A Survey and Comparison of Transformation Tools Based on the Transformation Tool Contest

Edgar Jakumeit*, Sebastian Buchwald[a], Dennis Wagelaar[b], Li Dan[c], Ábel Hegedüs[d], Markus Herrmannsdörfer[e], Tassilo Horn[f], Elina Kalnina[g], Christian Krause[h], Kevin Lano[i], Markus Lepper, Arend Rensink[j], Louis Rose[k], Sebastian Wätzoldt[h], Steffen Mazanek

[a]*Karlsruher Institut für Technologie, Germany*
[b]*Vrije Universiteit Brussel, Brussels, Belgium*
[c]*Faculty of Science and Technology, University of Macau, China*
[d]*Budapest University of Technology and Economics, Hungary*
[e]*Institut für Informatik, Technische Universität München, Germany*
[f]*Institute for Software Technology, University Koblenz-Landau, Germany*
[g]*Institute of Mathematics and Computer Science, University of Latvia, Latvia*
[h]*Hasso-Plattner-Institut, Universität Potsdam, Germany*
[i]*Dept. of Informatics, King's College London, Strand, London, UK*
[j]*Department of Computer Science, University of Twente, The Netherlands*
[k]*Department of Computer Science, University of York, UK*

Abstract

Model transformation is one of the key tasks in model-driven engineering and relies on the efficient matching and modification of graph-based data structures; its sibling graph rewriting has been used to successfully model problems in a variety of domains. Over the last years, a wide range of graph and model transformation tools have been developed – all of them with their own particular strengths and typical application domains. In this paper, we give a survey and a comparison of the model and graph transformation tools that participated at the Transformation Tool Contest 2011. The reader gains an overview of the field and its tools, based on the illustrative solutions submitted to a Hello World task, and a comparison alongside a detailed taxonomy. The article is of interest to researchers in the field of model and

*Corresponding author
Email addresses: eja@ipd.info.uni-karlsruhe.de (Edgar Jakumeit), post@markuslepper.eu (Markus Lepper), steffen.mazanek@gmail.com (Steffen Mazanek)

graph transformation, as well as to software engineers with a transformation task at hand who have to choose a tool fitting to their needs. All solutions referenced in this article provide a SHARE demo. It supported the peer-review process for the contest, and now allows the reader to test the tools online.

Keywords: tool survey, tool comparison, hello world, transformation tool contest, SHARE, graph rewriting, model transformation

---

1. Introduction

The work in this article is part of the special issue on Experimental Software Engineering in the Cloud of the journal Science of Computer Programming, which collates articles that compare software engineering tools in a pragmatic manner, for example by having tool developers solve a shared case study. The focus of this special issue is on reproducibility and accessibility, to be satisfied with virtual machine demos.

1.1. Background

The Transformation Tool Contest [99] held 2011 in Zurich, Switzerland, was a research workshop where providers of various transformation tools from the worlds of model transformation and graph rewriting gathered to compare their tools alongside a number of selected case studies. At this workshop, SHARE [25] virtual machines were employed to support the peer-review process.

The compared solutions were implemented with complex software toolkits, often requiring a substantial amount of time and knowledge to install and set up. The SHARE cloud allowed all participants to start up the pre-installed software environment of their competitors including their solutions with just one click, drastically lowering the barrier for comparison and evaluation.

While being meant originally for peer-based evaluation, the SHARE images of this scientific workshop were kept available for further empirical evaluation by prospective tool users from industry, who can explore the effect of inputs and parameters better suited to their needs.

While held as a scientific workshop for comparing transformation tools alongside common case studies, one of the case studies consisting of small and illustrative tasks resulted in instructive solutions that are simple enough to

be understood with limited knowledge and in a very limited amount of time. Based on those solutions, an introduction into the tools and their approaches can be given.

We take the chance offered by those facts and summarize the results and findings of the Transformation Tool Contest 2011 in the following not only for the tool building community, but also for a prospective user on the search for the right graph or model transformation tool to employ.

1.2. Motivation

We aim at assisting software engineers that are facing a specific model transformation or graph rewriting problem in choosing a tool that is well-suited to their needs. This statement gives rise to the questions: What is *a model transformation* or *a graph rewrite problem*? And: What does *well-suited to someone's needs* mean? In Appendix A we investigate these questions and their answers in detail, together with a discussion of the advantages and disadvantages of using those tools; here, we give a brief summary.

In a nutshell, we are confronted with a graph rewriting or model transformation problem if the most adequate representation of the data at hand is a mesh of objects, and we need to change its structure or map it to another mesh of objects.

The transformation tools introduced in this article offer *faster* development of solutions compared to manual coding in a traditional programming language, by *reusing* existing functionality. The transformation languages offered by the tools are typically of higher *expressiveness* than general-purpose programming languages for tasks of the transformation domain, leading to more *concise* solutions, thus *lower*ing maintenance *costs*. They are *declarative*, i.e., they allow you to concentrate on the *what* instead of on the *how*. Often, these tools offer a *visual* style of specification and debugging (or simulation).

1.3. Survey and Comparison

The problem of *reuse* is: Are the available features really the ones that are needed? Using a caliper when a hammer is needed would not offer the expected benefits. You must *choose* according to *your needs*.

In order to assist you in choosing a promising tool we present different kinds of information in the following sections, at an increasing level of detail. They allow to incrementally reduce the large set of potential candidates to a small set of the most promising candidates, which can then be evaluated

in depth, based on their article in the proceedings of the workshop [99], but especially by having a look at their SHARE images.

Please note that our focus is on assisting you in *choosing* a tool, under the assumption that model transformation or graph rewriting are not foreign words to you. For a tutorial *introduction* into the field please have a look at Graph Transformation in a Nutshell [28] and Model-Driven Software Engineering in Practice [9] or MDA Explained — The Model Driven Architecture: Practice and Promise [55].

Hello World. We start with a description of the Hello World case [70], which was posed at the Transformation Tool Contest 2011 [99]. The Hello World case is a mini-benchmark consisting of several prototypical tasks of the domain (and is thus a lot more revealing than the Hello World program from traditional programming [54], which only requires to print a greeting message to the console). Those tasks are *simple* yet highly *illustrative*. The tasks as well as the solutions can be read and understood *quickly*, even by non-experts. All tools are introduced with their solution of *one* of the tasks. The solutions of *all* tasks are available in the SHARE images of the tools.

Discussion. The section discusses the Hello World task used for introducing the tools, as well as the solutions. It explains the performance of the tools on that task, and corrects possibly wrong impressions.

The World of Transformation Tools. This section gives an introduction into the field. It unfolds a coarse grain map highlighting the locations of the tools just introduced. The major high-level discrimination points and their consequences are discussed.

Classification in Detail. The section allows for an in-depth comparison of the tools. It refines the taxonomy introduced in *The world of transformation tools* with examinations going into greater detail. As in Section 6, the tools are compared regarding their support for the specified aspects with feature matrices. They allow to quickly reduce the large set of tools according to the criteria of the task at hand to a small set of tools to be evaluated in depth.

The importance of the aspects listed there depends on what *you* need. We explain the circumstances under which the features are of relevance, and discuss their consequences. We must note that you could have to skip certain parts here that are overly detailed for your needs; they are targeted at the second group of readers, members of the tool building community.

1.4. Guide to Reading

This article sums the results of the Transformation Tool Contest 2011, and it does so for two groups of readers: on the one hand for prospective users, and on the other hand for the community of tool builders.

If you are a prospective user, you are interested in the sections just introduced in the previous Section 1.3.

If you are a member of the tool building community, you are likely interested in those sections written for an imaginary user, too. The field is very wide and heterogeneous, you presumably will get a better overview of it by reading through the tool introductions; besides, you will be likely interested in seeing the effect of the availability or unavailability of declarative means for processing structures splitting into breadth. The sections on *The World of Transformation Tools* and the *Classification in Detail* are founded on a taxonomy that is developed in two steps at an increasing level of detail. It is applied to the diverse tools that participated, yielding a survey of the field. The taxonomy offers a first, high-level break-up of the notion of expressiveness for transformation tools. Besides, you will be interested in the following sections.

Transformation Tool Contest and SHARE. The section introduces the workshop format of the Transformation Tool Contest and explains the role of SHARE [25], highlighting its support of the review process. Furthermore, the votes cast by the tool providers for the solutions at the Transformation Tool Contest (TTC) are listed (hinting at the usability of the tools for *many different tasks*), and finally discussed and put into perspective.

Related Work. The section introduces related work (in-depth tool comparisons, the other comparisons of the TTC, and related taxonomies) and compares them with this article.

1.5. Contribution

The contributions of this article are as follows:

- It introduces many of the state-of-the-art transformation tools, esp. based on the solutions of the tools to the Hello World case [70] posed at the Transformation Tool Contest 2011; thereby defining the first large-scale study of transformation tools based on a simple and instructive case.

- It introduces a novel taxonomy synthesized from the experiences gained from the TTC cases and their solutions, and compares the tools based on it, leading to a survey of the field, especially explaining what it means for general-purpose tools to be expressive.
- It describes how SHARE [25] was utilized during the TTC, exemplifying how software engineering research events (esp. their reviews) can benefit from cloud based virtual machines. Especially, it links to the SHARE images, which were produced by the tool providers for the Hello World case – they allow a prospective tool user to evaluate a tool in depth before drawing a final decision.
- Overall, it helps researchers in the field of model and graph transformation as well as software engineers searching for a tool to employ to get an overview of the world of transformation tools. It assists the latter in choosing a tool that is well-suited to their needs – much faster than reading through the numerous papers describing the tools, or the detailed case and solution reports, which were published in the Proceedings of the Transformation Tool Contest [99].

2. Hello World Case

The "Hello World!" case consists of a set of simple tasks, each of which can be solved with just a few lines of code. The solutions of the tasks provided by each tool resulted in an extensive set of small, instructive transformation programs. They are too simple to really motivate the strengths of the tools introduced in this article, but they highlight the different approaches taken by them, and they illustrate the languages offered by the tools. Only a small amount of time is needed to understand the case and the solutions.

Below we describe the very first task for illustration, in addition to the task "Deletion of model components", which we have chosen as running example for our introduction into the tools. The two tasks are copied verbatim from the original task description. The other tasks are given in Appendix B as digests. There, a basic model migration and diverse simple queries, as well as simple transformations on graphs are listed. The original descriptions can be found in [70].

The aim of the case has been to cover the most important kinds of primitive operations on models, i.e., create, read, update and delete (CRUD),

Figure 1: The "Hello World" metamodel and the example instance

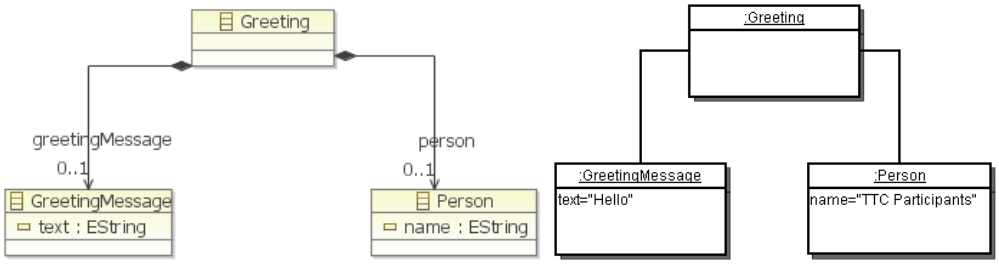

Figure 2: The extended "Hello World" metamodel and the example instance

defining a base for the space of transformation tools with simplified prototypical tasks. The coverage of the subtasks is discussed in detail alongside a number of transformation-related properties in Appendix C.

2.1. Constant Transformation and Model-to-Text Transformation

1(a) Provide a *constant transformation* that creates the example instance of the "Hello World" metamodel given in Figure 1.

1(b) Consider now the slightly extended metamodel given in Figure 2. Provide a *constant transformation* that creates the *model with references* as it is also shown in Figure 2.

1(c) Next, provide a *model-to-text transformation* that outputs the GreetingMessage of a Greeting together with the name of the Person to be greeted. For instance, the model given in Figure 2 should be transformed into the String "`Hello TTC Participants!`".[1]

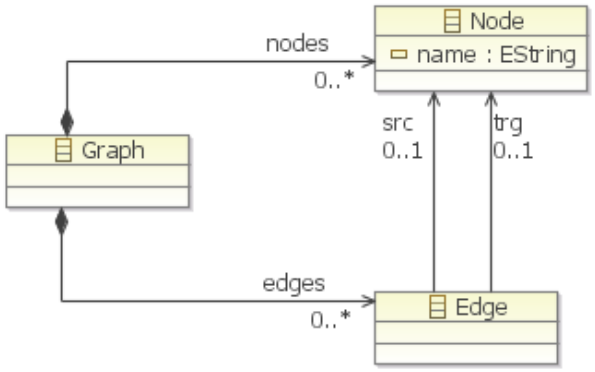

Figure 3: The simple graph metamodel

2.2. Deletion of Model Components

Given a simple graph conforming to the metamodel of Figure 3. Provide a transformation that *deletes* the node with name "n1". If a node with name "n1" does not exist, nothing needs to be changed. It can be assumed that there is at most one occurrence of a node with name "n1". Optional: Provide a transformation that removes the node "n1" (as above) and all its incident edges.

3. The Tools and Their Hello World Solutions in a Nutshell

In the following, we introduce the tools with a calling card. The calling card consists of three parts: first an introduction-in-a-nutshell to the tool is given, which is then followed by an example solution of the *deletion* task of Hello World, and finally closed by a discussion of what the tool is suited for, why so, and what it is not suited for, as seen by the tool's authors.

The purpose of the example is to give an *impression* of the tool and its languages. The "Delete Node with Specific Name and its Incident Edges" task of the Hello World case introduced in Section 2.2 that we use as running example is a small task that illustrates several aspects of processing *structural* information, which defines the functionality at the core of the large majority

[1]Note that we provide as accompanying material on the case website a metamodel, `Result.ecore`, that contains classes for returning primitive results such as strings or numbers.

of these tools. The task including its optional part defines a simple rewriting that involves a small node-edge-node structure, splitting into breadth, which further employs an attribute condition.

Please take a look at Section 4 for a discussion of the implications of choosing this task, and the consequences of the results under display.

3.1. Edapt

Edapt[2] is the official Eclipse tool for migrating EMF models in response to the adaptation of their metamodel. Edapt records the metamodel adaptation as a sequence of operations in a history model [38]. The operations can be enriched with instructions for model migration to form so-called coupled operations. A coupled operation performs an in-place transformation of both the metamodel and the model. Edapt provides two kinds of coupled operations—reusable and custom coupled operations [38].

Reusable coupled operations enable reuse of migration specifications across metamodels by making metamodel adaptation and model migration independent of the specific metamodel through parameters. Currently, Edapt comes with a library of over 60 available reusable coupled operations [39]. *Custom coupled operations* allow to attach a custom migration to a recorded metamodel adaptation. The custom migrations are implemented in Java based on the API provided by Edapt to navigate and modify models.

Edapt's user interface—depicted in Figure 4—is directly integrated into the existing EMF *metamodel editor*. The user interface provides access to the *history model* in which Edapt records the sequence of coupled operations. The user can adapt the metamodel by applying reusable coupled operations through the *operation browser*. When a reusable coupled operation is executed, its application is recorded in the history model. A custom coupled operation is performed by first modifying the metamodel in the editor, and then attaching a *custom migration* to the recorded metamodel changes.

3.1.1. Delete Node with Specific Name and its Incident Edges

Figure 4 shows how the history model looks like for all non-migration tasks of this case. For these tasks, the custom coupled operation always consists of a custom migration, which is attached to an empty metamodel adaptation. The custom migration is implemented as a Java class that inherits from a special super class.

