Interactive Visual Analytics for Efficient Maintenance of Model Transformations

Andreas Rentschler, Qais Noorshams, Lucia Happe, and Ralf Reussner

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{rentschler,noorshams,kapova,reussner}@kit.edu

Abstract. Maintaining model transformations remains a demanding task due to the sheer amount of metamodel elements and transformation rules that need to be understood. Several established techniques for software maintenance have been ported to model transformation development. Most available techniques proactively help to design and implement maintainable transformations, yet however, a growing number of legacy transformations needs to be maintained. Interactive visualization techniques to support model transformation maintenance still do not exist. We propose an interactive visual analytics process for understanding model transformations for maintenance. Data and control dependencies are statically analyzed and displayed in an interactive graph-based view with cross-view navigation and task-oriented filter criteria. We present results of an empirical study, where we asked programmers to carry out typical maintenance tasks on a real-world transformation in QVT-O. Subjects using our view located relevant code spots significantly more efficiently.

1 Introduction

Model transformations are of key importance in model-driven software development (MDSD). With the growing application of model-driven techniques in industry, maintenance costs of transformations move into a stronger focus.

During the life-cycle of a model-driven software system, transformations have to be adapted to evolving models, requirements, and technologies. Many established techniques to achieve maintainable software exist, including software quality metrics, program analysis, software testing, and modular programming. In recent years, substantial effort has been expended in porting such techniques to the specific field of model transformation development. The *program comprehension* community studies the way humans understand source code and how tools and techniques can support maintenance [1]. *Program analysis techniques* assist software maintainers in understanding unknown code and detecting code anomalies. When understanding a program, most parts are irrelevant as they do not contribute to a particular functional or non-functional concern. Slices can be automatically computed by statically analyzing the data and control flow of a program (static program slicing), or for a certain input to a program (dynamic program slicing). *Software visualization tools* [2] combine program analysis with information visualization techniques to create various displays of software structure, behavior, and evolution. In the recent past, approaches for detecting code anomalies and visualizing data and control flow had been transferred to the world of model transformation development, as well.

K. Duddy and G. Kappel (Eds.): ICMT 2013, LNCS 7909, pp. 141–157, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Comprehension of programs with computations on complex data structures has rarely been studied so far. Existing static analysis approaches to support model transformation maintenance do not integrate data and control flow [3], and proposed graphical representations of dependencies [4] are too complex to effectively support understanding for a particular task. None of the approaches has been empirically validated. What is needed is a *visual analytics approach* [5] that allows for task-oriented data reduction and interaction between data and users in order to discover knowledge.

In this paper, we propose an interactive visual analytics process to understand model transformations for maintenance. One part of the process is to statically analyze model transformations for their data and control dependencies. Task-oriented filtering aligns the level of detail to a variety of comprehension and maintenance tasks. Results are displayed in an interactive graphical view with cross-view navigation and dynamic selection of filter criteria. To the best of our knowledge, this is the first validated approach to interactively support maintenance of model transformations.

In a qualitative and quantitative study, we asked programmers to carry out maintenance tasks on a real-world transformation in QVT-Operational (QVT-O). Programmers using our editor extension located relevant code spots significantly more efficiently than programmers without the extension.

To sum it up, this paper makes the following contributions.

– We define an interactive visual analytics process to support understanding of model transformations for maintenance.
– We define a generic dependency model for model transformations, unifying control and data flow information in a graph-like structure.
– We propose a set of task-oriented filter rules to exclude details from the dependency graph that can be considered irrelevant to carry out a certain maintenance activity.
– We present results from an empirical experiment to see how task-oriented dependency graphs improve the efficiency for locating features in a transformation.

This paper is structured as follows. First, we present a motivating example in Section 2. In Section 3, we introduce our visualization process, in Section 4 a model for a generic dependency graph, and in Section 5 task-oriented filtering. Section 6 illustrates how the approach had been empirically validated. In Section 7 related work is discussed. Finally, Section 8 presents conclusions and proposes directions for future work.

2 Motivating Example

Consider a QVT-O implementation of the UML2RDBMS example scenario. One rule, `Attribute2Column`, is mapping elements of type `Attribute` to elements of type `Column`:

```
mapping Attribute::Attribute2Column() : Column
merges Attribute::IdAttribute2AutogenColumn {
    result.name := self.name;
    result.type := UmlTypeToDbtype(self.type.name);
}
```

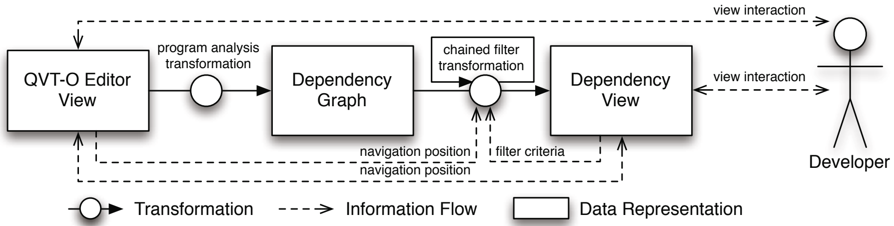


Fig. 1. Visual analytics process

By means of two practical scenarios, a comprehension scenario and a maintenance scenario, we demonstrate that even in an editor as advanced as that of Eclipse QVT-O, there is little support to cope with a transformation's inherent complexity.

