA [2], PROGRES [11], GReAT [2], ATL [1], VIARTRA2 [1] and CheckVML [12].

There are also some graph transformations languages that can be compiled to code. However these also use their own data structures for the actual graph representation. Examples include GrGen.NET [9] and Optimix [8].

There is also research about translating Java code to graph transformation systems [3]. However, this is only a one-way translation.

4. APPROACH

The idea of our approach is that the user specifies the basic structure of the graph using annotations. The annotations specify which parts of the Java program are supposed to be treated as nodes or edges and which methods can be used to modify the graph (and thus the Java data). Our approach has support for simple graphs containing nodes and binary directed edges with edge labels. During compile time the Java code is processed using an annotation processor. The annotation processor processes the annotations and checks the annotated source code to see if the annotations are valid. When all annotations are valid a GROOVE typegraph is constructed based on these annotations. The constructed typegraph uses a subset of the typegraph features available in GROOVE. The implementation does not support features like abstractness, inheritance or edge multiplicity. This leads to the following typegraph definition:

Definition 4.1 (type graph):
A type graph is a structure \((\mathcal{T}_n, \mathcal{T}_e, L, \text{source}, \text{target}, \text{label})\), in which:

- \(\mathcal{T}_n, \mathcal{T}_e, L\) are the disjoint sets of node types, edge types and labels, respectively;
- \(\text{source} : \mathcal{T}_e \rightarrow \mathcal{T}_n\) associates each edge type with a source node type;
- \(\text{target} : \mathcal{T}_e \rightarrow \mathcal{T}_n\) associates each edge type with a target node type;
- \(\text{label} : \mathcal{T}_e \rightarrow L\) associates each edge type with a label;

A running Java program is treated as an instance of the generated graph. GROOVE uses a built in graph model called host graphs to define graphs which can be transformed by graph transformations. The graph in our implementation is a custom implementation of a host graph, which uses reflection to call annotated methods. Using these methods the custom graph implementation can view and modify the underlying Java data and treat it as a graph. The custom implementation supports almost all host graph features. Our implementation does not support numbered nodes. However, this feature is not needed for our implementation. Our implementation also does not support creating nodes without a type. Nodes in our graph always have to be typed. The labels of the edges are determined by their edge type, which is similar to the behavior in GROOVE. The resulting definition of a graph instance is:

Definition 4.2 (graph):
A graph \(\mathcal{G}\) is a structure \((N, E, \text{type}_n, \text{type}_e, \text{source}, \text{target})\), in which:

- \(N, E\) are the disjoint sets of nodes and edges, respectively;
- \(\text{type}_n : N \rightarrow \mathcal{T}_n\) associates each node with a node type;
• \(\text{type} : E \rightarrow T_n\) associates each edge with an edge type;
• \(\text{source} : E \rightarrow N\) associates each edge with a source node;
• \(\text{target} : E \rightarrow N\) associates each edge with a target node;
• \(\text{sourcetype}(\text{type}_n(E)) = \text{type}_n(\text{source}(E))\) for each edge holds that the source type node of the associated type edge is equal to the type node of the associated source node;
• \(\text{targettype}(\text{type}_n(E)) = \text{type}_n(\text{target}(E))\) for each edge holds that the target type node of the associated type edge is equal to the type node of the associated target node;

The method annotated with the \textit{@NodeCreate} annotation is used to create a new node instance of this node type. This method should return a boolean whether the deletion was successful. The user implementation should also remove the deleted instance from the \textit{Set} returned by the \textit{@NodeVisit} method if the deletion was successful.

Every class that represents a node type has to contain these three methods, otherwise they are not a valid node type. The nodes that are in the graph are allowed to be modified in the Java program itself, even without calling one of the annotated methods. This also means that even when nodes have been created or deleted they can be removed or re-added by the Java program. By using the \textit{@NodeVisit} method the implementation knows which nodes are in the graph, even if it was modified by the program itself. This allows for a flexible structure in which the data can be modified in both domains.

4.2 Edges

A node type can also contain annotated methods that define an edge type \(T_e\). The \textit{sourcetype} is determined by the node type that contains the methods, the \textit{label} is determined by the value of the edge type annotations and the \textit{targettype} is determined by the class that is used by the method. The target class should always be a node type. Edges are always directed. The annotated edge methods belong to a source node instance. This is also the \textit{source} of an edge instance \(E\). The \textit{target} of \(E\) is the target node instance given to or returned by the methods. Because the methods belong to a node instance none of them are static. They are called to view and/or modify the edge instances associated with the node instance. The implementation can call the methods using a node instance of the \textit{sourcetype}.

Example 4.2.1 (edge type example):
The following piece of code defines a node type \(T_n\) with annotated methods for visiting (\textit{@VisitNode}), creating (\textit{@CreateNode}) and deleting (\textit{@DeleteNode}) the node.

\begin{verbatim}
public class Author {
    // Node methods omitted.

    @Node("Author")
    public class Author {
        // Node methods omitted.