[2] http://www.eclipse.org/edapt

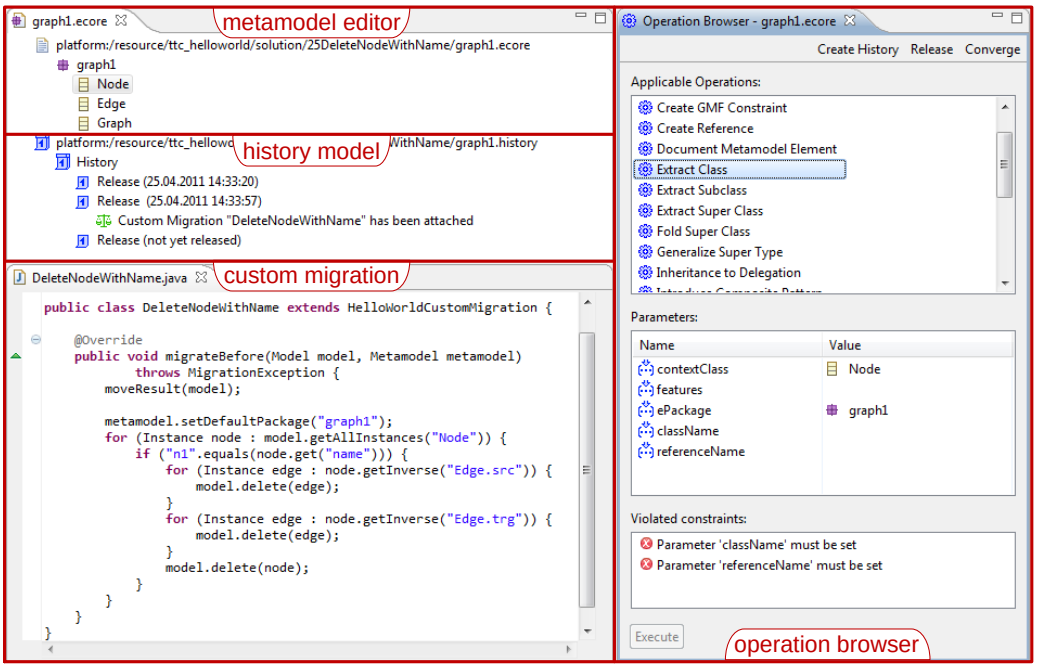

Figure 4: Edapt's user interface

Figure 4 also shows how the deletion task is implemented using the migration language provided by Edapt. The language provides methods to obtain `all Instances` of a class or `get` the value of a feature. The task can be implemented quite easily, since Edapt provides a method to `delete` instances of classes. To also delete all incident edges, we can use the method `getInverse` to navigate to the edges that have the node as source or target.

To avoid unnecessary copying of elements that do not require migration, the transformation in Edapt is always performed in-place. For storing the result at another location, we use the helper method `moveResult` that is provided by the superclass `HelloWorldCustomMigration`.

3.1.2. What is the Tool Suited for and why?

Edapt is tailored for model migration in response to metamodel adaptation. Edapt is based on the requirements that were derived from an empirical study on real-life metamodel histories [36]. The case study showed that metamodels are changed in small incremental steps. As a consequence, Edapt records the model migration together with the metamodel adaptation,

in order not to lose the intention behind the metamodel adaptation.

Moreover, the study revealed that a lot of effort can be saved by reusing migration specifications across metamodels, motivating the need for reusable coupled operations. The migration tasks can be solved by applying only reusable coupled operations. Thereby, not a single line of custom migration code needs to be written. However, the study also showed that in rare cases the migration specifications can become so specific to a certain metamodel that reuse makes no sense. For these cases, Edapt provides custom coupled operations in which the migration is implemented using a Turing-complete Java-based language [37].

While Edapt is tailored for incremental metamodel evolution and model migration, it was not designed to perform model-to-model or model-to-text transformations in general. Even though it is possible to apply Edapt for other use cases, it is more awkward to use, the more one moves away from its original use case. It is certainly not suitable, when there is no difference between source and target metamodel, or when source and target metamodel are completely different from each other.

3.2. EMFTVM

The EMF Transformation Virtual Machine (EMFTVM)[3] [108] is a runtime engine for the ATL Transformation Language (ATL) [47]. Apart from mapping a set of read-only input models to a set of write-only output models – the default execution model for ATL – it supports in-place rewrite rules. The rewrite rules are written in the textual SimpleGT language, and are compiled to the same EMFTVM byte code as ATL. Trace models are generated implicitly, and can be inspected at runtime.

For the Hello World case, solutions written in both ATL and SimpleGT are provided. Because both languages can be composed in a fine-grained way in the VM, one can choose which transformation rules to write in which language for each transformation sub-problem.

3.2.1. Delete Node with Specific Name and its Incident Edges

By default, ATL maps input models to output models. However, the transformation problem at hand is an in-place problem, and ATL provides a *refining mode* for this situation. The ATL solution looks as follows:

[3] http://soft.vub.ac.be/soft/research/mdd/emftvm

```
module graphDeleteN1Incident;
create OUT : Graph refining IN : Graph;

helper context Graph!Edge def : linksToN1 : Boolean =
 self.src.isN1 or self.trg.isN1;
helper context OclAny def : isN1 : Boolean = false;
helper context Graph!Node def : isN1 : Boolean = self.name = 'n1';

rule Node {
 from s : Graph!Node (s.isN1)
}
rule Edge {
 from s : Graph!Edge (s.linksToN1)
}
```

The transformation module `graphDeleteN1Incident` creates a modified `OUT` model from the input `IN` model. Both models conform to the `Graph` metamodel. The paths to the actual (meta-)models are given as runtime parameters. Two helper attributes are defined to abbreviate OCL expressions, as well as to cache the expression values: `linksToN1` and `isN1`. These helper attributes are evaluated in the transformation rules: `Node` and `Edge`. Because it does not have a "**to**" section, the `Node` rule deletes all input elements `s` that are instances of the metaclass `Node` in the metamodel `Graph`, and are in fact "n1". The `Edge` rule deletes all input elements `s` that are instances of `Edge`, and are incident to an "n1" node.

In ATL, the order of the rules does not matter; specifying the deletion of node "n1" before the deletion of incident edges does not prevent matching/applying the edge deletion. EMFTVM implements this using a one-shot matching phase for all rules, followed by a one-shot application phase. Furthermore, deleted elements are merely marked during the application phase, and only processed at the phase end.

The SimpleGT solution for the same problem looks as follows:

```
module graphDeleteN1Incident;
metamodel Graph;
transform g : Graph;

def : n1 : String = 'n1';

abstract rule N1 {
 from n1 : Graph!Node (name =~ env.n1)
 to n1 : Graph!Node (name =~ env.n1)
}
rule DeleteIncomingEdge extends N1 {
 from e : Graph!Edge (trg =~ n1), n1 : Graph!Node
 to n1 : Graph!Node
}
rule DeleteOutgoingEdge extends N1 {
```

```
 from e : Graph!Edge (src =~ n1), n1 : Graph!Node
 to n1 : Graph!Node
}
rule DeleteN1 extends N1 {
 from n1 : Graph!Node
}
```

The SimpleGT transformation module `graphDeleteN1Incident` rewrites the model `g`, which conforms to the metamodel `Graph`. The `n1` helper attribute defines the name of the "n1" node to match. The `N1` rule just matches all "n1" nodes. The output pattern reflects the state of the input pattern after the rule is applied: everything is left intact. This rule is abstract, and is never applied by itself. The `DeleteIncomingEdge` rule extends the `N1` rule, and is applied *as-long-as-possible* to all input patterns `(e, n1)`, where `n1` is an "n1" node (inherited behavior), and `e` is an edge targeting `n1`. The output pattern no longer contains `e`, so it is deleted. The `DeleteOutgoingEdge` rule does the same, but for edges that depart from `n1`. Then, the `DeleteN1` rule is applied *as-long-as-possible* to all `n1` nodes. There is no output pattern, so the entire input pattern match is deleted.

3.2.2. What is the Tool Suited for and why?

EMFTVM focuses on reuse, modularization, and composition of model transformations. It is therefore well-suited to specifying large and complex transformations. EMFTVM is the third generation VM for ATL, adding new ATL features as well as improving performance.

EMFTVM provides cross-language internal composition by defining the composition mechanisms, module import and rule inheritance, at the VM level. The EMFTVM is based on the Eclipse Modeling Framework (EMF), and can share its models with other EMF-based tooling.

EMFTVM currently provides compilers for ATL, SimpleGT, a minimal graph transformation language on top of EMF, and EMFMigrate [107], a model migration language for EMF.

The tool is not suited for model-to-text-transformations. Furthermore, the matching engine is not able to compete with the execution speed reached by matching engines of high-performance rewriting tools.

3.3. Epsilon

Epsilon is a component of the Eclipse Modeling Project[4] and a family of model management languages. Epsilon seeks to capture patterns of – and best practices for – model management. Specifically, Epsilon provides several inter-related task-specific languages. Each language makes idiomatic patterns and concepts that are important for a specific model management task. For example, Epsilon Flock [85] provides constructs for updating a model in response to changes to its metamodel.

To solve the Hello World case, three Epsilon languages were used. The Epsilon Object Language (EOL) [56] – which is the base language of Epsilon and is an extension to and reworking of OCL – was used for direct model manipulation, Epsilon Flock was used for model migration and rewriting, and the Epsilon Generation Language [87] was used for model-to-text transformation. For each problem in the Hello World case, we have chosen the Epsilon language that provided the constructs that we feel were most well-suited to solving that category of problem.

3.3.1. Delete Node with Specific Name and its Incident Edges

We solved the node deletion task with EOL [56] and with Flock. The latter is more concise, but arguably more difficult to understand. In EOL, the `delete` keyword removes a model element and all nested model elements from a model. To delete a `Node` and its incident `Edges`, three[5] delete statements have been used (Listing 1).

```
1 var n1 : Node = Node.all.selectOne(n|n.name == "n1");
2
3 delete n1.incoming();
4 delete n1.outgoing();
5 delete n1;
6
7 operation Node incoming() : Collection(Edge) {
8  return Edge.all.select(e|e.trg == self);
9 }
10
11 operation Node outgoing() : Collection(Edge) {
12  return Edge.all.select(e|e.src == self);
13 }
```

Listing 1: Deleting a node and its incident edges with EOL.

[4] http://www.eclipse.org/epsilon

[5] A single, cascading delete statement could have been used if containment references had been used in the graph metamodel.

An alternative solution, using a Flock migration strategy, is shown in Listing 2. Like all of the task-specific languages in Epsilon, Flock re-uses and extends EOL with additional language constructs. For example, Flock provides the `delete` construct for specifying model elements that should be removed for a model. Deletions are guarded using the `when` keyword. The difference between the two solutions is subtle, but important. The Flock solution is declarative, and the Flock execution engine consequently has complete freedom over how the deletion of the node and edges is scheduled. In contrast, the EOL solution is imperative, and the EOL execution engine must first find the "n1" node, then delete its edges, and then delete the node itself.

```
1 delete Node when: original.name == "n1"
2
3 delete Edge when: original.src.name == "n1" or original.trg.name == "n1"
```

Listing 2: Deleting a node and its incident edges with EOL.

3.3.2. What is the Tool Suited for and why?

Epsilon appears to be well-suited for many common model management tasks. This claim is supported by the use of Epsilon by numerous industrial partners, including in ongoing collaborations with BAE Systems [11], IBM Haifa, Telefonica, Western Geco, Siemens, and the Jet Propulsion Laboratory at NASA. Additionally, the Universities of Texas, Oslo, Kassel and Ottawa teach MDE by using Epsilon. A further benefit of Epsilon is that it is technology-agnostic: model management operations written in Epsilon languages are independent of the technology used to represent and store models. Epsilon can be used to manage XML, EMF, MDR, BibTeX, CSV, and many other types of models.

Due to its task-specific languages, Epsilon is well-suited to solving a range of model management problems, such as model transformation, merging, comparison, validation, refactoring and migration. The opponents assigned to the Epsilon solution described in this paper remarked that most of the solutions to the Hello World are very concise and readable when formulated with Epsilon. Counter to this, one of the opponents suggested that learning the similarities and differences between the family of languages might be a challenge for new users of Epsilon.

At present, Epsilon is not well-suited for matching complicated patterns. The opponents remarked that solutions that required matching complicated patterns (such as finding cycles of three nodes in a graph) were less concise and readable due to the use of imperative constructs for specifying patterns

in EOL. Additionally, not all of the Epsilon languages scale well for very large models in some situations. We have tailored Epsilon for use with large models to solve the specific problems of industrial partners (e.g., [11]), and we are beginning to address more general issues of scalability in Epsilon as part of our ongoing research at York (e.g., [2]).

3.4. GReTL

GReTL (*Graph Repository Transformation Language*, [42, 15]) is a graph-based, extensible, operational transformation language. Transformations are either specified in plain Java using the GReTL API or in a simple domain-specific language. GReTL follows the conception of incrementally constructing the target metamodel together with the target graph, a feature distinguishing it from most if not all other transformation languages. When creating a new metamodel element, a set-based semantic expression is specified that describes the set of instances that have to be created in the target graph. This expression is defined as a GReQL query [14] on the source graph.

For transformations with pre-existing target metamodel like in the Hello World case, there are also operations with the same semantics working only on the instance level.

GReTL is a kernel language consisting of a minimal set of operations, but it is designed for being extensible. Custom higher-level operations can be built on top of the kernel operations. This extensibility was exploited to add some more graph-replacement-like in-place operations for solving the Compiler Optimization case [10].

3.4.1. Delete Node with Specific Name and its Incident Edges

As said, GReTL can be extended. To compete in the TTC Compiler Optimization case, several in-place-operations with semantics similar to graph replacement systems were added.

The following operation call deletes all nodes of type `Node` whose `name` attribute equals "n1" and its incident edges.

```
1 transformation DeleteNodeN1AndIncidentEdges;
2
3 Delete <== from n: V{Node}
4            with n.name = "n1"
5            reportSet n, n <--{src, trg} end;
```

The first line simply declares the transformation. In line 3, the `Delete` operation is invoked. It receives a (possibly nested) collection of elements to be deleted. Those are specified with the GReQL query following the arrow

symbol. It selects all vertices of type `Node` whose name is "n1" together with their incident edges.

In TGraphs, deleting a vertex also deletes incident edges; there cannot be dangling edges. However, since the example model represents edges as vertices of type `Edge` that refer to their start and end `Node` with `src` and `trg` edges, a *regular path expression* is used to select the `Edge` nodes that start (`src`) or end (`trg`) at the `Node n` that is to be deleted, in order to delete them, too.

3.4.2. What is the Tool Suited for and why?

Due to GReQL's regular path expressions, GReTL is especially suited for model transformation tasks similar to the Program Understanding case [40], where complex non-local structures have to be matched in the source graph.

GReTL is not suited for transformations of EMF models because it is implemented for (and included in) JGraLab[6], which uses TGraphs as model representation. However, EMF models can be imported/exported.

Also, GReTL was designed for model transformations that create a new metamodel (optionally), and thereby also create a new instance graph. The support for in-place transformations has been added only for competing in the TTC cases, and it does not really fit into the original conception of GReTL except that it proves its extensibility.

3.5. GrGen.NET

GrGen.NET[7] [46] is an application-domain-neutral graph rewrite system with a focus on performance, expressiveness, programmability, and debugging. It offers textual languages for graph modeling and rule specification, which are compiled into .NET assemblies, and a textual language for rule application control, which is interpreted by a runtime environment. The user interacts with the system via a shell application and a graph viewer (alternatively via an API) allowing for graphical and step-wise debugging.