Comprehension scenario. In this first scenario, we want to trace a bug resulting in missing `Column` objects in the target model. As a first step, we want to understand the context in which a rule `Attribute2Column` is used. In the QVT-O editor under Eclipse, we need to start a text-based search by the rule's name. If the transformation is modularized, we need to repeat the search in any other module with an `import` of the rule's containing module. To detect occurences, it is important to know that in QVT-O, keywords `disjuncts`, `merges`, `inherits` and `map` all have call-semantics.

Maintenance scenario. In the second scenario, we want to add a third attribute to class `Column`. Before we do so, it is useful to know about further locations in the program where `Column` or possible subclasses are instantiated. Again, we need to carry out a text-based search by the class name on all existing modules. In QVT-O, there are three ways to instantiate objects: implicitly via a `mapping`, or explicitly via the `object` operator, or by calling the corresponding `constructor` via `new` operator.

From these two scenarios it is clear that developers, especially those who are less familiar with QVT-O concepts, need to invest a lot of effort to identify relevant control and data dependencies in complex transformation programs. Program analysis techniques can automatically extract dependencies and present these graphically. However, large dependency graphs are difficult to read. In the next section of this paper, we propose a visual analytics process which takes the burden off developers to visualize only those dependencies which are considered as relevant for a particular activity.

3 Methodology Overview

To solve the information overload problem, *visual analytics processes* combine analytical approaches together with advanced visualization techniques. They can be characterized by four properties [6]:

> "Analyse First – Show the Important – Zoom, Filter and Analyse Further – Details on Demand"

We define an interactive visualization process which adheres to these principles. In this paper, we apply our approach to the QVT-O language. However, our concepts are

general enough to be transferred to further declarative and imperative languages. The process contains two transformation steps, a dependency analysis and a filter transformation (Figure 1). Code and graph representation are kept in sync regarding navigation position and code modifications. A user can interact with both views in parallel, and he can select and configure filters to fit his current task. In the following we show that our process meets all four properties of a visual analytics process.

Analyze first. Eclipse QVT-O features a textual representation, the *QVT editor view*. QVT-O automatically maintains a second representation, a model of the transformation and referenced metamodels. This model adheres to the QVT specification [7], thus offering a standardized interface for code analysis. The leftmost transformation in our process extracts data and control dependencies from the QVT model. Static control and data flow analysis results in an instance of a *dependency graph model*. This model is presented in Section 4.

Show the important. We already showed in the motivating example that, depending on what maintenance task is performed, a different subset of elements and element types available in the graph is required for reasoning. Which elements will be filtered out is configurable by the user. In Section 5, we explain our concept of *task-oriented filters* and we further define a set of useful types of filters.

Zoom, filter and analyse further. Humans are capable to only perceive a small subset of information at a time. Thus, we provide filters which remove any information out of focus. Depending on the task to be solved, a user's focus may lie on a certain method in the program, on a particular type of dependencies, or on a higher abstraction level. Because each view is limited regarding the information conveyed, *interaction* is needed. A feedback loop is established by dynamic filters, which can be quickly exchanged and configured, and which react on changes to the editing location. In Figure 1, filter dynamics are reflected by information flows leading to the filter transformation. Views are navigable, as pointed out by interactions between the developer and the views.

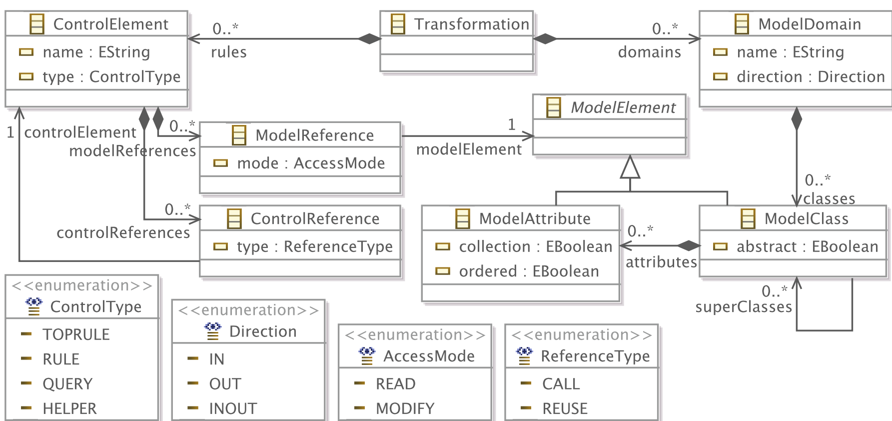
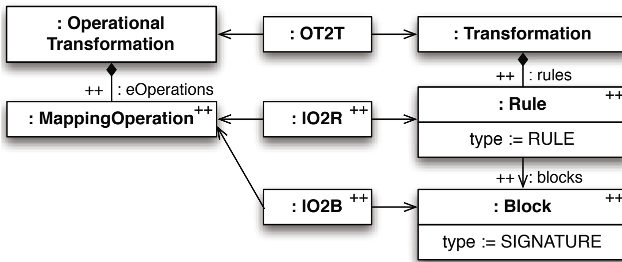


Fig. 2. Dependency graph model

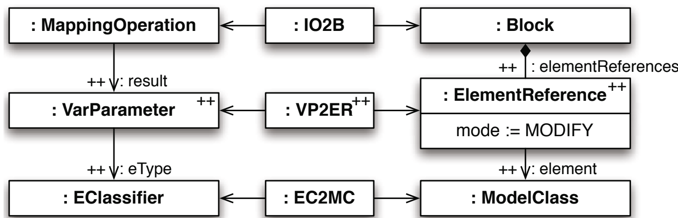
Details on demand. Occasionally, if maintainers require full details, the dependency view enables users to navigate to the underlying program code of a mapping or the actual definition of a data element. This kind of feature is called *cross-view navigation* and is illustrated in Figure 1 by an information flow pointing back to the editor view.