        @EdgeCreate("writtenBy")
        public Set<Book> visitBooks() {
            // Edges omitted.
        }

        @EdgeCreate("writtenBy")
        public boolean addBook(Book book) {
            // Edges omitted.
        }

        @EdgeCreate("writtenBy")
        public boolean deleteBook(Book book) {
            // Edges omitted.
        }
    }
}
\end{verbatim}

The method annotated with the \textit{@EdgeCreate} annotation is used to create new edges between the node instance and the target node instance. Therefore the method has to return a \textit{Set} containing all the target node instances. The target node type of this method is determined by the class contained in the \textit{Set}. All the edges are simple. This means that there can only be a single edge with the same source, label and target. This is enforced by returning a \textit{Set}, so the same target can never occur twice.

The method annotated with the \textit{@EdgeDelete} annotation is used to create new edges between the node instance and the given target node instance. This method should return

As shown in example 4.1.1 the class contains both static and non-static methods. The methods that are static belong to the node type and the methods that are not belong to the node instances. The method belonging to the node type have to be static so the implementation can call them without needing a node instance. All annotated methods have to be public.

The method annotated with the \textit{@NodeVisit} annotation is used to retrieve all node instances belonging to the node type. The method has to return a \textit{Set} containing all node instances of this node type. By calling this method the implementation knows which nodes of this node type are in the graph.

The method annotated with the \textit{@NodeCreate} annotation is used to create a new node instance of this node type. This method should return the created instance so the host graph implementation knows which node has been created by this method call. The user implementation should make sure the new instance is also added to the \textit{Set} returned by the \textit{@NodeVisit} method.

The method annotated with the \textit{@NodeDelete} annotation belongs to a node instance. It is used to delete the node instance it belongs to. The method should return a boolean whether the deletion was successful. The user implementation should also remove the deleted instance from the \textit{Set} returned by the \textit{@NodeVisit} method if the deletion was successful.

4.1 Nodes

A node type \(T_n\) must be defined by annotating a Java class with a \textit{@Node} annotation. This annotation also has a value which represents the identifier of the type node. A node instance \(N\) is an instance of the annotated class.

Example 4.1.1 (node type example):
The following piece of code defines a node type \(T_n\) with annotated methods for visiting (\textit{@VisitNode}), creating (\textit{@CreateNode}) and deleting (\textit{@DeleteNode}) the node.

\begin{verbatim}
public class Book {
    @Node("Book")
    public static Book {
        @NodeVisit
        public static Book {
            @NodeCreate
            public static Book {
                @NodeDelete
                public boolean deleteNode() {
                }
            }
        }
    }
}
\end{verbatim}

The following piece of code defines a node type \(T_n\) with annotated methods for visiting (\textit{@VisitNode}), creating (\textit{@CreateNode}) and deleting (\textit{@DeleteNode}) the node. This method should return the created instance so the host graph implementation knows which node has been created by this method call. The user implementation should make sure the new instance is also added to the \textit{Set} returned by the \textit{@NodeVisit} method.

The method annotated with the \textit{@NodeCreate} annotation is used to create new nodes between the node instance and the given target node instance. This method should return

\begin{verbatim}
public class Book {
    @CreateNode
    public boolean createNode() {
    }
}
\end{verbatim}

The following piece of code defines a node type \(T_n\) with annotated methods for visiting (\textit{@VisitNode}), creating (\textit{@CreateNode}) and deleting (\textit{@DeleteNode}) the node. This method should return the created instance so the host graph implementation knows which node has been created by this method call. The user implementation should make sure the new instance is also added to the \textit{Set} returned by the \textit{@NodeVisit} method.

The method annotated with the \textit{@NodeCreate} annotation is used to create new nodes between the node instance and the given target node instance. This method should return

\begin{verbatim}
public class Book {
    @NodeCreate
    public boolean createNode() {
    }
}
\end{verbatim}

The following piece of code defines a node type \(T_n\) with annotated methods for visiting (\textit{@VisitNode}), creating (\textit{@CreateNode}) and deleting (\textit{@DeleteNode}) the node. This method should return the created instance so the host graph implementation knows which node has been created by this method call. The user implementation should make sure the new instance is also added to the \textit{Set} returned by the \textit{@NodeVisit} method.

The method annotated with the \textit{@NodeCreate} annotation is used to create new nodes between the node instance and the given target node instance. This method should return

\begin{verbatim}
public class Book {
    @CreateNode
    public boolean createNode() {
    }
}
\end{verbatim}

This graph is based on the results of the annotated methods which will be discussed in the next sections.
a boolean whether the creation was successful. The user implementation should also add the target node instance to the Set returned by the @EdgeVisit method of this edge type if the creation was successful.

The method annotated with the @EdgeDelete annotation is used to delete the edge between the node instance and the given target node instance. The method should return a boolean whether the deletion was successful. The user implementation should also remove the target node instance from the Set returned by the @EdgeVisit method of this edge type if the deletion was successful.

Every edge type, thus every combination of source node type, label and target node type, has to contain these three methods. Otherwise it is not considered a valid edge. The edges in the graph are also allowed to be modified in the Java program itself.