3.5.1. Delete Node with Specific Name and its Incident Edges

Rules in GrGen consist of a pattern part specifying the graph pattern to match and a nested rewrite part specifying the changes to be made. The example rule `deleteN1AndAllIncidentEdges` below matches a node `n` of

[6] http://www.jgralab.uni-koblenz.de
[7] http://grgen.net

type `graph1_Node`, `if` it bears the name searched for. The incident edges are collected with the `iterated` construct, which munches the contained pattern eagerly as long as it is available in the graph and not yet matched. The contained pattern here consists of a graphlet `n <-- e:graph1_Edge` that specifies an anonymous edge of type `Edge` leading from the source node `e` to the target node `n`. The rewrite part is specified by a `replace` block nested within the rule; graph elements that are declared in the pattern but are not referenced in the replace-part are deleted. Since the `replace` parts are empty, all matched elements are deleted.

```
rule deleteN1AndAllIncidentEdges {
    n:graph1_Node;
    if {n._name == "n1";}

    iterated {
        n <-- e:graph1_Edge;

        replace { }
    }
}
```

The rule is executed from the rule application control language with the syntax `exec deleteN1AndAllIncidentEdges`. When the rule is executed in the debugger of the shell, you can watch how it is applied on the host graph as illustrated by the screenshot shown in Figure 5.

3.5.2. What is the Tool Suited for and why?

GrGen.NET is suited to tasks requiring notational expressiveness, with its support for rewriting structures that extend into depth *and* into breadth, notably structures with a tree backbone [45], as they are required for processing natural languages in computer linguistics [3]. Another highlight is its support for retyping of graph elements.

The tool fits well to performance-oriented tasks, with its optimized code generator yielding high execution speed at modest memory consumption, and its support of matching large patterns. These qualities are of interest in the domain of compiler construction [89] where GrGen originates from.

GrGen.NET is a general-purpose graph rewriting tool. To this end it offers a rich metamodel and a highly programmable control language, which allows to program even a state space enumeration, a concept not built in as such. GrGen.NET was employed in mechanical engineering [33], architecture [26], and bio-chemistry [88].

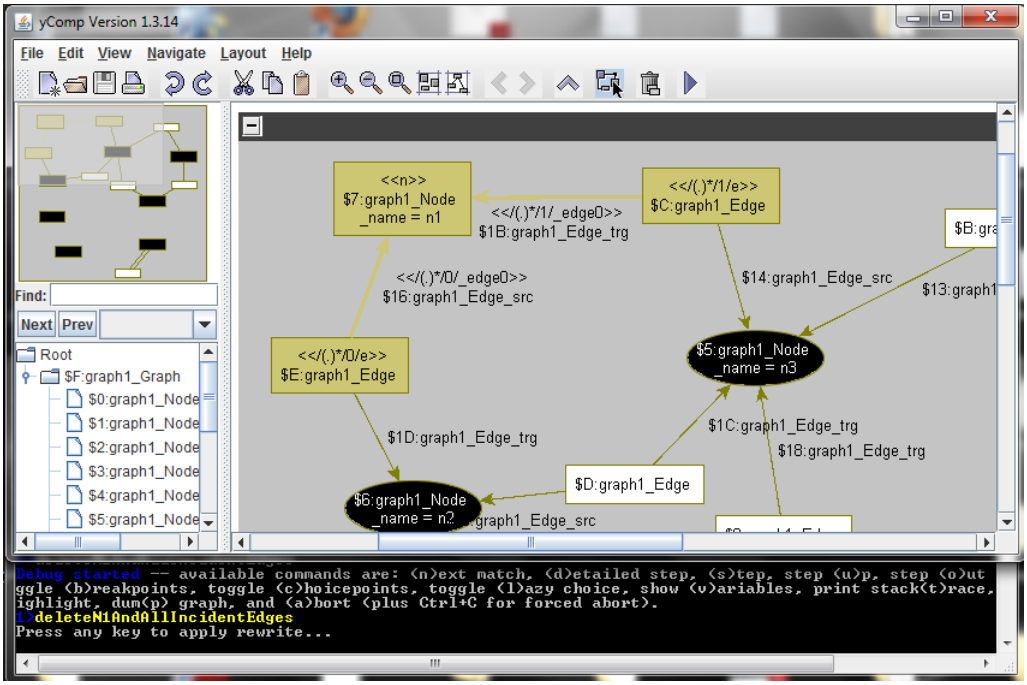

Figure 5: Screenshot from the GrGen.NET debugger

The tool is well adapted to being used by external developers, due to its extensive user manual [7], and due to the languages, which were designed to be understandable to a common software engineer, with a direct representation of graphical patterns in a textual notation.

GrGen.NET is *not* well suited for EMF/.ecore based transformations because of the name mangling applied by the importer, and the lack of an exporter. The implementation with a code generator resp. compiler (to achieve high-performance solutions) slows down the test-debug cycle, and rules out (programmatic) changes to the rules at runtime. Tool-internal meta-programming is not supported, so for tasks requiring large amounts of highly repetitive specifications (that cannot be factored out into subpatterns) an external code generator emitting a GrGen specification needs to be employed.

3.6. GROOVE

GROOVE[8] is a general-purpose graph transformation tool offering a user-friendly user interface and a very expressive rule language. Its main distinguishing feature is automatic exploration and analysis of the complete space of reachable graphs, using any of a number of search strategies and other settings. The analysis capabilities include LTL and CTL model checking [53] and Prolog queries [20]. See [23] for a recent overview of the tool and its use in practice.

In a typical usage scenario, GROOVE is accessed through its built-in GUI, but for the analysis of predefined grammars there is a headless, command-line version available (offering better performance). A screenshot of the GUI is shown in Figure 6.

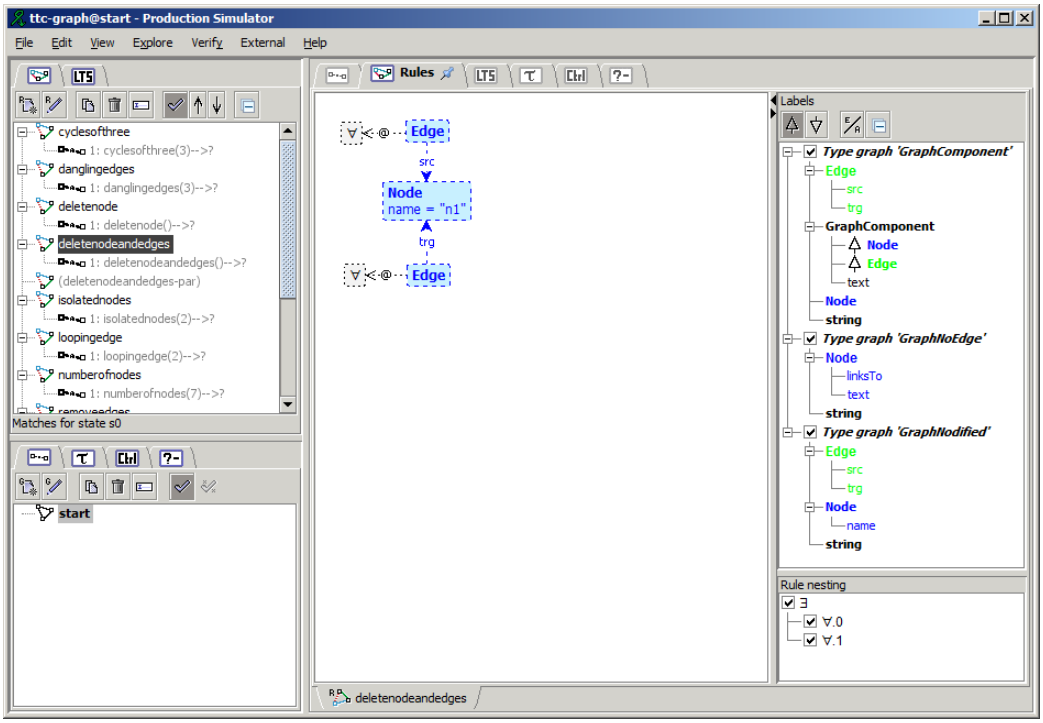

Figure 6: Screenshot of the GROOVE simulator

In the way of interoperability, GROOVE supports import from and export

[8] http://sf.net/projects/groove

to a number of external formats, including EMF, as well as a limited form of textual output.

3.6.1. Delete Node with Specific Name and its Incident Edges

It is typical of the power of the GROOVE rule language that each task of the Hello World case can be solved in a single rule that captures precisely the desired functionality or effect. For instance, Figure 7 shows the GROOVE rule implementing the task described in Section 2.2, namely to delete a node with a given name (here "n1") and all its incident edges.

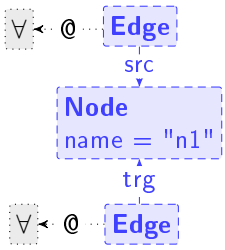

Figure 7: GROOVE rule for deleting a node and its incident edges

The phrase "all its incident edges" gives rise to the two nodes labeled ∀ in the graph, each of which universally quantifies over the patterns connected to it by a `@`-labeled edge. Thus, the upper ∀-node captures all incoming edges, and the lower one all outgoing edges. The dashed (blue) outline of the nodes is the visual representation of the fact that they are deleted when the rule is applied.

3.6.2. What is the Tool Used for and why?

Experience has shown that GROOVE is very easy to use for prototyping all kinds of systems. In essence, any scenario involving the dynamics of a system that has a natural representation as graphs is amenable to modeling in GROOVE.

The advantage of building such a model is especially the ease with which the resulting behavior can be analyzed, both through simulation and visualization and through the automatic exploration of the set of reachable graphs. Central to this capability is the notion of a *labeled transition system* (LTS), which shows precisely how graphs may evolve under rule application: in this view, every reachable graph is itself a node (i.e., state) in the LTS, and every rule application corresponds to an edge (i.e., a transition).

In settings where the issue is not to model the dynamics of a system but to specify a (mode-to-model) transformation, GROOVE is still a useful tool as it can show, on concrete example models, that the rule system specifying the transformation is confluent and terminating — namely, this is the case if and only if the LTS is acyclic and has only a single terminal state.

The Hello World case offers no opportunity to demonstrate this capability, but in [23] we review examples from several, very distinct domains.

Given the fact that GROOVE is really a general-purpose graph transformation tool, it should come as no surprise that it is less suited for dedicated model transformation applications. On the one hand, there is no built-in support for model transformation (all aspects of the transformation would have to be constructed manually); on the other hand, the strength of GROOVE, namely the capability to explore and analyze state spaces, is wasted in the context of model transformation, where the purpose is really to have a confluent rule system that can be explored in a linear fashion.

Related to this, model-to-text transformation is also outside the scope of GROOVE: in addition to sharing the characteristics of model-to-model transformation mentioned above, there is the simple fact that GROOVE does not support unformatted textual output.

3.7. Henshin

Henshin[9] [1] is a high-level graph rewriting and model transformation language and tool environment for Eclipse. The transformation language of Henshin is based on declarative and procedural features. Complex transformations can be specified in a modular and reusable way using nested rules and a small set of control-flow structures. The specification of transformations is supported by a compact visual syntax provided in a graphical editor. For formal analysis, Henshin includes a state space generation and model checking tool, and an export functionality to external model checkers and other graph rewriting tools, such as AGG [93].

3.7.1. Delete Node with Specific Name and its Incident Edges

Figure 8 shows an example rule in Henshin for deleting a node with a given name and all its incident edges (modeled here by nodes of type `Edge`). The parent graph and the node name are supplied as rule parameters. To

[9] http://www.eclipse.org/henshin

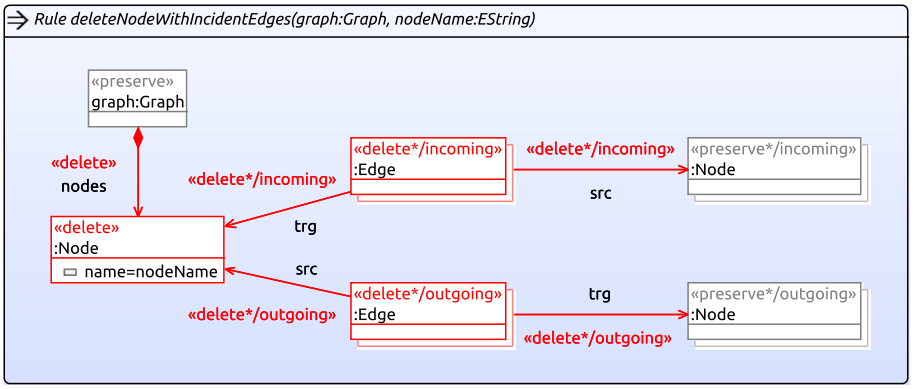

Figure 8: Henshin rule for deleting a node and its incident edges.

implement the deletion of all incident edges, the rule contains two nested rules, called `incoming` and `outgoing`, which are matched and applied as often as possible. Note that these two nested rules are specified indirectly with the stereotypes used on nodes and edges, e.g., ⟪`delete*/incoming`⟫. The nesting of such rules is specified using a simple path-like syntax, e.g., ⟪`delete*/x/y/z`⟫, where `x`, `y` and `z` are the names of nested rules. This allows the user to define complex transformations in a single rule and without the need for any control-flow structures, such as loops. However, control-flow structures as well as composite application conditions and attribute calculations based on scripting languages are also supported.

3.7.2. What is the Tool Suited for and why?

Henshin targets the transformation of structural data models in the Eclipse Modeling Framework (EMF). EMF is an implementation of a subset of the MOF standard by the OMG and is the basis of a wide range of tools and frameworks, including editors for domain-specific languages as well as an implementation of UML 2.4.1 (at the time of writing). Henshin is suited to define transformations for these languages. Since Henshin implements a rewrite approach, it can be used to modify models in-place, such as required for refactorings. Additionally, Henshin provides a generic trace model to support the specification of transformations from one language to another. For example, Henshin has been used in an industrial case study [44] to implement an automatic translation of programming languages for satellite technology to a standardized satellite control language, called SPELL. Due to its state

space generation facilities, Henshin can be also used for formal verification. For instance, Henshin has been used for a quantitative analysis of a probabilistic broadcasting protocol for wireless sensor networks [58].

Since Henshin does not include a template language, it is not suited for model-to-text transformations. Furthermore, out-place transformations where major parts of the input model have to be copied are not well supported, because this copying has to be defined explicitly.

3.8. MDELab SDI

Story diagrams [17, 105] are a visual domain specific language similar to UML activity diagrams with an easily comprehensible notation for expressing graph transformations. Activity nodes and edges describe control flow and so-called story nodes can be embedded, which contain graph transformation rules.

Story diagrams can be interpreted by the MDELab Story Diagram Interpreter (SDI)[10] [24], which features a dynamic pattern matching strategy when executing the graph transformation rule inside a story node. At runtime, this strategy adapts to the specifics of the instance model on which the graph pattern matching is executed to improve performance. Furthermore, the SDI provides seamless integration with EMF and supports OCL to express constraints and queries in a story diagram. A debugger allows to execute story diagrams step-wise (including back-stepping), to inspect and modify the state and the diagram itself, as well as to visualize the execution.

3.8.1. Delete Node with Specific Name and its Incident Edges

The problem to delete a particular node with its incident edges can be split into four steps: (1) Finding the node, (2) deleting the incoming edges, (3) deleting the outgoing edges, and (4) deleting the node itself. The story diagram in Figure 9 contains a story node for each of these steps. First, the *graph* that is modified has to be provided as a parameter. Then, the node will be sought. An OCL constraint is used to check the name of the node. If one can be found, its incoming and outgoing edges are deleted. This is done in the subsequent *for-each* nodes, i.e., the contained graph transformation rules are executed for all matches that can be found. Finally, the node itself is deleted. Instead of hard-coding the name of the node to delete, it is also possible to provide the name as an additional parameter.