4 Dependency Graph Model

An important step in the visualization process is to analyze rule-based model transformations for control units, data units, and interdependencies. Control-units are transformation rules and other top-level constructs to specify behavior, for example query functions and helper methods in QVT-O. Data units are metamodeling concepts, classes, attributes, and references. Figure 2 shows how we defined these concepts in the Ecore metamodeling language. In the model, a transformation (`Transformation`) consists of control elements (`ControlElement`) and metamodel domains (`ModelDomain`). Each domain refers to a metamodel name and has a direction (`IN`, `OUT`, or `INOUT`). Control elements have references (`ControlReference`) to other rules for reuse purposes (`inherits` or `merges` in QVT-O), tagged `REUSE`, but also for calls (`map` or `disjuncts` in QVT-O), tagged `CALL`. Control elements are typed. In QVT-O a `mapping` equals a `RULE`, method `main` equals `TOPRULE`, query and helper functions correspond to `QUERY` and `HELPER`. A control element can have data dependencies, represented by instances of `ModelReference`. Referencing a model element (`ModelElement`) can be either read-only (`READ`), or the referenced model element



(a) Constructing an instance of `Rule` from a QVT-O mapping operation



(b) Constructing an `ElementReference` instance from a QVT-O mapping's result parameter

Fig. 3. Constructing dependency graphs from QVT-O transformations

can be instantiated or its content modified (`MODIFY` for both cases). The model differentiates between access to a class and access to an attribute or reference (`ModelClass` and `ModelAttribute`). Like in Java, there is no distinction between references and attributes. Classes can further reference super classes. Any model element belongs to exactly one domain.

A graph model instance is constructed from instances of QVT-O's abstract syntax metamodel [7]. Figure 3 demonstrates by two exemplary triple graph grammar (TGG) rules in compact notation [8], how QVT-O mappings correspond to rules in the graph, and result parameters are represented by element references with write access mode.

For our implementation and in this paper, we decided to visualize dependency graphs as node-link diagrams (NLDs). An illustrative mapping between graph elements and notational elements is given in Figure 4.

5 Task-Oriented Filtering

Dependency graphs of model transformations can be huge, bearing the danger that useful information gets lost when studied by humans. Our idea is to identify and remove details which are irrelevant for performing a particular task before display. In this section, we present three filter functions and four useful combinations thereof.

As prerequisite for defining the filter functions, we formalize our dependency graph as follows: Let $ControlElement$, $ModelClass$, and $ModelAttribute$ be sets that represent instances of metamodel classes of the same name, respectively. Furthermore, let $ModelElement := ModelClass \dot{\cup} ModelAttribute$ denote the set of `ModelElement` instances, i.e., the set containing instances of `ModelClass` and `ModelAttribute`. For referenced instances, let functions $modelReferences$, $modelElement$, $controlReferences$, $controlElement$, $superClasses$, $attributes$ be defined by representing sets of instances of their respectively named metamodel reference. When applied to sets, functions are applied element-wise and the result is the union of each mapping.

Then, we create the dependency graph G as a tuple of vertices V and Edges E, i.e., $G = (V, E)$. V and E elements are created using $ControlElement$, $ModelClass$, and $ModelAttribute$ elements by defining sets $V^{ControlElement}$, $V^{ModelClass}$, and $V^{ModelAttribute}$, as well as bijective functions ϕ_X (i.e., mapping rules), where

$$\begin{aligned}\phi_{ControlElement} &: ControlElement \rightarrow V^{ControlElement}\\ \phi_{ModelClass} &: ModelClass \rightarrow V^{ModelClass}\\ \phi_{ModelAttribute} &: ModelAttribute \rightarrow V^{ModelAttribute}\end{aligned}$$

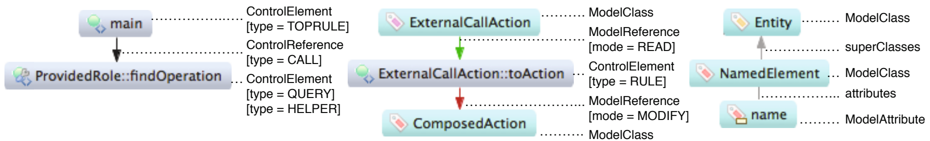
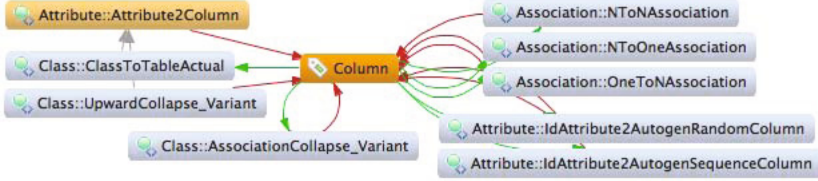