The edge implementation does not enforce edge multiplicity. In other words, there is no lower or upper bound on the amount of edges a node has. The user can however enforce a multiplicity in their implementation by returning false for the @EdgeCreate and @EdgeDelete methods when the lower or upper bound would be invalidated.

### 4.3 GROOVE rules

Now we have a graph implementation that matches the one in GROOVE, but we still need to define graph transformation that can be applied to this implementation. To do this we need to have a GROOVE model with the same type graph. The typegraph of the Java program can be exported to GROOVE, or one could use an existing GROOVE model that already has a type graph and modify the annotations of the Java program accordingly. The GROOVE type graph may not contain unsupported features and must adhere to definition 1.

Once the user has a valid GROOVE model, it can be loaded in the implementation. A GROOVE model contains a start graph, which is the initial instance of the graph. This start graph will be initialized by the implementation by creating the specified nodes and edges. This will be done by calling the annotated method reflectively. Once this is done the rules contained in the model can be matched. The implementation contains predefined methods for finding and applying rules. There is also a method to do this automatically based on the priority of the rules, until it is no longer possible. By applying these rules the Java data changes and thus the graph transformation and the Java program have been combined.

### 5. VALIDATION

The source code of the implementation, including all examples mentioned in this section, can be found on Github.

The implementation is compatible with the GROOVE type graph. Type graphs defined in GROOVE can be used in the implementation and type graphs generated by the implementation can be used by GROOVE. The implementation has also almost completely implemented GROOVE’s graph model. This means almost all graph transformation used in GROOVE can be used in the implementation. The only exceptions are the use of untyped graphs, rules or nodes and the use of unsupported type graph features.

The first example to validate the implementation is to extend the GROOVE model of the river crossing game with a textual user interface (TUI) written in Java.

Figure 4 shows the type graph of the river crossing model. Each type node of this graph has been implemented by a Java class. The TUI implementation loads the GROOVE model. The start graph, in which all passengers are on the left bank, is automatically initialized. The TUI accepts four different commands: go, cabbage, goat and wolf. The command determines who crosses the river (go means an empty boat crosses the river) and the TUI applies the respective rules of the model based on the command. After every command, the rules whether one of the passengers can be eaten or the game is finished are checked. If a match has been found the game is over, otherwise the user can input the next command. The example works as intended and, as seen from the TUI implementation, does not require a lot of Java code while being fully functional.

The second example is a network of connected nodes that changes over time. The network is written in Java and will be extended with graph transformations that can change the network based on patterns that might occur. The network consists of a single node type that can have edges to nodes of the same type. Every 10 seconds, for every combination of nodes, there is a change that the edge changes. If there is no edge, it will be created, and otherwise it will be deleted. This has the result that the network constantly changes. When the network is initialized there are 10 node instances, so 100 combinations of nodes. The user can type in the name of a rule of the associated GROOVE model to match it in the network. It is possible that the pattern for which the rule would be applicable does not exist, in that case no match is found. Otherwise, the rule is applied. The user implementation of the methods uses synchronized sets to avoid concurrency issues. The example works as intended and shows it is possible to change the graph within Java and use graph transformations at the same time. The graph transformations could even be automatically applied after a change in the network to immediately change matched patterns when they occur.
Furthermore, the performance is validated. The first performance test is done by applying the rule depicted in Figure 5 repeatedly to a graph. This rule creates a new node at the end of a chain of nodes. By applying it repeatedly, the chain keeps getting longer and longer.

Figure 5. Performance test rule

Figure 6. Performance test results

Figure 6 shows the time it took to apply the rule in nanoseconds. N is how many times the rule was applied. As shown in the figure, the incoming edges take a lot more time to calculate than the outgoing edges. The time it takes to calculate the outgoing edges is independent of the graph size, while the time it takes to calculate the incoming edges increases.

6. CONCLUSION

This paper introduces an approach for combining graph transformations and Java. The implementation successfully applied the different types of graph transformation rules defined in GROOVE to Java. It is compatible with the GROOVE typegraph and has implemented almost every method of GROOVE’s graph structure. The implementation was complete enough to extend the river crossing model with a Java TUI and to extend the changing Java network with graph transformations.

However, the implementation does not scale well due to the way the matching of rules is implemented in GROOVE. This causes the whole edge set to be calculated far too often as the implementation is not cached. This also caused the performance difference between incoming edges and outgoing edges has been analyzed and it is clear incoming edges are a lot slower and do not scale well because they have to be calculated. We would recommend the creator of GROOVE to optimize their rule matching to better support non-cached graphs.

6.1 Future work

The current implementation could be extended with features. Features like edge multiplicity, inheritance, attributes and flags currently have not been implemented. These could be added to improve functionality and to add support for GROOVE models with advanced type graphs.

The implementation of the matching of GROOVE could also be optimized to better support our implementation. This way the performance of different types of graph transformations could be researched in greater much detail. This would allow one to research how the order of matching and types of transformations would affect the performance.
7. ACKNOWLEDGEMENT
We would like to thank Arend Rensink for supervising the project and giving useful feedback throughout the process.

8. REFERENCES

7