[10] http://www.hpi.uni-potsdam.de/giese/gforge/mdelab/

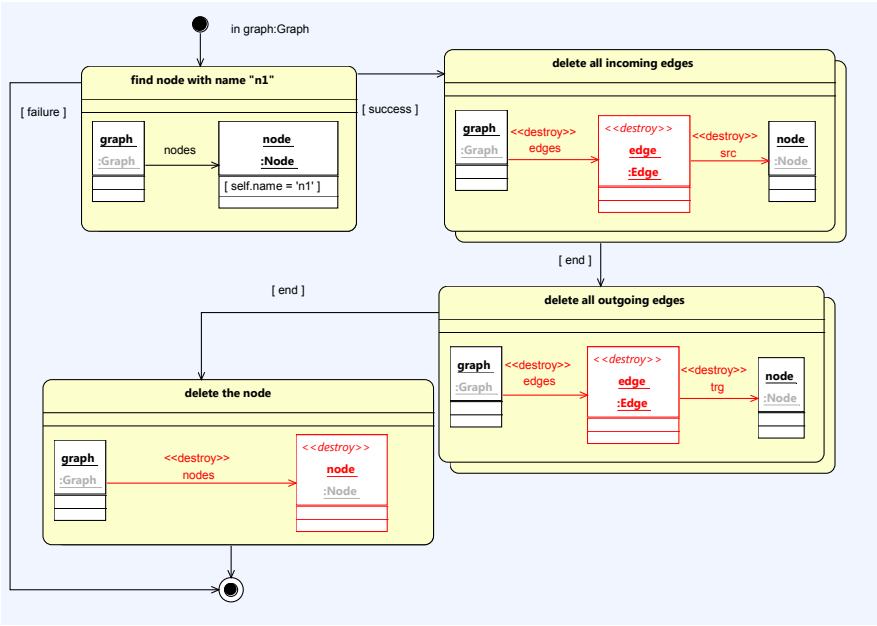

Figure 9: Searching for node *n1*, deleting its incoming and outgoing edges, and the node itself.

Arguably, this solution is much more verbose than the solutions of other, especially textual, languages. However, the advantage of this graphical notation becomes clear if the patterns are complex and contain many links. Then, a corresponding textual representation would be much harder to understand.

3.8.2. What is the Tool Suited for and why?

As already stated, a major strength of story diagrams is their good comprehensibility, especially for people unfamiliar with graph or model transformation tools. This is supported by allowing to use OCL expressions in any place, where constraints or queries can be used. They can be executed by an interpreter, which offers high flexibility because it allows to generate story diagrams at runtime and execute them right-away. This feature is used, e.g., in [19], where behavior models are derived from requirement specifications. The interpreter simulates different scenarios in several iterations and derives a valid story diagram, which reflects and fulfills the requirements. Besides the interpreter, a graphical editor with model validation and a graphical debugger are also provided. Another advantage is the tight integration with EMF, which allows to use other EMF-based tools with story diagrams.

Story diagrams are primarily designed for in-place transformations. Therefore, they lack common features of model-to-model transformation tools like implicitly created traceability links. Moreover, all patterns in a story diagram are matched using a local search. Hence, a binding must be provided for at least one object in the LHS of each pattern. However, this is usually not a major restriction in practice because model elements are mostly contained in a common container, which has to be included in the patterns, e.g. *graph* in Figure 9.

3.9. metatools

metatools[11][66, 67, 96, 97] take the opposite approach to dedicated execution machines: They support a declarative style of programming by integrating high-level transformation devices, like visitors, rewriters, pattern matching, relational algebra, etc., into a general-purpose programming language (currently: Java), as seamlessly as possible. For this, a small run-time library co-operates with source code that is *generated* from compact declarations in dedicated domain specific languages (DSLs). Applications are founded on this generated code, but are otherwise totally free to use any features of the hosting programming language and its libraries.

The reader is kindly invited to look behind the façade: The second source text fragment below looks like plain Java code, but indeed only initiates the fully automated rewriting machine, which has been generated according to the model definition in the first fragment.

3.9.1. Delete Node with Specific Name and its Incident Edges

metatools do not come with one fixed definition of "graph", but, contrarily, apply graph theory to arbitrary data structures. In the context of the "hello world" test case, the properties of a graph and the mapping of its components to Java objects are subject to explicit design decisions. The following solution assumes that the graph is rooted, i.e., can be represented by one single node, and may contain cycles, but no "dangling" edges, and that edges are directed and named. (The SHARE demo [65] shows an alternative.)

The first code snippet shows the declaration from which the source code for objects and visitors is generated. It is written in `umod`, one of metatools' DSLs. The class hierarchy is defined by indentation. Fields may have (fully

[11] http://bandm.eu/metatools

compositional!) collection types. Constructor signatures and visitor/rewriter traversal rules may follow the field definitions: The instructions `V 0/0` mean that all visitors using the traversal rule "0" go from each `Node` to all `Edge` objects contained in `.outgoing`, and from there to the `Node` in `.target`. The instructions `C 0/0`, `C 0/1`, etc., mean that there is only one public constructor per class, which takes the value to initialize the field `.name` for `Node` objects, and the values for `.name`, `.src` and `.trg`, in this sequential order, to create `Edge` objects.

```
MODEL Model =

VISITOR 0 Rewriter IS REWRITER ;

TOPLEVEL CLASS
Item
    name        string          ! C 0/0 ;
| Node
    outgoing    SET Edge        !       V 0/0 ;
| Edge
    src         OPT Node        ! C 0/1  ;
    trg         OPT Node        ! C 0/2  V 0/0 ;
END MODEL
```

The second snippet shows how a node with a given name is deleted by applying rewriter code, which is derived from the generated rewriter. Only the rewriter's local behavior needs to be specified, all traversal and the update of all member fields is done by the generated code.

```
 public static Node deleteNodeAndEdges  (final Node root,
                                         final String nodeName){
   return new Rewriter(){
     public void action(final Node n){
       if (nodeName.equals(n.get_name()))
         substitute(null);
       else
         super.action(n);
     }
     public void action(final Edge e){
       if (rewrite(e.get_trg())==null)
         substitute_empty();
       else
         super.action(e);
     }
   }.rewrite_typed(root);
 }
```

3.9.2. What is the Tool Suited for and why?

The *generative* approach of the metatools determines their applicability: Only small runtime libraries must be packaged with an application. All integration of software components is done by the normal protocols of the hosting

language, "packages" in horizontal and "inheritance" in vertical direction.

Therefore metatools are well suited when legacy sources shall be combined with more declarative techniques in an incremental way. This may even include use of opaque libraries: e.g. the "paisley" pattern matching subsystem is fully compatible with arbitrarily-shaped pre-defined data [96].

They are also well suited when experienced programmers want to retain immediate use of the full range of the hosting programming language and its libraries, stay with the tools they are used to, and do not want to change their style of coding completely. Features, tools and strategies can be employed selectively, according to the user's needs and preferences.

The output of metatools is source code, which can be treated together with hand-written code in a uniform way, by humans (inspecting) and by tools (generating doc, debugging, profiling, etc.). There is no "magic behind the scene", so programmers have full control over the applications' behavior, if they want to.

metatools include advanced support for XML import, export and processing, as long as the format is given as a W3C DTD.

They have been successfully employed in different industrial and academic medium-scale professional programming projects, in the fields of compiler construction for DSLs, web content management, financial systems, etc.

Since all typing issues in metatools are checked, resolved and decided as strict and as early as possible, there are longer programming turnarounds times than in dynamically typed systems or systems with dynamic metamodels. Currently there is *no support* for IDE integration, and for XML type definitions beside DTD. Esp. there is *no* connection to Eclipse and to EMF. But the SHARE demo on the "compiler optimization task" [65] shows how easily a new XML based file format (here: `gxl-1.0`) can be connected to metatools.

3.10. MOLA

MOLA[12] [51] is a graphical general-purpose transformation language developed with a focus on comprehensibility. It is based on traditional concepts among transformation languages: pattern matching and rules defining how the matched pattern elements should be transformed.

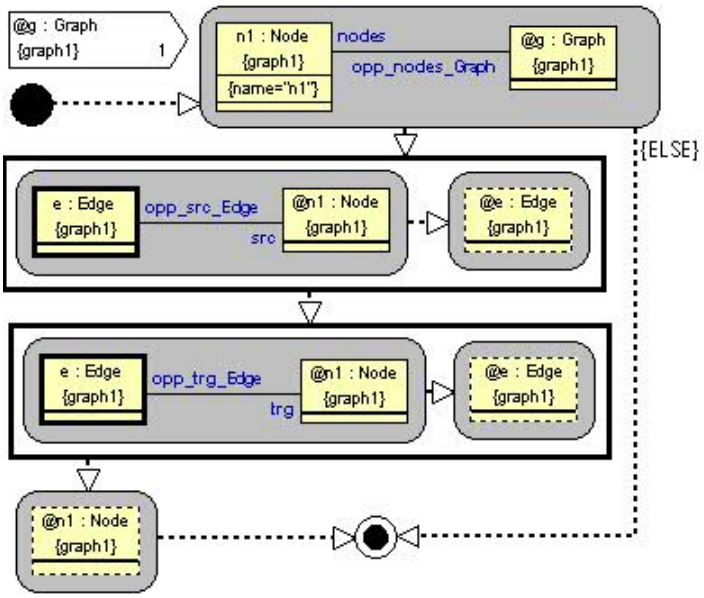

Figure 10: MOLA transformation for deleting a node named "n1" and its incident edges.

3.10.1. Delete Node with Specific Name and its Incident Edges

MOLA is a procedural transformation language. The MOLA procedure solving this task is given in Figure 10. The structure of a MOLA procedure is in a sense similar to UML activity diagrams. The execution order of MOLA statements is determined using Control Flows (closed arrows with dashed line). The key element of the MOLA language is a rule (gray rounded rectangle), which contains a declarative pattern that specifies the instances of the classes that must be selected and how they must be linked. The first rule is used to find a `Node` named `n1` in a graph. The graph to be processed `(@g)` is given as a parameter to ensure that only nodes in this graph are examined. We check with an association link that the `Node` is contained in the `Graph`. An OCL-like constraint is used to check the `name` of the `Node`. Here the node name is directly used in the expression, however a variable containing the name of the node could have been used as well.

When the node `n1` was found, then two similar MOLA foreach loops (rectangles with bold border) are executed to process the outgoing and incoming edges respectively. As a pattern is only matched once, we have to

[12] http://mola.mii.lu.lv

employ a loop to process all the edges. Each loophead (first rule in a loop) contains an iterator – the loop variable, a class element depicted with a bold border. A loop is executed for each distinct instance of the loop variable satisfying constraints defined by the loophead. Here, the constraint is that the edge's source (respectively, target) is the `n1` node. The second rule in the loop deletes this edge (deletion is marked using a dashed border). Afterwards the `n1` node itself is deleted using a similar MOLA rule. Execution ends by reaching the end symbol.

3.10.2. What is the Tool Suited for and why?

The main design goals of the MOLA language are readability and comprehensibility of the transformations. Therefore, MOLA is a graphical transformation language. The comprehensibility of the MOLA language helps to reduce the number of errors in a transformation definition.

MOLA has rich pattern definition facilities. In many cases a solution to a complicated transformation task can be defined in MOLA with a few rules. This way it is possible to solve many real transformation tasks easily, where it is required to process languages with complicated metamodels like UML. This is proved by a case study performed in the ReDSeeDS project [91], where MOLA was used to process UML models in a model-driven application development. MOLA is well suited for building tools for graphical languages with complicated domain metamodels and dissimilar presentation models. An example is the MOLA editor [52] built in the METAclipse framework.

MOLA offers an Eclipse-based graphical development environment – the MOLA tool, incorporating all the required development support: a graphical editor (with support for graphical code completion and refactoring), a syntax checker and a compiler to three different target environments including EMF. The MOLA tool has a facility for importing existing metamodels, including the EMF (Ecore) format.

MOLA is a general-purpose transformation language, so for specialized tasks, such as model-to-text transformations or model migrations, specialized languages perform better. The same could be said about simple model navigation and look-up transformations where languages such as lQuery [69] or EOL [56] perform better. It should also be noted that there are no constructs in the MOLA language that allow modifying the metamodel or the transformation rules at runtime. Though, it is possible to generate MOLA rules before the execution using HOTs (Higher-Order Transformations), for example in Template MOLA [50]. In addition, the current version of the

MOLA tool is also not well suited for tasks requiring high performance.

3.11. QVTR-XSLT

QVT Relations (QVT-R) is a declarative model transformation language proposed by the OMG as part of the Query/View/Transformations (QVT) standard. QVT-R has both textual and graphical notations; the latter provides a concise, intuitive and yet powerful way to specify transformations. In QVT-R, a *transformation* is defined as a set of *relations* (rules) between source and target metamodels, where a relation specifies how two object diagrams, called *domain patterns*, relate to each other. Optionally, a relation may have a pair of *when*- and *where*-clauses specified with OCL to define the pre- and post-conditions of the relation, respectively.

The QVTR-XSLT tool [68] supports the graphical notation of QVT-R, and the execution of a subset of QVT-R by means of XSLT programs. It consists of two parts: (1) a graphical editor that is used to define the metamodels and specify the transformation rules, and (2) a code generator that automatically generates executable XSLT programs for the transformations.

3.11.1. Delete Node with Specific Name and its Incident Edges

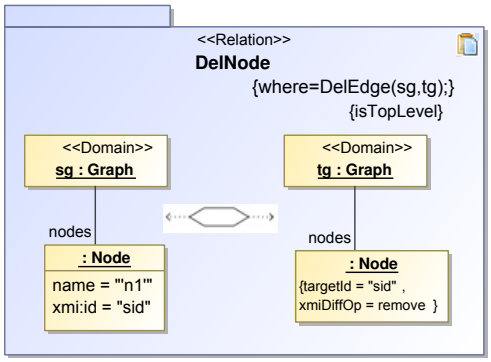

Figure 11: Delete a node with name "n1"

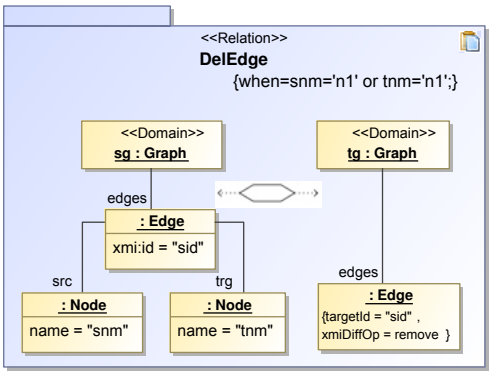

Figure 12: Delete the incident edges

This task is accomplished by a simulated in-place transformation, which is defined in QVTR-XSLT by modification annotations (insert, remove, replace) of the existing model elements. The metamodel has been already defined in Figure 3. The transformation consists of two relations shown in Figure 11 and Figure 12.

The top-level relation `DelNode` is the starting point of the transformation. Its source domain pattern (left part) matches a `Node` named "`n1`" in a graph. If the "n1" node is found, its identifier (`xmi:id`) is bound to variable `sid`, and then the target domain pattern (right part) marks the `xmiDiffOp` of the node whose id is the `sid` as `remove`. Invoked from the `where`-clause of relation `DelNode`, the source domain pattern of relation `DelEdge`, along with its `when`-clause, is used to find all the `Edges` whose `src` or `trg` nodes are the "n1" node. Similarly, these edges are marked as `remove` in the target domain pattern.

An XSLT program of about 80 lines of code is generated for the transformation. Execution of the program copies all the model elements from the input model to the output model, except the ones marked as `remove`.

3.11.2. What is the Tool Suited for and why?

The QVTR-XSLT tool can be applied to a wide variety of source-to-target model transformations, under the condition that the source and target models are stored in XML documents, and each model element has an unique identifier. Many transformation scenarios, such as platform independent model (PIM) to platform specific models (PSM) transformations, or the transformations of UML models to models of formal languages (such as CSP), are well supported. Data transformation in data engineering is another potential application field for the tool. Using the tool, the structures of the XML data can be described concisely by the metamodels, and the mappings between the data can be specified by relations of QVT-R using the high-level graphical notation. In addition, the tool can be used in the fields of semantic web and ontologies, often there is a need to convert between different knowledge models, which are also in XML formats.

The generated XSLT programs for the transformations can be directly executed under any XSLT processor on any platform, or can be easily integrated into other applications and systems. As there are already many industrial-strength XSLT processors, such as *Saxon* and *Xalan*, our transformations can then run fast, and efficiently process large-scale models.

Using the tool, we have successfully designed complicated transformations that work within CASE tools. One transformation may include more than 100 rules and queries, and generate more than ten thousands lines of XSLT code.