Fig. 4. Notational elements

(a) Control dependencies for QVT-O mapping `Attribute2Column`



(b) Data and control dependencies for class `Column`

Fig. 5. Filtered dependency graphs for the introductory example

We imply $V^{ControlElement}$, $V^{ModelClass}$, $V^{ModelAttribute}$ are pairwise disjunct by construction and define $V := V^{ControlElement} \dot\cup V^{ModelClass} \dot\cup V^{ModelAttribute}$. We further define $V^{ModelElement} := V^{ModelClass} \dot\cup V^{ModelAttribute}$. To obtain E elements, where $E \subseteq V \times V$, we use the following four derivation rules:[1]

$$\forall r_1, r_2 \in ControlElement :$$
$$r_2 \in (controlElement \circ controlReferences)(r_1) \iff (\phi_{ControlElement}(r_1), \phi_{ControlElement}(r_2)) \in E$$
$$\forall r \in ControlElement, e \in ModelElement :$$
$$e \in (modelElement \circ elementReferences)(r) \iff (\phi_{ControlElement}(r), \phi_{ModelElement}(e)) \in E$$
$$\forall c_1, c_2 \in ModelClass :$$
$$c_2 \in superClasses(c_1) \iff (\phi_{ModelClass}(c_1), \phi_{ModelClass}(c_2)) \in E$$
$$\forall a \in ModelAttribute, c \in ModelClass :$$
$$a \in modelAttributes(c) \iff (\phi_{ModelAttribute}(a), \phi_{ModelClass}(c)) \in E$$

Now we can define three filter functions on top of the computed graph structure G.

Filtering control nodes. We define a filtering function removing data dependencies by $f^{controlflow}(V, E) := (V', E')$, where

$$V' := V \setminus V^{ModelElement}$$
$$E' := E \setminus (V^{ControlElement} \times V^{ModelElement}) \setminus (V^{ModelClass} \times V^{ModelClass}) \setminus (V^{ModelClass} \times V^{ModelAttribute})$$

[1] The function composition operator "$\circ$" is defined as: $(g \circ f)(x) = g(f(x))$.

Filtering direct dependencies. Let $v_{current} \in V$ be the node whose direct dependencies shall be filtered. Contextual filtering is defined by function

$$f^{context}_{v_{current}}(V, E) := (V', E'), \text{ where}$$
$$V' := \{v \in V \mid (v, v_{current}) \in E \vee (v_{current}, v) \in E\} \cup \{v_{current}\}$$
$$E' := \{(v_1, v_2) \in E \mid v_1 = v_{current} \vee v_2 = v_{current}\}$$

Filtering classes without attributes. To reduce complexity, attributes of classes and dependencies can be filtered by

$$f^{classes}(V, E) := (V', E'), \text{ where}$$
$$V' := V \setminus V^{ModelAttribute}$$
$$E' := E \setminus (V^{ControlElement} \times V^{ModelAttribute}) \setminus (V^{ModelAttribute} \times V^{ModelClass})$$

These filter functions can be arbitrarily combined. However, the order in which functions are applied is relevant, and not all combinations can be conceived as useful. We define four filter combinations together with their primary field of application:

$$F1 := f^{controlflow}, \qquad F2 := (f^{context}_{v_{current}} \circ f^{controlflow}),$$
$$F3 := f^{context}_{v_{current}}, \qquad F4 := (f^{context}_{v_{current}} \circ f^{classes}).$$

F1: Show control dependencies of the whole transformation. Resulting view helps to initially grasp the overall control structures of an unknown transformation. The filter is useful for investigating transformations of smaller size.

F2: Show control dependencies in context of the currently selected control node. When navigating the rules, data dependencies are not always of interest. For the comprehension scenario from Section 2, this filter is the optimal choice. In context of mapping `Attribute2Column` the filter would yield the graph shown in Figure 5a. A developer quickly recognizes two rules calling mapping `Attribute2Column`. The filter helps at understanding larger transformation programs.

F3: Show control and data dependencies in context of the currently selected control or data node, $v_{current}$. When reading transformation rules, it makes sense to primarily concentrate on direct dependencies. In contrast to slicing criteria, only direct dependencies are displayed, both in forward and backward directions. In the maintenance scenario from Section 2, we apply the filter in context of class `Column` to show all operations modifying or reading the element. There are seven mappings modifying the element (incoming connections in Figure 5b. The filter helps to cope with change requests where details at the attribute-level are required to locate concerns.

F4: Show control and data dependencies in context of the currently selected control or data node, but remove information about accessed class attributes. The resulting view leaves only data dependencies at the class-level. This filter is useful for change requests where details at the class-level are sufficient to locate concerns.

6 Empirical Evaluation

To evaluate how our approach supports important maintenance tasks, we carried out a case study, where users had to identify code locations affected by typical change requests. For the case study, we implemented our approach for QVT-O under Eclipse.[2]

6.1 Design

The purpose of the study is to empirically show that our approach makes maintaining model transformations more efficient (process-related improvement), the outcome is of higher quality (product-related improvement), and the developer experiences less effort (improvement regarding user experience).

In order to accomplish our set goal, we explored the following three hypotheses:

H1: Effectiveness. Subjects who use dependency graphs locate affected places *more effectively* than equally classified subjects who do not use it.

H2: Time expenditure. Subjects who use dependency graphs are *faster* in performing the maintenance tasks than equally classified subjects who do not use it.