QVTR-XSLT is *not* suited for model-to-text transformations. Furthermore, it only supports models with unique identifiers.

3.12. UML-RSDS

UML-RSDS[13] (Reactive system design support, [60]) is a general-purpose language and tool for model-based development. The language is a precise subset of UML and OCL, in which software systems can be specified and designed using UML class diagrams, use cases, state machines and activities.

In UML-RSDS a transformation specification is expressed by:

1. A class diagram, showing the source and target metamodels of the transformation.

2. A use case, defining the transformation effect. A use case can have a precondition, defining assumptions made about the source and target models at the start of the transformation. It also has a postcondition, defining the intended state of the source and target models at the end of the transformation.

3. The postcondition constraints are written in a subset of OCL, which has a unique procedural interpretation: from these a design (as a UML activity) and executable code (in Java or C#) can be automatically synthesized.

Source and target metamodels are defined using the visual class diagram editor of UML-RSDS.

3.12.1. Delete Node with Specific Name and its Incident Edges

The named node deletion transformation on graphs is an update-in-place transformation, which operates on models of the metamodel shown in Figure 13. This transformation has the following two postconditions:

```
n : src.name or n : trg.name  implies  self->isDeleted()
```

operating on instances of `Edge` (the notation `n : src.name` abbreviates `src.name->includes(n)`), and

```
name = n  implies  self->isDeleted()
```

on instances of `Node`, where `x->isDeleted()` expresses that `x` is removed from the model.

Logically, these postconditions can be read as expressing that in the end state, there are no edges that have a source or target node with name `n`, and

[13] http://www.dcs.kcl.ac.uk/staff/kcl/uml2web

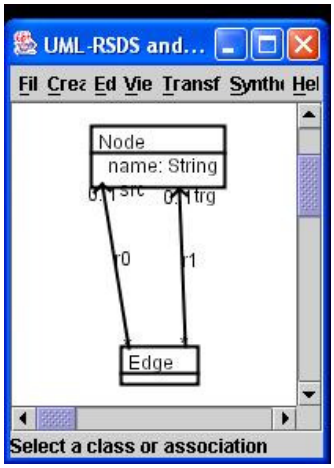

Figure 13: Graph metamodel in UML-RSDS

that all such nodes have also been removed. Operationally, the constraints define two transformation rules that remove the edges first, then remove the nodes. We need to remove the edges first, in order to avoid introducing dangling edges without target or source nodes during the transformation.

3.12.2. What is the Tool Suited for and why?

UML-RSDS has been successfully used for all categories of transformation, except text-to-model or model-to-text. It is particularly suited for update-in-place transformations (refactoring, restructuring, etc.), and for refinements and abstractions. It can be used for migration transformations, e.g., [62], however it does not have specific support for defining migrations. Similarly, it does not have specific support for model-to-text or model-merging transformations, but these can be defined. It supports the definition and instantiation of generic transformations.

UML-RSDS has the advantage of using standard UML and OCL notations to specify transformations, reducing the cost of learning a special-purpose transformation language. It also has the advantage of making explicit all assumptions on models and providing global specifications of transformations, independent of specific rules. Verification support is provided for proving transformation correctness [64]. For the above example, we can prove the logical postcondition that exactly those nodes with name n have been removed using these techniques.

The generated executable implementations of transformations have high

efficiency and are capable of processing large models of over 500,000 elements [63].

The Transformation Tool Contest has been of significant benefit to the development of UML-RSDS. In particular, the "Hello world" case identified the need for rules to distinguish pre-state and current-state versions of features.

UML-RSDS cannot be used if transformations are to be carried out within Eclipse/EMF. It is not suited to model-to-text transformations, or transformations, which require runtime metamodel or rule changes.

3.13. VIATRA2

The VIATRA2 (VIsual Automated model TRAnsformations [104]) framework is a component of the Eclipse Modeling Project[14] and its objective is to support the entire life-cycle of model transformations consisting of specification, design, execution, validation and maintenance. VIATRA2 uses the VPM (Visual and Precise Metamodeling) approach [102] that supports arbitrary metalevels in the model space. It combines graph transformation and abstract state machines (ASM) [8] into a single framework for capturing transformations within and between modeling languages [101]. Transformations are executed using an interpreter and both (1) *local-search-based* (LS) and (2) *incremental pattern matching* (INC) are available, providing additional opportunities to fine-tune the transformation either for faster execution (INC) or lower memory consumption (LS) [43].

3.13.1. Delete Node with Specific Name and its Incident Edges

Model transformations written in VIATRA2 consist of graph pattern definitions and both graph transformation rules and ASM rules. The `deleteN1NodeAndAllIncidentEdges` transformation below uses graph patterns for identifying the node with the specific name (`N1Node`) and the edges connected to a given node (`connectedEdge`). The example is complete and executable on any VPM model space that contains the proper metamodels. Upon execution, the `main` rule is called, which first attempts to find one node with the given name (`choose` semantics) and then the found node is used as an input parameter to find all incident edges in the model (`forall` semantics). Each incident edge is deleted from the model and finally, the node itself is removed as well.

[14] http://www.eclipse.org/viatra2/

```
import datatypes; // imported parts of the model-space are usable by local name
import nemf.packages;
import nemf.ecore.datatypes;

@incremental // uses incremental pattern-matcher
machine deleteN1NodeAndAllIncidentEdges{
  pattern N1Node(Node) = {// finds Node with name "n1"
    graph1.Node(Node); EString(Name); // type constraints
    graph1.Node.name(NameRel,Node,Name); // relation constraint
    check(value(Name) == "n1");} // attribute value constraint

  pattern connectedEdge(Node,Edge) = {// Edge is connected to Node
    graph1.Node(Node); graph1.Edge(Edge);
    graph1.Edge.src(SourceRelation,Edge,Node);
  } or { // Node can be source or target of Edge
    graph1.Node(Node); graph1.Edge(Edge);
    graph1.Edge.trg(TargetRelation,Edge,Node);}

  rule main() = seq{ // transformation entry point
    try choose N1 with find N1Node(N1) do seq{ // selects one match
      forall Edge with find connectedEdge(N1,Edge) do // iterates on all matches
        delete(Edge); // delete model element
      delete(N1);}}}
```

Listing 3: Delete node transformation

3.13.2. What is the Tool Suited for and why?

As a direct consequence of the metamodeling approach of VIATRA2, models taken from conceptually different domains (and/or technological spaces) can be easily integrated into the VPM model space. The flexibility of VPM is demonstrated by a large number of already existing model importers accepting the models of different BPM formalisms, UML models of various tools, XSD descriptions, and EMF models.

The VIATRA2 transformation framework has been applied to a wide variety of problems and its interpreted transformation language, incremental pattern matcher engine and transactional model manager provide a solid foundation for a wide range of applications. The change notifications offered by INC are used to drive a trigger engine to create live transformations [79] where rules are executed in response to specific changes and change driven transformations [4] that translate changes on an input model to changes on output models. The transformation engine also supports interactive execution of rules to drive simulations [80] or perform design-space exploration [30]. Finally, the development of transformations in VIATRA2 are supported by customizable model space visualization [31] and dynamic transformation program slicing [98].

VIATRA2 is *not* well suited for usage scenarios where the transformation rules change or evolve as part of the execution of the transformation itself. Although VIATRA2 is capable of importing and transforming EMF models, its performance and the conciseness of EMF transformation programs are lower than those of native EMF tools. This is mainly caused by the inefficient importer and the canonical model represention of the VPM approach that requires a large number of model elements to represent EMF models.

4. Discussion

In the previous section, the tools that competed at the TTC were introduced alongside their solutions to the "Delete Node with Specific Name and its Incident Edges" task of the Hello World case. The solutions are well suited to explain the approach of the tools and to illustrate their look and feel. The task fits well to the majority of the tools, but not all. Special-purpose tools without own transformation languages can only display their extensibility; this was the case for Edapt designed for model migration. Mapping-based tools are put at an disadvantage, as the task is easier to solve with an in-place change; this holds for GReTL and QVTR-XSLT, and partly for other tools offering both kinds, cf. Table E.6.

The remaining general-purpose rewrite-based tools are split into two classes by it, as the task reveals the ability of the languages of the tools to match and modify breadth-splitting structures. Some pattern-matching-based tools show solutions that appear overly complex for such a simple task. This is caused by a lack of declarative language constructs to process breadth-splitting structures, as explained in Section 7.2 and Table F.15 – the solutions specified in the languages of MDELab SDI, MOLA, and VIATRA2 had to revert to rule control to iterate over the incident edges.

This does not mean those tools are not suited to other tasks or to your task at hand, especially since the real strengths of the tools compared in this article do not shine on such a simple task. But if a lot of structures extending into breadth need to be processed in solving your task at hand, you should prefer the tools displaying a concise solution for the running example.

The Hello World case consists of several further tasks that are as simple as the one used in the running example. We want to encourage you to take a look at all of them. At least at the ones that fit best to your task-at-hand. All tools and their solutions to the Hello World case, as well as to the other

Subject	SHARE	Cases solved
Edapt	[35]	Hello World, Prog. Understanding, Model Mig.
EMFTVM	[106]	Hello World
Epsilon	[84]	Hello World
GReTL	[41]	Hello World, Prog. Understanding, Compiler Opt.
GrGen.NET	[16]	Hello World, Prog. Understanding, Compiler Opt.
GROOVE	[81]	Hello World, Compiler Opt.
Henshin	[57]	Hello World, Prog. Understanding
MDELab SDI	[110]	Hello World, Prog. Understanding
metatools	[65]	Hello World, Compiler Opt.
MOLA	[49]	Hello World, Prog. Understanding
QVTR-XSLT	[13]	Hello World, Compiler Opt.
UML-RSDS	[61]	Hello World, Model Mig.
VIATRA2	[29]	Hello World, Prog. Understanding

Table 1: SHARE images and solved cases of the tools

cases of the TTC, are available in the SHARE images listed in Table 1, ready to be reproduced and interpreted.

We close the discussion with the remark that some of the tasks are underspecified. This allowed on the one hand e.g. mapping based tools to solve the running task, just in a less optimal way compared to rewriting based tools; but on the other hand does it render the task of comparing the solutions more difficult. We must advise for some care when doing so: implementations may implement a task correctly regarding its pragmatics, but may do so semantically in different ways.

5. Transformation Tool Contest and SHARE

This section summarizes the organizational aspects of the Transformation Tool Contest (TTC), recapitulates the voting results, and explains the use of SHARE for the tool builders community or other communities with similar requirements. Feel free to skip it if you are just searching for a tool.

The Transformation Tool Contest[15], the hosting event of the Hello World case and the organizational umbrella for this survey, is a scientific workshop with the aim of comparing the expressiveness, the usability and the performance of graph and model transformation tools alongside a number of

[15] http://planet-mde.org/ttc2011/

selected case studies. Participants of this workshop want to learn about the pros and cons of each transformation tool considering different applications, and especially the pros and cons of *their* tool. A deeper understanding of the relative merits of different tool features helps to further improve graph and model transformation tools, advancing the state of the art. The workshop comprises several so-called offline cases, which have to be solved by the participants before the workshop, but also a live case not known in advance that has to be solved by the participants during the workshop. The live case, which is not covered in this paper, allows to evaluate how well-suited the transformation tools are for rapid prototyping.

The review process of the TTC was based on SHARE (Sharing Hosted Autonomous Research Environments), which is described by its creators [25] as: "SHARE is a web portal that enables academics to create, share, and access remote virtual machines that can be cited from research papers. By deploying in SHARE a copy of the required operating system as well as all the relevant software and data, authors can make a conventional paper fully reproducible and interactive."

5.1. Offline Solutions Workflow

We describe the workflow regarding the offline solutions in more detail, because they allow you to understand the role of SHARE.

Solving the cases and submitting the solutions: The offline cases were published several months before the workshop. Potential participants solved them with their favorite tools, and submitted their solutions, with a document describing the solution, but especially with an installation in a remote virtual machine offered by SHARE.

Solution review: This SHARE image together with the accompanying paper was then the basis for the peer review. Some time ahead of the workshop, two reviewers also called opponents were chosen by the organizers from the set of all participants to investigate a given solution in detail, and to give a first vote on the solution needed for accepting it to the workshop. The other participants had equal rights to access the SHARE images to inform themselves about the competing solutions. To emphasize it: the very task of the opponents was to find out whether the solutions available in the share images were adequately described by the accompanying papers, whether ugly things were swept under

the carpet, or even false claims were made – which can easily happen with traditional, paper-only-based reviews.

Solution presentation: During the workshop, the solution submitters presented their work in front of all other participants. They were able to focus on the strong points of the solutions. But after the presentation a discussion was scheduled, in which the general audience could ask questions, and especially the opponents were on duty to report about the weak points of the solutions, thus balancing the presentation of the solution submitter.

Solution voting: The presentation and discussion were then followed by the voting: All the contest participants were asked to fill out an evaluation sheet for each solution (except their own). The criteria to be used in the evaluation sheet were defined by the case submitters, the participants had to score each solution with 1–5 points regarding each criterion. The solutions were ranked and awarded prices along the votes of the participants.

5.2. Cases

There were 4 offline cases to be solved, "Program Understanding" [40], "Compiler Optimization" [10], "Model Migration" [34], and "Hello World!" [70].

The complex cases Program Understanding, Compiler Optimization, and Model Migration complement the Hello World case with non-primitive tasks. They allow to assess the ability of the tools to cope with difficult tasks and large workloads, evaluating expressiveness, performance, and scalability. A short introduction into the results of these tasks was already given with the *Related Work*, for the detailed results we must hint at the proceedings of the TTC [99].

5.3. Votes

As described above, the solutions of all Hello World tasks were (1) investigated in detail by builders of competing tools (called opponents) and (2) voted by the participants at the transformation tool contest.

The votes for the Hello World case, which were cast alongside the evaluation criteria understandability, conciseness, and correctness, are listed in Appendix D and discussed in detail. Here we give a brief summary. Please note that the complete set of solutions was voted, not only the solution of

the running example we used to introduce the tools. This is why tools with a complicated solution for that task are ranked above tools that showed a concise solution.

The votes on *understandability* were influenced by three points: (i) the distinction into graphical versus textual languages, with a general bonus for graphical tools, (ii) the concepts the tools are built upon, constructs from formal logics received a malus, and (iii) whether the tool offers a syntax similar to well-known programming languages, which was preferred.

Regarding the point *conciseness*, the availability of (i) lightweight means for simple CRUD tasks played a role, as offered by imperative solutions, but even more so (ii) the general expressiveness of the tools, as expressed by the availability of the features referenced in the feature matrices in 7.2; they had not to be used into great depth, but their general availability already lead to more compact solutions compared to competing tools and better voting results.

The three top-scoring tools were Epsilon, GROOVE and GrGen.NET. The Epsilon solution was able to employ a language fitting to the task at hand for nearly each task of the Hello World case, yielding high scores for conciseness and understandability. The only caveat was pattern matching, where an imperative solution with nested loops had to be applied. GROOVE was able to solve each task with one rule, which resulted in the highest conciseness vote of all competing solutions. But the quantified nodes leading to the high conciseness gave it only a midfield result regarding understandability (exemplifying the negative effect of formal logic constructs). GrGen.NET was as balanced as Epsilon regarding conciseness and understandability, without a major notable single point of weakness, but also without a major noteworthy single point of strength.

The votes cast assess the performance of the tools in solving the set of tasks Hello World is built of regarding understandability and conciseness; this gives a rough indication on the usability of the tools for many different tasks. But they need to be taken with a grain of salt – they depend on many subjective influences as discussed further in Appendix D. You may come to a different conclusion when you inspect the SHARE images listed in Table 1; they allow *you* to reproduce and interpret the results on your own.

The votes were cast along a third point, *correctness*. The Hello World task descriptions, despite being simple, contain several ugly corner cases and ambiguities, that showed when they were to be implemented. This lead to an astonishing amount of less-than-full-points results, considering how

simple the tasks are. Those missing votes had not much influence on the overall outcome; they hint at a possible improvement of the Hello World case, though.

5.4. Review Process Support

The SHARE images employed during the contest served two functions.

Primary Function. This review of the solution and tool together with the paper allows for much more honest results than paper only solutions. A paper can be tweaked easily to show a compelling solution by emphasizing the strong points of the solution and leaving out the dark corners entirely. An executable environment cannot be tweaked like this. Especially not with opponents, who explicitly want to find negative aspects so the competing tool can be voted down, and their own tool look correspondingly better.

SHARE was used massively for this purpose, as can be seen by the comments the opponents gave[16], e.g.: "Unfortunately, only the first task (...) is reproducible in the SHARE demo, the other solutions fail for some reason (...). Please fix this."

Besides the SHARE hosting computer was used as the base for performance comparisons in the complex cases; this made the original numbers comparable, which were typically measured at wildly different machines.

Secondary Function. The aim of the Transformation Tool Contest is to compare the participating tools, and to allow the tool providers to learn from each other. The SHARE images are helpful in this regard, too, because they allow to inspect competing tools with a minimum of effort.

A case submitter that first only sent an archive file was asked by another participant for a SHARE demo: "(...) a SHARE demo really should be created for this solution. I just did not figure out how to operate the Java program in your submitted zip file." With SHARE, the reviewers can concentrate on reviewing the tool already set up, and prospective users on assessing the tool – they do not need to invest their time into getting the tool running (this effort is only needed for the tools the user finally chooses).

[16] http://planet-research20.org/ttc2011/index.php?option=com_community&view=groups&task=viewgroup&groupid=13&Itemid=150

6. The World of Transformation Tools

In the following, we introduce you into the world of transformation tools. To this end, we ask motivational questions, which we answer with explanations, and with feature matrices highlighting the positions of all the tools in the field. The explanations introduce core notions of the domain, while the feature matrices give a direct overview of the tool landscape.

The introduction is organized along five areas:

Suitability What are the goals of the tool, what is it suited for?

Data Which data is to be transformed?

Computations What kinds of computations are available, how are they organized?

Languages and user interface How does the interface of the tool to the user look like?

Environment and execution How does the interface of the tool to the environment look like, how is it executed?

The aim of this section is to give you an overview, a coarse grained map of the field.

6.1. Suitability

"Is the tool suited to my task?" is the first question that comes to mind when you have to decide whether to use a tool or which tool to use. In Table 2, we summarize the information from the "What is the tool suited for and why?" sections of the tool introductions, where the tool providers described the design goals and the strong points of their tools. Everything is available on one page, so the tools can be easily compared against each other.

Nearly all tools were built with the goal of offering general-purpose transformations, which is the reason why we for one discussed the results of solving all the prototypical little tasks of Hello World in Section 5.3, and for the other the reason why we compare in the following and esp. in Section 7 with a taxonomy derived from aspects and subtasks of complete tasks. The feature matrices used to this end allow to deduce tool performance for your task-at-hand, and accumulated give a hint on tool performance for many different tasks.

Suitability and strong points of the tool
Edapt: *model migration* in response to metamodel adaptation. High automation by reuse of recurring migration specifications. In-place transformation, seamless metamodel editor integration.
EMFTVM: *general-purpose model transformation.* ATL is a mature language for mapping input models to output models. The EMFTVM runtime introduces composition and rewriting.
Epsilon: *general-purpose model management and transformation.* For each task the right language; with editing and debugging support. Abstracts from the underlying modeling technology.
GReTL: *general-purpose model transformation.* Expressive graph query language (regular path support and set semantics). Highly extensible transformations.
GrGen.NET: *general-purpose graph rewriting.* Pattern matching of high performance and expressiveness; highly programmable. Excellent debugging and documentation. Excels at compilers, computer linguistics.
GROOVE: *state space exploration, general-purpose graph rewriting.* Rapid prototyping, visual debugging, model checking. Expressive language (nested rules, transactions, control); isomorphism reduction.
Henshin: *graph transformations for EMF models* with explicit control flow. Expressive language (nested rules, support for higher-order transformations) JavaScript support, light-weight model & API, state space analysis
MDELab SDI: *graph transformations for EMF models* with explicit control flow. Expressive language, mature graphical editor, support for debugging at model level. High flexibility, easy integration with other EMF/Java applications.
metatools: *general-purpose model transformations.* Seamless integration of hand-written and generated sources, of imperative and declarative style. Full access to host language, libraries and legacy code.
MOLA: *general-purpose model transformations* with explicit control flow. Expressive language, graphical editor with graphical code completion and refactorings, built-in metamodel editor, EMF support.
QVTR-XSLT: *general-purpose model transformations.* Supporting the graphical notation of QVT Relations with a graphical editor to define transformations, and generate executable XSLT programs for them.
UML-RSDS: *general-purpose model transformation* with verification support. Declarative transformation specification using only UML/OCL. Efficient compiled transformation implementations.
VIATRA2: *general-purpose multi-domain model transformations.* Model space with arbitrary metalevels, excellent programming API. Incremental pattern matching.

Table 2: Suitability and strong points of the tool

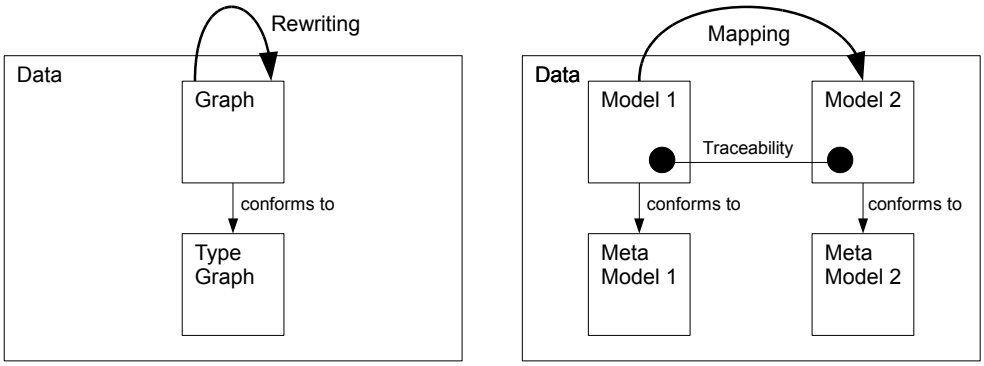

Figure 14: The data refined

Many tool providers claim that their tool is *expressive* (thus allowing to achieve *concise* solutions), we will explain what it means for a transformation tool to be expressive in greater detail in Section 7.2, so we can understand why (and judge whether) these claims hold.

6.2. Data

"Can I adequately model my domain?" is the most important question concerning the data. A first answer is given by Table E.6. There, the domain of the tool and the kind of the tool are distinguished. In Figure 14 are the two prototypical ones illustrated, to the left the graph rewrite tools with AGG [93] as a typical tool, and to the right the model mapping tools with ATL [47] as an example tool.

Domain. The tools compared at the TTC 2011 originate from two historically distinct worlds, the realm of graphs, where the data is called a graph and conforms to a type graph. and the realm of models, where the data is called a model and conforms to a metamodel. To a large degree the difference between graph and model transformation tools is a question of self-definition, to which community and culture the tool authors have stronger links, or what they define as their goals. Under the abstract setting put forward in the beginning "the most adequate representation of the data at hand is a mesh of objects, and we need to change its structure or map it to another mesh of

objects" they are unified. In the prelude to Table E.6[17] we take a closer look at the blurred boundary between the two worlds and the locations of the tools inside them.

Kind. The second discrimination point is the kind of the tool, with a distinction into mapping versus rewriting[18]. A mapping tool operates on several models. It typically maps from a constant source model to a target model (sometimes are further source or target models included). Mapping tools offer explicit syntactical means to distinguish between several models/instances. The models might conform to the same metamodel; this way rewriting tasks can be processed by mapping tools. A rewriting tool in contrast operates on one model. The model might be a union of several metamodels; this way mapping tasks can be processed by rewriting tools.

Rewriting tools are more adequate for tasks where only a small part of the model needs to be changed and the rest should stay untouched (i.e. local changes), as only the changes need to be specified and executed. In mapping tools you must specify and execute the copying of the parts, which should stay untouched. For tasks where one representation is to be mapped to another one (tasks showing a high rate of turnover), mapping tools are better suited: they do not need to take care of a model partly built from elements from the source and partly from the target model, and they offer implicit traceability handling (see below). Complicated tasks, which need to be decomposed into a series of smaller tasks tend to favor the rewriting approach: a series of rewriting steps (each responsible for a small part of the overall work) is easier to specify and more efficiently executed compared to a series of full mappings from one representation into the other. But the expressiveness of the computational constructs we visit later on has an even stronger impact on those tasks.

Traceability. Traceability information is stored when a source element is mapped in a transformation to a target element; it allows to fetch the source from the target element or the target element from the source element later on. Traceability support may be a built-in service of the transformation

[17] You find the table in the appendix, as nearly all tables, for they would hamper readability when being included inline.

[18] Sometimes transformation is used in the literature as a synonym for mapping; in this article we use it in the wider sense, comprising both rewriting and mapping.

engine, as it is typically the case for mapping tools, or may require manually coded assignments to variables of map type. Keeping the identity of the transformed entity may emulate this behavior in a rewrite-based tool supporting retyping.

Typically only mapping tools offer the user-convenient and declarative implicit traceability. For a rewriting-based tool, it is possible to model traceability links between nodes (only nodes) with an edge in between the source and the target. This "pollutes" the metamodels but allows for a trivial visualization as a surplus (in case the tool supports visualization). But this does not extend to storing and visualizing traceability information for attributes, as it is typically offered by tools that maintain traceability links implicitly.

6.3. Computations

"Can I adequately specify my computations?" is the most important question concerning the transformation of the data (and for the field of transformation tools as such). A first answer is given in Table 3, after an introduction into the most important approaches and concepts up-front, in order to understand the differences and their consequences.

Programs. The most basic approach is the program-based one: the computations are implemented in an imperative, object-oriented programming language, either a general-purpose programming language or a domain-specific programming language as offered by some of the tools. The program-based approach builds on *simple queries* and *simple updates* against the API of the model, which are glued together by *state variables* and *control structures*. With simple queries we mean read operations, which return i) all elements of a certain node type, or ii) the attribute values of a node, or iii) an iterator over all incident edges/adjacent nodes of a given node. With simple updates we mean create element operations, delete element operations, and attribute value assignments. They are combined by state variables, either of basic type for storing single elements, or of container type, for storing collections of elements; and by the commonly known control structures, i.e., sequences, conditions, loops, and subprogram calls. A large transformation is built from multiple subprograms. The image to the left in Figure 15 illustrates the programmed approach.

Alternatively to a model or graph API offering operations to access all elements of a certain type, there could be variables containing root nodes as entry points; typically this leads to computations, which are organized into

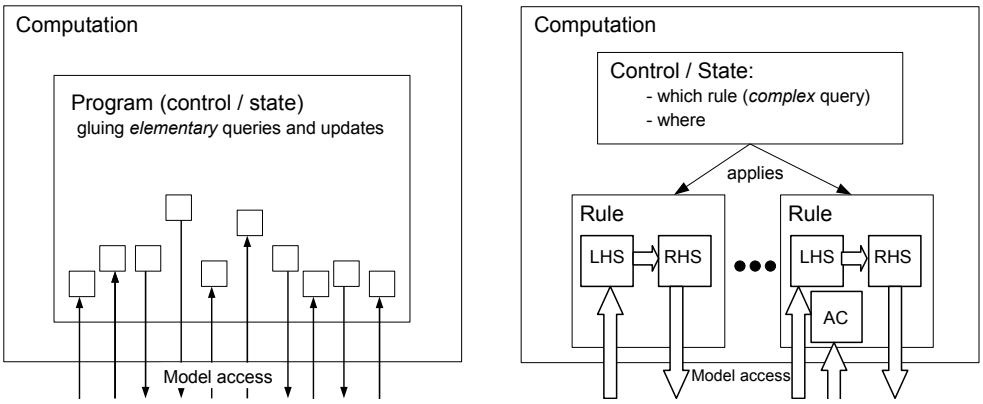

Figure 15: The computations refined

passes navigating the object structure following the contained or referenced elements. This defines the traditional, non-model way of processing object structures.

Navigational Expressions. The first step away from general-purpose programming languages and towards domain specific programming is carried out with the inclusion of OCL [109] expressions. The Object Constraint Language combines attribute comparisons with logical operations and especially with navigational expressions, which allow to formulate constraints on the *connection structure*. Being expressions, they are executed without side effects on the model and return a value: for references with one target element, the return value is the single element, but for references with multiple targets, a collection of elements is returned (the same holds for retrieving all elements of a certain type).

Rules and Query-Update Units. A further step away from traditional programming taken by most of the tools is the creation of declarative transformation units as central element. They consist of a *complex query* – also known as left hand side (LHS) or precondition – and a *dependent update*, i.e., an update depending on the query result – also known as right hand side (RHS) or postcondition. The main service offered by these transformation tools is to relieve the programmer from writing the glue code with its control constructs and state variables by hand, which is needed in order to realize those complex queries and dependent updates. These tools offer an interface at a higher level of abstraction. The model state is changed in big steps,

one half-step reading and one half-step writing, compared to a series of small interwoven steps in the program-based approach.

The declarative transformation unit is distinguished into *rules* and into *query-update*-units. Within rules, the update is *strongly bound* to the query. The postcondition is specified directly referencing the elements from *the* precondition (as single element variables). The query-update-units, which are executed similarly to database queries, are *loosely coupled* in contrast: the query fills some explicitly declared intermediate container variables, accumulating all queried elements, which are then read by the update.

Separation into Layers. The tools building on declarative transformation units separate the computations into a query plus update layer (the transformation unit) and a control plus storage layer. Only the transformation layer queries and updates the model directly. The control layer on top of it is responsible for orchestrating the rules, it defines *which* rule is to be executed next (scheduling) and *where* it is to be executed next (location).

Implicit control is exercised by a control engine deciding which rules to apply where. You can communicate hints or constraints on rule scheduling to the execution engine, e.g., with rule priorities. We speak of *explicit control* if you define with a control program which rules are to be applied where. This may occur in one of several ways: the tool might offer a dedicated rule control language, or a tool specific programming language that is used for control, too, or the control layer might be programmed in a general-purpose programming language. If rules are triggered by the engine when the model changes we speak of *event-driven* control. The image to the right in Figure 15 illustrates the rule plus control approach.

Application Conditions. A further step of separation is carried out in the declarative transformation unit based tools with application conditions. They specify constraints, which must hold for applying the rule. The elements captured by them are *not* available for rewriting or mapping, though. They are the main means of constraining the location of rule application in rule languages that are mapping one element (and not a pattern) to multiple elements. In application conditions you may especially ask for elements to *not exist*. In query-and-update-based languages both parts are combined, and explicit projection of the values of interest is used instead.

Patterns. Tools following the rule-based approach often employ *patterns*. Patterns specify a subgraph that is to be matched in the host graph, a small

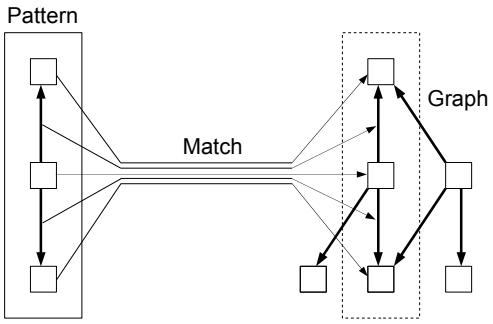

Figure 16: Pattern matching illustrated

submodel that is to be sought after in the model, as illustrated in Figure 16. The hallmark of the pattern-based approach is the *direct representation* of the pattern elements within the matched spot inside the host graph. The semantics of patterns are based on existence, and a search for a match, binding each pattern element to a graph element. If a match is found, the rule – which consists of two patterns – will be applied: elements only available in the left hand pattern will be deleted, elements only available in the right hand pattern will be created, and common elements will be retained.