H3: Perceived strain. Subjects who use dependency graphs are *less strained* than equally classified subjects who do not use it.

We varied the availability of the tool (control variable) and took measurements to assess effectiveness, time consumption and perceived strain (response variables). We used the following metrics: To evaluate the effectiveness, we determined the numbers of false positives and false negatives of each given answer. Based on these, we used the f-measure with $\beta = 1$ to compute the harmonic mean value of precision and recall. The f-measure is a widely-used measure for assessing quality of information retrieval. It is also suitable for evaluating feature location tasks [9].

Subjects had been asked to record starting and ending time for each task, so we could calculate the actual time needed. Regarding the subjectively felt level of strain, we asked for the user's personally experienced difficulty level on six-level Likert items in the post-session questionnaire.

According to a classification scheme by Juzgado and Moreno [10], this experiment follows a *between-subjects design*. Its single dimension *tool usage* has two levels (with or without). It is a *quasi-experiment*, since assignment to groups had been done based on information determined from the pre-questionaire, rather than purely randomly.

The study had been carried out in an exam-style situation on 2 bachelor students, 12 master students and 8 experienced researchers, all of them reasonably trained in the tools and activities of model transformation development.

We decided for an example transformation from a large scientific project, the Palladio Research Project [11]. The project provides a set of methods and tools for predicting the reliability and performance of software architectures. The QVT-O transformation PCM2QPN transforms Palladio Component Model (PCM) instances to Queuing Petri Nets (QPN). It encompasses 2886 lines of code, plus 952 lines of code distributed over 4 library modules. We decided for QVT-Operational because of its stable integration

[2] The tool we implemented can be downloaded from `http://sdqweb.ipd.kit.edu/wiki/Transformation_Analysis`.

into the Eclipse IDE, its adherence to a language standard, and its wide acceptance. There is one source metamodel, the Palladio Component Model (PCM), consisting of 154 classes, and SimQPN's Petri net model as target model, consisting of 20 classes.

Each subject was asked to understand certain aspects, and to locate concerns of a set of change requests to a correct set of code places. In software engineering, there are typically four types of maintenance tasks [12], preventive, corrective, perfective and adaptive. In accordance to this widely used classification, we name five classes of change requests:

Bug fix request (Corrective): This request is about finding a bug which is present in the transformation, either the program does not compile, or the output is wrong. For instance, as a reaction to modifications to the metamodels, developers need to adapt the transformation accordingly.

Feature request (Perfective): To match new functional requirements, new functionality needs to be added, or existing functionality needs to be changed or removed. Because changes may originate from or have impact on depending artifacts, feature requests may result not only in modification of a transformation but also of metamodels, models and documentation.

Non-functional improvement (Perfective): A perfective task can also target non-functional requirements, it can even have an impact that is orthogonal to code structure.

Refactoring request (Preventive): Refactoring means organizing code structure without adding or removing existing functionality. For instance, a class attribute could be pulled up, which can result in an attribute assignment being pulled up a rule inheritance chain accordingly.

Environmental change request (Adaptive): Transformation programs do not function in isolation. Input metamodels change, or language concepts are updated.

Subjects were asked to *not* perform the actual change, because of the limited time frame, and because people brought a varying level of knowledge and experience with QVT-O and the underlying Object Constraint Language (OCL). Subject had to handle the following seven tasks:[3]

T1: Comprehension task / Searching for keywords

"Where does the transformation create elements in the target model? Name one example for each of the three variants for element instantiation."

Subjects had to look for the keywords `constructor`, `object`, `mapping`. They were asked to name one example for each instantiation type, 3 locations in total. Note that for mappings, non-tool users had to check for a return type, with implicit instantiation semantics. Tool users could check write-access edges in the graph, but still had to check the underlying code for the actual used instantiation type. Filter 4 combined with cross-navigation was the optimal choice. This task served as a warm-up question.

T2: Comprehension task / Analyzing control flow

"Which call trace leads from the entry method to a method creating instances of

[3] The experiment's tasks can be understood without knowledge of the PCM and QPN metamodels.

`ExternalCallAction`*? Do not use the debugger or execute instrumented code."*
A manual depth-first search using the browsing history had to be conducted, in order to find paths in the overall control flow leading from the entry method's node to the target method. Non-tool users had to use text-based search and hyperlink navigation. Tool users could use Filter F2, after identifying the target method as a method having write-access to the named class (Filter F4). The trace was 8 methods deep, and included downcasts regarding the contextual type of a mapping.

T3: Refactoring request / Analyzing control flow.

"Name all unused methods."

Non-tool users were required to search for occurences of each method's name. The transformation's main file had 41 mappings, 117 helpers, and 10 queries. Tool users could check for unreferenced nodes in the general control-flow view (Filter F1). There were 12 unused methods in total, 7 mappings and 5 helpers.

T4: Non-functional improvement / Analyzing data element usage

"Assert statements need to be added to check consistency of created connections. Please name all methods that instantiate objects of type `ConnectionType`*."*

Subjects had to look for the keywords `constructor`, `object`, `mapping`, in conjunction with `ConnectionType`. Non-tool users had to use the search command. Tool users could rely on write-access dependencies (Filter F4). There were 33 instantiating methods in total, 30 mapping rules and 3 helper methods.