Direct Reuse. Instead of relying on more complex means to combine computations and esp. complex queries, you could aim for direct reuse, with a library of predefined computations you choose from and parameterize. This approach obviously only works for constrained domains or task where some predefined routines and a few simple parameters are sufficient to define the required computations. If it can be applied, it offers the largest amount of reuse.

Helper Code. Instead of stepping up the abstraction level of the computations with domain specific languages, you could stay at the level of user programmed code (and the program-based approach), just aided by a code generator, which emits some helper code. The main service offered here is the unfolding of a concise specification of the data and some processing aspects to boilerplate code, which would have been cumbersome to write manually; everything else is still directly programmed in a general-purpose programming language against the generated code.

Approach
Edapt: Library with predefined operations for adapting a model in reaction to metamodel changes, custom adaptations are to be programmed using an API in Java.
EMFTVM: Rules mapping from an element and OCL expressions to elements and a connecting program with implicit control, or rules rewriting a pattern to a pattern with explicit control.
Epsilon: Task specific languages built on a base programming language (incl. OCL-like expressions). For transformations: rules mapping from an element and OCL expressions to elements and a connecting program, with implicit control.
GReTL: Library to be extended, offering a query language yielding data containers; the updates are to be written in Java, some basic ones are predefined.
GrGen.NET: Rules rewriting patterns to patterns with explicit control.
GROOVE: Rules rewriting patterns to patterns with explicit or implicit control, and strategies for state space enumeration.
Henshin: Rules rewriting patterns to patterns with explicit control.
MDELab SDI: Rules rewriting patterns to patterns with explicit control.
metatools: Code generator for Java classes and access helpers, visitors, rewriters. The computations as such are to be programmed as visitors in Java and are carried out during visitor runs.
MOLA: Rules rewriting patterns to patterns with explicit control.
QVTR-XSLT: Rules from patterns to patterns with implicit control.
UML-RSDS: Rules which ensure a postcondition OCL expressions is satisfied when a precondition OCL expression was matched, with implicit or explicit control.
VIATRA2: Rules rewriting patterns to patterns with explicit or event-driven control, or direct usage of the control language as programming language.

Table 3: Approach for specifying computations

Discussion of Suitability. Patterns and pattern-based rules define a simple and intuitive approach for expressing computations over graph structured data, with the potential benefit of direct visualization. But this comes at a cost: patterns with their fixed structure are less expressive than query or programming languages. For tasks that only require to match and rewrite fixed shapes, the solutions based on them are as concise and declarative as can be. But they fit badly to tasks that require to process all neighboring elements of a node, or that require to search for a path from a source node to a target node. For these kinds of tasks, a high amount of control over tiny patterns is needed, virtually eliminating the advantages of the pattern-based approach, falling back to a programmed solution. Even worse: falling back to a programmed solution with the additional weight of being separated into multiple layers. To counter the aforementioned deficiencies, the pattern-based languages were extended with further constructs; we will inspect those extensions later on in Section 7.2. Especially as the support for processing breadth-splitting structures and depth-extending structures is of high importance for all kinds of tools.

The program-based approach in contrast does not suffer from being incapable of expressing anything, it is highly flexible and adaptable — it only does not offer many advantages over traditional imperative or object-oriented programming languages. The program-based approach works well and yields concise solutions if the task at hand requires only small queries. This is typically the case for mostly 1:1 mapping tasks of one model to a structurally similar model, where only a bit of local context needs to be queried. But for these kinds of tasks the mapping tools built on OCL and implicit control offer a compelling alternative. Being expressive enough for those tasks, they lead to a more declarative, functional-style specification.

If the task at hand requires to specify queries comprising more than a handful of nodes though, the query-update or pattern-based tools become a must-have in order to achieve a concise solution, as only they can evade the large amount of glue code needed in implementing the complex queries. For simple 1:1 mapping task on the other hand they may be a bit heavyweight (especially due to their typical separation into layers).

In order to decide in between the pattern- and the query-based tools, which both offer declarative queries, you should ask "Do I need large patterns?" but especially "Do you I need large, non-uniform updates per matched spot?", or rephrased "Are the updates highly context dependent?". If so, the pattern-based tools are better suited, as they allow to specify more directly

and in finer detail what should happen in what situation with what elements, whereas the query languages must operate through the bottleneck of the query result data structure. The container variables between the queries and updates allow for an easy accumulation of extracted data on the other hand, surpassing the pattern based tools when the data from the matched spots needs to be integrated in some way.

An argument similar to the finding that simple queries are well-suited for 1:1 mapping tasks and complex queries are needed for mining of distributed data or matching large patterns holds for implicit control and explicit control: Implicit control typically allows for more concise specifications (no control program needed) for tasks which are well-suited for the control engine's strategy. Explicit control is more robust regarding problems for which the implicit control engine was not designed; it then yields faster execution times (the constraints of the task at hand can be exploited), or even allows to handle a task at all.

6.3.1. Examples

To highlight the differences in between the approaches, to illustrate the gain of the declarative constructs, and to ease understanding, we implement a small example query in each of them, that shows how *structural* information is collected.

Patterns. We query the model for all nodes `a` of type `A`, which are connected via an edge `v` of type `V` to an opposite node `b` of type `B`, but are not at the same time connected via an edge `w` of type `W` to a node `c` of type `C`. In pattern-based languages this query is expressed with the following pattern, often in graphical instead of the chosen textual notation:

```
a:A -v:V-> b:B
negative {
    a -w:W-> c:C
}
```

We note that it is not specified how this structure is to be matched. You can easily add a further node being adjacent to some of the already available nodes.

Query Languages. In query-update-based languages the example query is expressed using a term similar to the following one:

```
from v:V
with a := startVertex(v), b := endVertex(v)
     and hasType{A}(a) and hasType{B}(b)
     and not (v -w:W-> c:C)
report a, v, b
```

The from clause declares elements of interest that are to be found in the graph. The with part specifies the conditions that must be fulfilled for the elements of interest, maybe introducing further helper elements. The elements satisfying the conditions are reported back in the way specified by the report clause.

OCL. In OCL-based languages the example query is noted down with an expression similar to the following one:

```
A.allInstances->select(a |
  a.V->exists(b | b.oclIsTypeOf(B)) and not
  a.W->exists(c | c.oclIsTypeOf(C)))
```

All instances of the type of the node are selected, and for each of them it is checked whether one of the `V` or `W` references leads to a node of type `B` or `C`, the results are combined with boolean operators. Here the order in which the elements are to be visited is partially fixed, the search is carried out by nested expressions.

Programs and Control. In program-based languages the example query is implemented with a program similar to the following one:

```
found = foundNAC = false
for a in model.getAll(A)
  for b in a.V
    if b instanceOf B
      found = true
      break
  for c in a.W
    if c instanceOf C
      foundNAC = true
      break
  if found and not foundNAC
    result.add(a)
```

The order in which the elements are to be visited is fixed, the search is carried out by loop statements utilizing variable assignments. A further adjacent node requires another nested loop, inserted at the right nesting level.

6.4. Languages and User Interface

"Does the user interface of the tool fit to my needs or preferences?" is the primary question concerning tool-user interaction, this includes especially the languages offered by the tool. We give a first answer in Table E.7.

Languages. Most tool-supplied languages employed in solving the Hello World tasks are separated vertically into several sublanguages, for data definition (see Section 6.2), and for specifying the computations (see Section 6.3) with the means available according to the approach, i.e. a programming language, or a query language, or a rule language coming commonly together with a control language.

Some tools are horizontally split into several special-purpose languages, which are offering an own domain specific language for each of several goals. They offer additional functionality outside of the focused topic of this article, the transformation of structures; the same holds for special-purpose tools that do not allow for general transformation programming.

External Languages. The question "What languages does the tool offer?" we raised above needs to be complemented by the question "What languages does the tool require?" for the parts programmed in an external programming language. Here we have to distinguish whether the computations *can* or *must* be programmed in an external language. In some tools the computations are meant to be programmed in a general-purpose programming language *by-design*. In others, only the computations that are exceeding the functionality of the supplied library are to be programmed this way. Most tools offer own languages, their functionality is then typically made available via an API. For those tools we have a further look in Section 7.3 on the other direction of usage (for using entities from the outside inside the tool languages).

Employing a general-purpose programming language allows you to reuse the knowledge and skills you have already acquired in programming in it – and the tools that are available for it. Everything happening is fully transparent at the level of the programming language and its debugger. But only at this level – you miss the typical advantages of conciseness and declarativeness displayed by the domain specific languages of the transformation tools; and

the potentially available visual style of programming and debugging offered by the pattern-based tools.

Tools that are offering an API allow you to use the specified transformations from the outside. This is crucial if you want to benefit from the advantages of the transformation languages in case other parts of the task at hand require an integration with network communication or a 3D rendering engine, which are not supplied by the tools.

A transformation can be reused as such; this defines the most easy and beneficial form of reuse. Alternatively, you could start a step down in the layering of Figure 15 by using the declarative transformation units from a general-purpose programming language[19]. This is helpful if restrictions of the control languages would render the task at hand overly complicated.

You could start a step even further down in the layering of Figure 15 by writing the computations to a large degree in a general-purpose language, reusing only a data API plus some helping code. For some tasks this may be a useful approach; but even more so would be a hybrid solution, where you employ declarative transformation units for the subtasks where they fit well, and manually code the solutions to the subtasks where they do not.

Form of Specification Language. In Table E.8, we list the form of the specification languages. We distinguish *graphical* languages like UML class or activity diagrams from *textual* languages. Graphical languages are normally more intuitive, often better readable, and can be learned quicker. This holds especially for pattern-based languages, which offer the most intuitive encoding of structural changes [20]. Textual languages are typically more concise and expressive, and can be edited more easily. They offer a better integration into existing source code management systems and their textual difference engines, but especially they allow to generate specifications by some text-emitting scripting code. For this reason even pattern-based tools – for which graphical languages are the more natural encoding – may still offer textual languages. To a certain degree the choice is a matter of personal taste, you

[19] Such an architecture would be very similar to the layering applied in a lot of applications written in a general-purpose programming language, which delegate the subtask of storing data persistently and retrieving it again via an API to a database engine offering declarative SQL queries and updates. Here the programming language is notably used to glue the single SQL statements into larger activities.

[20] this depends on the number of elements though, the advantage of immediate visual understandability of structures fades as they grow

should have found out about your own preferences by inspecting the listings in Section 3, and can find out further by playing around with the SHARE images.

6.5. Environment and Execution

"In which environment can I use the tool?" and "How are the tool specifications executed?" are the primary questions we have a look at in this section.

Execution Host. "How are the transformations executed which I specified?" is answered in Table E.9. The tool may contain an external executable, which allows to run the transformation, it may contain a plugin for an IDE that is able to execute and debug the code, or an API may be offered so that the transformations can be executed from an user application

If the transformation should be integrated as the algorithmic core into a user application, an API is required. If the transformation service you need consists only of the mapping of one file to another file, an external executable is the most advantageous execution host, as neither further code files are needed, nor have the startup costs of an IDE to be payed. An IDE integration offers the most convenient development, especially if a computation consists of transformation code and external code.

Operating System. The most binding decision regarding the environment are the operating systems supported. The tools supporting all major desktop operating systems (Windows, Linux, Mac OS X) are built on the Java virtual machine or the Common Language Runtime; they can be used on all operating systems for which those platforms are available or will be available. Native programs are bound to their underlying OS unless they are ported with high effort to another one, but they typically offer performance advantages over VM-based tools, better integration into their host system, and do not require that a VM is available; this point is investigated in-depth in Section 7.4.

Tool Execution. "Is the tool able to handle my workload?" is a question whose answer depends on the workload, but a broad hint at the *performance* characteristics is possible by having a look at the execution model, the matching engine, and the memory consumption.

Execution Model. In Table E.10, we compare the execution model of the specification languages, distinguishing compilation or code generation from interpretation. Compilation results in general in higher execution speed[21], at the price of having the parts implemented this way being fixed at compile time. Interpretation in contrast allows for faster development turn-around times and gives the flexibility of runtime adaptation.

Engine. Querying for graph information, esp. matching patterns (also known as subgraph isomorphy solving) is the most expensive operation in transformation tools. The different approaches search-based, incremental, and user programmed and their performance characteristics are introduced in Table E.10.

Memory. Besides the time needed to execute the transformation, the memory consumed by the metamodel is of importance. You can estimate the memory consumption of the task at hand from the values given in Table E.10.

7. The Tools in Detail

In the following, we work again based on the setup we already employed in the previous section, asking motivational questions, which are then answered with feature matrices and a discussion of the consequences of the features, explaining why and when the features are of importance. Here we refine the initial answers that were geared towards giving an overview of the field with a step into greater detail. The focus is on answering questions that allow you to choose the tools that are best-suited to your task-at-hand; on questions that are normally quickly raised, and on questions you would raise if their importance was known.

The tool comparison is organized along the areas that were already employed in Section 6, namely:

Data: Which data is to be transformed?

Computations: What kinds of computations are available, how are they organized?

[21] While compilation yields faster results for frequently executed operations, it might fall back behind interpretation if queries are only to be executed once.

Languages and user interface: How does the interface of the tool to the user look like?

Environment and execution: How does the interface of the tool to the environment look like, how is it executed?

Suitability as seen by the providers was already deepened in Section 3 and summarized in Section 6.1, and is thus not taken care of any more in its direct high-level form – but the features compared with the feature matrices are chosen to allow you to assess suitability on your own, by breaking up your task-at-hand to the subtasks and aspects listed there. In addition to the areas named, the support for **Validation and Verification** is investigated in more detail.

7.1. The Data Refined

"Can I adequately model my domain?" received only a very coarse grain answer in Section 6.2, but is a question of high importance, as the model comprises the foundation on which all of the computations and all of the other tool features are built. So we investigate it more deeply here, regarding the expressiveness of the metamodel and the import/export capabilities of the tool.

7.1.1. Input and Output

"Does the tool support the file formats I need?" is the question answered in Table F.11, which compares the tools regarding their import and export capabilities. A tool built on a *standard* modeling technology and API supports the serialization format of its technology out of the box. Sharing a common modeling technology allows for easy *integration* with other tools. A tool built on its own modeling technology may still be able to import and export the serialization format of another modeling technology by mapping those concepts to its own concepts, allowing for *interoperability* in between the tools.

7.1.2. Metamodel Expressiveness

For the following parts we are back again at the original question "Can I adequately model my domain?", now having a look in greater detail at what the tools are able to express, i.e., what the metamodels (or modeling technologies) support. We respond to the query "Do the nodes, edges, and attributes allow me to directly encode my problem?", with Table F.12 and Table F.13.

Nodes and Edges. The elements defining the structure may be untyped or equipped with types. The types may be simply disjoint, or organized in a single-inheritance hierarchy, or even a multiple inheritance hierarchy. Additionally, the elements may or may not bear attributes.

Typing is an essential part of modeling; it prevents nonsensical models from getting constructed successfully, and nonsensical transformations from getting executed successfully. Single inheritance is an improvement over disjoint typing. It allows you to factor out common parts according to a compatibility relationship, and to process them with a single piece of code, thus enabling more concise transformation formulations. Multiple inheritance further improves on this by allowing you to organize the elements in as many compatibility relationships or hierarchies as you like.

In contrast to the nodes that are equipped with very few degrees of freedom, there are many differences in how edges (resp. references) are exactly realized, e.g. they may have an identity, they may be undirected or ordered, or may be distinguished into references and containment edges. Choose the ones that fit best to the needs of your task at hand.

Attributes. The graph elements may be attributed freely, or depending on their type. The liberal kind is more flexible, but rules out static type checking. The attributes of the graph elements may be typed with one of the basic types known from traditional programming. You can check in Table F.13 whether the ones you need to model your domain are supported.

We complete the model part of the comparison by stating that all the computations compared next operate on an in-memory-representation, in contrast to databases (e.g., graph databases).

7.2. The Computations Refined

"Can I adequately specify my computations?" was partly answered in Section 6.3 by an explanation of the different approaches and a discussion of their consequences. Here we go into detail: first, we compare the expressiveness of the general-purpose tools, then we inspect some selected software engineering aspects.