T5: Feature request / Analyzing data element usage

"A new subtype of `AbstractAction` *is planned to be added to the source metamodel. Where are* `AbstractAction` *elements handled? Name all occurences."*

Here, subjects had to check data dependencies. While non-tool users had to use the search command, tool users could select Filter F4 and check outgoing or incoming edges in context of class `AbstractAction`. The correct answer included 5 mappings and 5 queries, where 2 queries were located in an external module.

T6: Feature request / Analyzing data element usage

"Class `ForkAction`*, subtype of* `AbstractAction`*, should be used as blue print for the new subtype of* `AbstractAction` *to be added. Where does the transformation handle the* `ForkAction` *element? Name all occurences."*

The right answer encompasses 8 queries and 6 mappings. Tool users could check data dependencies in context of `ForkAction` (Filter F4), others had to do a text-based search for the name.

T7: Bug fix request / Analyzing data flow for unused class attributes

"Created target models contain objects of class `Place` *with an uninitialized attribute* `departureDiscipline`*. Identify the buggy lines of code."*

Only one single mapping did create elements of type `Place` without proper initialization. Tool users were able to use Filter F3 in context of class `Place`, and check attribute dependencies for each displayed method.

6.2 Execution

Students had participated in two practical training sessions on transformation development. Training was done within scope of a practical course on MDSD. Each session ended with graded exercise sheets. We sent our fellow researchers training material to brush up their knowledge of the QVT-O language.

Assignment to one of the two groups happened randomly. Beforehand, participants had to fill out a pre-session questionnaire, where they rated their own expertise level on a 5-point Likert item and stated their academic degree. Based on this information, we randomly swapped participants between both groups so that each group had 7 students and 4 researchers, and the mean expertise level for both groups was equally balanced.

The experiment started with a 30 minutes tutorial on how to use the tool. Each participant was assigned to one workstation with a preconfigured Eclipse IDE. Then, subjects were handed out the task sheets, they were asked to answer tasks in prescribed order, and to note down when they started and when they ended a task. Subjects could freely partition their available time to the tasks. Subjects could decide to end a task prematurely, without the option to resume later. After 75 minutes total, the experiment closed with a post-session questionnaire. Using a debugger or executing the code was not permitted.

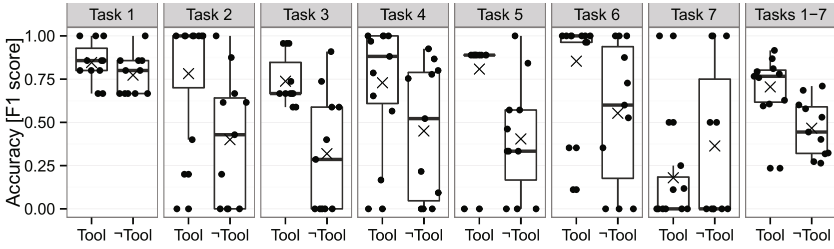
6.3 Analysis

For analysis, none of the outliers had been removed from the data set.

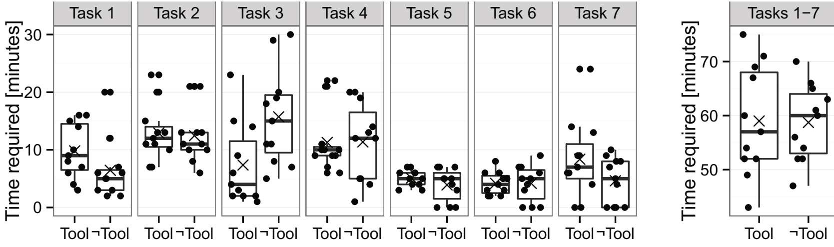
H1: Effectiveness. For hypothesis H1 we investigated the f-measure. Boxplots in Figure 6a contrast program with control group on a per-task level. Applying Welch's one-tailed t-test to the f1-measures at the default significance level of $\alpha = 0.05$, tool users showed a significant improvement over non-tool users for tasks 2-6 ($p_2 = 0.015$, $p_3 = 0.001$, $p_4 = 0.047$, $p_5 = 0.003$, $p_6 = 0.034$). For tasks 1 and 6, Welch's two-tailed t-test did not reveal a significant difference ($p_1 = 0.160$, $p_7 = 0.283$). We are able to reject H1's corresponding null hypothesis for all tasks but task one and task seven.

H2: Time expenditure. For H2 we tested time consumption. Figure 6b shows boxplots for the consumed time per task and added up. Welch's t-test had been used to test for significancy. We can confirm hypothesis H2 only for task 3 with a one-tailed test revealing $p = 0.010$. For the other tasks, two-tailed tests did not indicate any significant difference between groups.

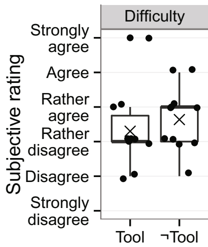
H3: Perceived strain. H3 was based on subjective data from the questionnaires. All answers were posed using 6-point Likert items, ranging from "strongly disagree" to "strongly agree". One question was if subjects would rate the tasks as difficult to solve (see Figure 6c for details). Further questions asked tool users if they think that the tool helps in understanding, debugging, refactoring, and extending a previously unknown transformation, based on their experience they gained at the respective task. Figure 6d shows respective boxplots. Mean values report that non-tool users found the tasks to be "rather hard", while tool users found the tasks to be "rather easy". Additionally, tool users rated the tool's usability for understanding, debugging, and refactoring a



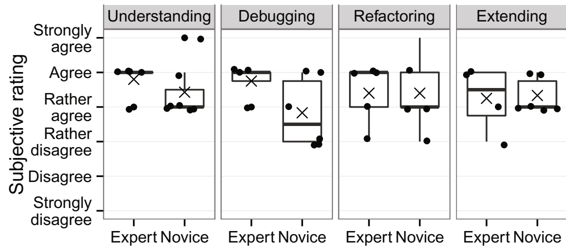
(a) Effectiveness calculated from precision and recall



(b) Timing



(c) Rating of the overall task difficulty



(d) Rating of the tool's ability to assist at one of the tasks, grouped by experts and novices

Fig. 6. Measured response variables[4]

transformation on a 6-level Likert item. On average, participants agreed that the tool helps in all four disciplines.

6.4 Discussion

H1: Effectiveness. Effectiveness is significantly improved for all tasks except the first and the last. For the first task, thought to be a warm-up exercise, we argue that even if tool users could profit from navigating the graph, they still had to check the underlying code for the keywords, making tool-based navigation only slightly better than a common text-based search. Many users did not find enough time for the last task, 4 out of 11 non-tool users and 3 out of 11 tool users did not even begin to process this task.

[4] In the boxplots, individual values are marked as dots jittered horizontally (and for discrete values also vertically) by a random value. A cross marks the mean, and a bar the median value.

A known problem lies in the tool's inability to detect attributes accessed from within a constructor.

H2: Time expenditure. The overall expenditure of time was almost indifferent. Task 3 was solved significantly faster, and with better results. This indicates that for some maintenance tasks our approach produces much better results than for others.

H3: Perceived strain. According to the ratings, tool users perceived the same tasks less difficult than non-tool users. Based on their experiences, most tool users found the tool to support understanding, debugging, refactoring and extending transformations, although to a limited extend. Novices were less convinced of the tool's help for debugging tasks. We expect developers to prefer the Eclipse debugging perspective over static analysis for most types of bugs.

Because of a significant improvement of the overall effectiveness, the indifferent overall expenditure of time, and a less perceived strain, we are able to attribute a higher efficiency to tool users. Results show that for some tasks, the abstraction level offered by our tool is too high.

6.5 Threats to Validity

Construct validity. The study's primary construct is the use of data and control dependencies to locate concerns. The choice of change requests was carefully chosen to represent a real-life situation. We are aware that there are change requests which are not in alignment to data flow and code structure, e.g. cross-cutting concerns, others require finer-grained knowledge, e.g. details of the program code. For instance, the tool's program analysis did miss two dependencies, resulting in a slightly smaller recall value for task 5 (cf. Figure 6a). A second threat is due to the metrics we used. In feature location scenarios, using the f-measure is considered as a common method to compare product-related quality [9]. Since people could freely partition their available time to the tasks, recorded times are not accurate, particularly towards the end. Subjective ratings need to be treated with care.

Internal validity. Blind testing was not possible, but people were assigned to one group at the latest possible time. Subjects were equally trained, advice had been given for each task on how to optimally use the tool and an alternative IDE feature. None of the subjects had been involved in the tool's development. We refrained from asking subjects to perform the actual maintenance task, because we expect the tool to show its particular strength in understanding code and locating concerns rather than in editing code.

External validity. Generalizability is threatened by the fact that we investigated only a single transformation written in a single transformation language. We are confident the transformation together with the two incorporated Ecore metamodels, the Palladio component model (PCM) and the queueing Petri-net model (QPN), reflect industrial quality standards. Program and model artifacts had been reviewed at least once. We also believe that our mix of novices and experts approximated to a real-world situation. We compared our tool to the bare Eclipse QVT-O environment, as we do not know of similar tools for QVT-O. Yet, by further equipping non-tool users with diagrammatic

visualizations as those suggested by van Amstel et al. [4], we could check if our interactive approach would outcompete a static visualization approach.

7 Related Work

Program Analysis. Program analysis techniques are already applied to model transformations. Varro and his colleagues transfer graph transformations into Petri nets [13], where they are able to prove termination for many programs. Ujhelyi, Horvath and Varro analyze VIATRA2 VTCL programs for common errors in transformation programs [14]. The same authors suggest a dynamic backward slicing approach [15] to understand program behavior for a certain input. In comparison, our approach is based on static program analysis, aiming to support the process of understanding for maintenance rather than reasoning about program properties. Schönböck et al. use Petri nets to integrate data and control structures into a graphical view [16] to foster debuggability. Their approach is designed for declarative rule-based transformation languages and lacks validation.

Vieira and Ramalho developed a higher order transformation [3] to automatically extract dependencies from ATL transformations. Their objective is similar to ours, namely to assist developers inspecting transformation code. The unvalidated approach is ATL-specific, it lacks data dependencies and filters. A user interface is still missing.

Eclipse editors, including that for QVT-O, support hyperlinked syntax. However, control dependencies are not computed live, and navigation over calls is only possible in the forward direction. Learning about data dependencies from code requires good cognitive abilities and a thorough knowledge of all the relevant language concepts. Still, data dependencies derived from other methods can not directly be seen. Furthermore, it is not possible to directly learn about all the places a particular data element is accessed.

Program Visualization. Software visualization tools had been surveyed by Diehl in his book from 2007 [2]. Telea et al. make a comparison between hierarchical edge bundling (HEB) visualizations and classical node-link diagrams (NLDs) when used for comprehending C/C++ code [17]. Our graph notation can be classified as an NLD. Recently, van Amstel et al. have been conquering HEB diagrams for visualizing a transformation's data and control dependencies and metamodel coverage [4]. However, data and control dependencies are not integrated into a single view, and effectiveness and efficiency of static HEB diagrams for maintenance tasks remain to be validated.

8 Conclusions and Outlook

We demonstrated a novel approach to visualize data and control flow dependencies of metamodels and transformations using interactive node-link diagrams (NLDs). Efficiency of our approach has been shown in an experiment, where subjects using our approach were significantly more efficient and effective carrying out maintenance tasks. Results suggest that it is the large number of dependencies among metamodel elements and transformation rules that hampers understandability of model transformations.

Our study indicates that maintenance processes can be heavily improved by revealing dependency information to maintainers. Instead of utilizing program analysis techniques, transformation languages could proactively provide concepts to let programmers explicitly declare dependencies for program elements, e.g. rules and modules. Since prevalent module concepts are coined towards reuse rather than maintenance [18], we consider a module concept where dependencies can be declared upfront. For existing transformations, a module's dependencies can be derived automatically with our approach. A *clustering algorithm* based on dependency metrics could even propose suitable modular structures [19]. Next, we plan to try different types of visualizations, for example *filtered HEBs*. In future experiment runs, we would like to test transformations in further dialects, and compare our approach with other existing approaches. Additionally, analysis of OCL expressions [20] needs to be refined, as not all data dependencies are captured yet.

Acknowledgements. This research has been funded by the German Research Foundation (DFG) under grant No. RE 1674/5-1. We thank our experimentees for their valuable time.

References

1. Storey, M.A.D.: Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future. Software Quality Journal 14(3), 187–208 (2006)
2. Diehl, S.: Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Springer (2007)
3. Vieira, A., Ramalho, F.: A Static Analyzer for Model Transformations. In: MtATL 2011. CEUR Workshop Proceedings, vol. 742, pp. 75–88. CEUR-WS.org (2011)
4. van Amstel, M.F., van den Brand, M.G.J.: Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 108–122. Springer, Heidelberg (2011)
5. Keim, D.A., Kohlhammer, J., Ellis, G., Mansmann, F.: Mastering the Information Age - Solving Problems with Visual Analytics. Eurographics Association (2010)
6. Keim, D.A., Mansmann, F., Schneidewind, J., Thomas, J., Ziegler, H.: Visual analytics: Scope and challenges. In: Simoff, S.J., Böhlen, M.H., Mazeika, A. (eds.) Visual Data Mining. LNCS, vol. 4404, pp. 76–90. Springer, Heidelberg (2008)
7. Object Management Group (OMG): MOF 2.0 Query/View/Transformation, version 1.1 (January 2011), `http://www.omg.org/spec/QVT/1.1/PDF/`
8. Kindler, E., Wagner, R.: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report TR-RI-07-284, Univ. of Paderborn (2007)
9. Wang, J., Peng, X., Xing, Z., Zhao, W.: An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions.. In: ICSM 2011. IEEE (2011)
10. Juzgado, N.J., Moreno, A.M.: Basics of Software Engineering Experimentation. Kluwer Academic Publishers (2001)
11. Meier, P., Kounev, S., Koziolek, H.: Automated Transformation of Component-Based Software Architecture Models to Queueing Petri Nets. In: MASCOTS 2011, pp. 339–348. IEEE (2011)
12. Saleh, K.A.: Software Engineering. J Ross Publishing (2009)

13. Varró, D., Varró–Gyapay, S., Ehrig, H., Prange, U., Taentzer, G.: Termination Analysis of Model Transformations by Petri Nets. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 260–274. Springer, Heidelberg (2006)
14. Ujhelyi, Z., Horváth, Á., Varró, D.: A Generic Static Analysis Framework for Model Transformation Programs. Technical report, Budapest Univ. of Technology and Economics (2009)
15. Ujhelyi, Z., Horváth, Á., Varró, D.: Dynamic Backward Slicing of Model Transformations. In: ICST 2012, pp. 1–10. IEEE (2012)
16. Schönböck, J., Kappel, G., Kusel, A., Retschitzegger, W., Schwinger, W., Wimmer, M.: Catch Me If You Can - Debugging Support for Model Transformations. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 5–20. Springer, Heidelberg (2009)
17. Telea, A., Hoogendorp, H., Ersoy, O., Reniers, D.: Extraction and Visualization of Call Dependencies for Large C/C++ Code Bases: A Comparative Study. In: VISSOFT 2009, pp. 81–88. IEEE (2009)
18. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Fact or Fiction – Reuse in Rule-Based Model-to-Model Transformation Languages. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 280–295. Springer, Heidelberg (2012)
19. Dietrich, J., Yakovlev, V., McCartin, C., Jenson, G., Duchrow, M.: Cluster Analysis of Java Dependency Graphs. In: SoftVis 2008, pp. 91–94. ACM (2008)
20. Jeanneret, C., Glinz, M., Baudry, B.: Estimating Footprints of Model Operations. In: ICSE 2011, pp. 601–610. ACM (2011)