7.2.1. Introduction to Expressiveness

"Is the tool expressive enough to allow for a concise solution for my task at hand?" is the main question to be answered in the following. To this end we will have a look at four subtasks:

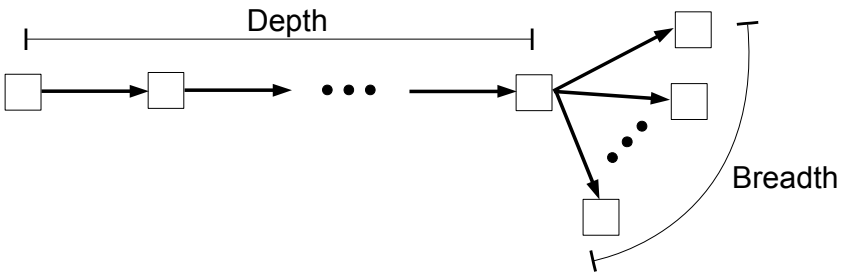

Figure 17: Depth extension and breadth splitting structures illustrated

Data collection How do I query for data, how do I check for context constraints?

Result storing Can I store results from queries for later processing?

Depth How do I capture structures that are extending into depth?

Breadth How do I capture structures that are splitting into breadth?

Regarding all of those subtasks, there are more general and less general language constructs available. The less general, i.e., less expressive constructs are before all *easy to learn and understand.* [22] If they are sufficient, they lead to a concise specification; typically even more concise than a solution employing more expressive, i.e., general constructs. But the "if sufficient" is a decisive constraint here. The more general constructs allow to *directly express solutions* to subtasks that require a combination of multiple constructs in the simpler language, e.g. using control to apply a rule multiple times to capture all neighboring nodes. Then, they lead to far more *concise* solutions – and the following rule of thumb: the more complex the languages, the simpler the solution. You must understand what level of expressiveness is required by your task at hand in order to choose a tool.

Depth and Breadth Structure Handling. Figure 17 illustrates two common and important subtasks of transformation tasks. Often chains of nodes linked

[22] Furthermore, they are typically easier to visualize, and to implement for the tool provider, which especially means they are more likely implemented correctly.

by edges need to be inspected, to gain access to values at the end of the chain (this is typically used to query for the transitive closure of a relation); maybe the elements on the chain should be even changed. If this is indeed the case for your task at hand, you should inspect the support of the tools for capturing structures extending into depth. The other repeatedly appearing subtask involves inspecting all neighboring nodes of an anchor node; maybe the neighboring nodes even need to be changed. If this is the case for your task at hand, you should inspect the support of the tools for capturing structure splitting into breadth.

The problem is to capture structures with a *statically not* known number n of depth-extending or breadth-splitting steps, with *less than* n notational elements. Structures that extend into depth by a *statically* known number of k steps can be described directly with patterns of k elements; or by OCL expressions with one nesting level per step, or by programs with one level of loop nesting per step. The same holds for the pattern based approach regarding structures that split into breadth with k branches.

The following Section 7.2.2 and Section 7.2.3 give a condensed survey over the programming resp. specification constructs offered by the tools, including a comparison of their expressive power. If you are a novice to the world of transformation tools just searching for a tool to use you might have difficulties to follow the comparison. What you should take away from this important topic is given in the directly following paragraph.

Expressiveness Results in a Nutshell. When your task comprises matching depth extending structures, look out for regular path expressions or recursive patterns in the left column of Table F.15, they allow to concisely and declaratively capture those structures. When your task comprises matching breadth splitting structures, look out for all-quantified patterns, iterated patterns, amalgamated rules, queries with set semantics, or OCL-expressions in the right column of Table F.15. They allow to concisely and declaratively capture those structures. When you need both combined, look out for recursive and iterated patterns. If pattern languages need to revert to control to implement them (because your task at hand requires them), they become uncompelling; have a look at the example solutions in Section 3 to see this effect on a simple example.

7.2.2. Expressiveness by Approach

Due to a strong cohesion within the approaches and the cross cutting nature of the features, we will first visit the four main subtasks related to expressiveness approach-by-approach, before we compare the features directly, discussing their consequences.

Patterns. The pattern-based approach employs in addition to the main pattern already introduced with the approaches patterns as negative application conditions and positive application conditions. Negative patterns prevent the containing pattern from matching in case they can be found (defining a not-exists operation). Positive patterns in contrast must be present in addition to the main pattern. These patterns define constraints, their elements are not available for rewriting. A reduced version of them may be available with single negative/positive elements. A generalized version of them are nested negative/positive patterns; a second level negative re-allows a pattern forbidden by a first level negative. The attributes are processed with additional attribute conditions, which allow to specify logical formulas over comparisons of attribute values and constants.

Furthermore, alternative patterns may be available to match alternative substructures, or iterated patterns, which allow to match a pattern as often as it is available in the graph. Subpatterns allow to factor out a recurring pattern and use it from different patterns. These constructs can be employed as plain application conditions comprising only a left hand side pattern. They may be endowed with an additional right hand side, then specifying a complex rule (which is applied in one step after everything was found, in contrast to a recursive call in a common programming language that carries out matching and rewriting in each single step).

Matching Structures Extending into Depth with Patterns. Subpatterns combined with alternatives allow to recursively match structures into depth, as chains of patterns. Subpatterns need to be called with explicit parameter passing to this end. The parameter passing can be omitted in the common case that paths, i.e., only chains of nodes connected by edges need to be matched. In this case the more concise regular and iterated path constructs may be employed. Regular path expressions allow to match an iterated path constraining the incident edges and adjacent node types on the path. They subsume the iterated path, which allows to find out whether a node is reachable from another one, without the possibility to constrain the types

(a special form of this is transitive containment along containment edges). The explicit parameter passing for subpatterns is less convenient and concise than the implicit passing in regular paths, but it allows in combination with the iterated patterns to declaratively match tree structures.

Matching of Breadth Splitting Structures with Patterns. Iterated patterns allow to capture breadth splitting structures. So do the equally expressive nested rules, which are following the notion of rule amalgamation from theory, where a kernel rule is extended by repeated rule parts as required. Reduced versions of these constructs are available with multi-nodes and loop-header-nodes. A multi-node allows to match a pattern and then a node incident to the pattern multiple times. A loop-header-node allows to match the header node multiple times, and then for each instance the pattern. It can be seen as a loop from a control language notationally integrated into a pattern language; especially when it can be directly assigned, which allows to follow an iterated path.

Patterns are typically matched with existence semantics, the right hand side of the rule is applied on the one match of the left hand side that was found. After this execution, the rule can be applied again, but only on the then-changed state. Alternatively, a pattern may be matched with for-all semantics, a rule is then applied on all matches found for its left hand side. The availability of for-all semantics for the main pattern of a rule alone is only a step towards matching a breadth splitting structure, as the fixed kernel part must be matched before and handed in via parameters. The for-all and existence semantics of the top level pattern can be generalized with quantified patterns: an all-quantified nested pattern is matched multiple times (similar to an iterated pattern), an existentially quantified nested pattern once. The ability to nest such patterns to an arbitrary (but statically fixed) depth lifts the expressiveness of pattern based tools considerably above the expressiveness of the basic approach.

Query Languages. Being built for SQL like querying of graph repositories or databases, query languages comprise a direct match regarding the question for the means available for general data querying. They allow to query graphs declaratively for connected structures without incurring side effects, and report the found data back in the form of collections of tuples, i.e., container variables of a complex type.

Geared towards the extraction of information distributed over a graph, they contain built-in means for querying into depth. GReQL, the only query

language which was used for solving the Hello World case, employs regular path expressions as already introduced above. Breadth structures are collected into a result set by the implicit for-all matches semantics of query execution. Patterns can be described but incur some notational overhead over dedicated pattern languages. The data structure with the data resulting from the query is passed as input to the transformation code.

OCL and Non-Pattern-Based Rules. In the non-pattern-based rules typically only one source element is matched, and OCL expressions are used as application conditions to constrain the rule application. Or the rule is described by its effects, formulated by a precondition OCL expression and a postcondition OCL expression. OCL may be employed from pattern-based tools and from program-based tools, too. The former is rather seldom, though.

OCL combines attribute conditions and their single-value semantics with navigational expressions for querying the neighboring elements (or all elements in a model); when only one neighbor is allowed by the metamodel, a single model element is returned, but otherwise a collection of all neighboring elements is returned. A further nested expression can then be formulated for all elements in the container. This allows to easily capture structures extending into breadth, but is inconvenient for simple existence matching, read: describing patterns. Depth structures can only be followed by nesting expressions, i.e. to a statically fixed depth.

Programs and Control. In the program-based approach, queries over the model are programmed, with elementary model queries, which are glued by statements, i.e., control flow and state variables. Depth structures can be matched by loops with a variable storing the current node; in each iteration step the variable is advanced following an edge by one step into depth. Alternatively they can be matched by recursive calls with the current node as input parameter. Breadth structures are typically matched by loops with a variable iterating over the neighboring elements. So everything can be expressed, but everything must be always explicitly expressed in solutions based on tool-supplied or external programming languages.

These looping or calling schemes may be applied from the control language of a language based on declarative transformation units, too. Besides loops iterating single-element variables you may employ container variables to store the results of queries, esp. of a statically not known number of nodes.

Programs allow for concise solutions for tasks, which are outside the expressiveness of the other approaches, or even a solution at all, but less concise

and not declarative solutions for tasks, which fall inside the expressiveness of the other approaches.

7.2.3. Expressiveness Compared

After the explanations of the important features alongside the approaches, we now turn towards the direct comparison. We start with the means available for data collection and for storing data collection results, under display in Table F.14, before we continue with the means available for matching structures extending into depth and structures branching off into breadth in Table F.15.

Data Collection. Different means for data collection are available. Patterns may be used and extended with certain kinds of application conditions. Or a one element query may be combined with an OCL expression – alternatively OCL expressions may be used stand alone. Furthermore, a dedicated query language may be used. Or simply a manually coded program. Not listed here are task-specific pre-coded data collection routines for constrained tasks, which may in fact build the backbone of a special-purpose tool. We repeat our recommendation to choose the more general constructs, e.g. to favour nested negative patterns over negative elements, unless you know that the simpler constructs are sufficient.

Result Storing. Different kinds of variables are available to store the results from the data collection for later processing. This is a basic feature of query languages, and commonly supported by programming languages. OCL expressions are explicitly designed to be free of side effects and do not allow to store results. Pattern based rules define an immediate effect not storing any results, but they may return matched or created elements to the control language, so for rule- and control-based languages this is a discrimination point of the control language. Storing results allows to omit repeating searches for elements that were already found, thus increasing performance. It furthermore allows to decompose a task into phases coupled by the stored data, which is beneficial for complex tasks. The price of this feature is a reduced declarativeness for transformations and a susceptibility to ordering effects.

Depth. The means available for solving this subtask were already introduced with the approaches. Best suited to query for structures into depth are regular path expressions, followed by the more powerful but less concise recursive

patterns, followed by the less powerful iterated path expressions. OCL expressions are not able to describe depth drilling. On the non-declarative side, we can capture depth-extending structures with loops or recursive calls. This holds for the program-based approach as well as for the control layer of a rule-and-control based language. In case that structures extending an unlimited time into depth *and* into breadth are to be matched (e.g. trees), only a combination of iterated and recursive patterns is up to the task, or a manually coded recursive subprogram. Please note that some constructs allow to modify the items matched into depth, e.g., reversing all edges visited, while others do not. Look out for this ability if your task at hand requires it.

Breadth. The means available for solving this subtask were already introduced with the approaches. OCL expressions offer a concise solution with their set based semantics for capturing attached nodes, and allow to capture more complex neighboring structures with expression nesting, albeit much less concisely then. GReQL offers a concise solution with its query result sets, and allows to capture and return more complex neighboring structures. Multinodes allow to capture attached nodes concisely, but cannot be generalized to more complex attached structures at all. Loop header nodes allow to capture attached nodes concisely, but can only be generalized to more complex structures with the help of rule control. The nested rules, or iterated patterns, or quantified patterns are a bit less concise for single attached nodes, but are especially well suited to capture attached patterns.

On the non-declarative side, we can capture breadth-extending structures with loops or recursive calls. This holds for the program-based approach as well as for the control layer of a rule-and-control based language. The availability of rules which can be applied with for all semantics is a help in that case.

Please note that some constructs allow to modify the items matched into breadth, e.g., linking all nodes visited to another node, while others do not. (OCL expressions for example are not capable of this modification, while rules of OCL expressions are.) Look out for this ability if your task at hand requires it.

7.2.4. Selected Features

In Table F.16 some more detailed features are listed; most of them were of importance for the Hello World tasks and played a role in the reviewer

comments and the votes. Again, it depends on your task at hand whether they are of importance for you.

Retyping. Retyping, also known as relabeling in graph rewriting, allows for an in-place change of the type of a node while keeping its incident edges. This feature allows rewrite tools to achieve concise solutions for simple 1:1 mapping tasks, it especially allows pattern-based tools with their weakness in describing a statically not fixed context to achieve concise solution.

Transaction Support. Transactions allow to roll back changes carried out on the model at request. Transactions render programming for search-based tasks easier, as it is possible to just try a transformation and to roll it back to the original state when it failed according to some criteria, instead of being forced to specify a complex condition that needs to be checked beforehand, or instead of being forced to explicitly undo the effects. Transactions are only important for tasks where you must search for an optimal model, for plain transformation tasks they are not needed.

Modifier Matching. The matching modifiers constrain the way in which pattern elements may be matched to graph elements; they only apply to pattern-based tools. A single graph element may be allowed to get matched by multiple pattern elements (homomorphic matching), or not (isomorphic matching). Languages that only offer isomorphic matching must duplicate patterns for tasks where non-isomorphic matching is needed. Languages that only offer homomorphic matching require the reader to always think of all the possibilities in which pattern elements may get coalesced by matching them to the same graph element. Tools offering both allow you to flexibly choose according to your current needs.

Modifier Rewriting. The rewriting modifiers constrain deleting nodes in case edges not mentioned in the pattern would dangle; they only apply to pattern-based tools. SPO allows to delete a node even if not all edges were specified in the pattern, which is for most tasks the more practical approach. With DPO semantics, proving propositions about graph rewrite systems is much easier (this is of interest for verification). This is another occurrence of the inability of the pattern-based tools to describe a statically not fixed context.

7.2.5. Programming in the Large and Reuse

"Is the tool suited to a large and complex transformation task?" is a question that is answered to a good degree by the points regarding expressiveness

we already visited, but additionally by answers to the questions "What means of abstraction and reuse are available?" and "What means of structuring large specifications are available?". In Table F.17, the capabilities of the tools for transformation programming in the large are listed. Of course it depends on the size of your task at hand whether or to what extent these abilities are really needed.

Model and Computation Modularization. Is it possible to combine the *metamodel* specification in a project from parts in different files or views, which can be reused and edited separately? This allows for metamodel structuring and reuse, as needed by tasks with large metamodels. And as a refinement: do they support different namespaces, so that the parts can be combined without name clashes? The refinement is important if models out of different origins need to be combined.

Is it possible to combine the *computations* in a project from parts in different files or views, which can be reused and edited separately? This allows for computation structuring and reuse, as needed by tasks with large specifications. And as a refinement: do they support different namespaces, so that the parts can be combined without name clashes? The refinement is important if computations from different programmers need to be combined.

Abstraction and Parameterization. The most basic unit of reuse in the world of transformation are entire (mapping-like) transformations between representations, which are combined by transformation concatenation, building a pipeline architecture. Transformations that would be too complex for one pass are split into multiple passes linked by intermediate representations.

Besides this external reuse one needs to ask in how far the units from which the transformations are built can be reused: When transformation elements are abstracted into own units, what are these units, and how can they be parameterized? We concentrate in Table F.17 on what a subprogram of the program-based approach would offer, just split into the different units employed by the approaches (as induced by their layering).

The more abstraction and parametrization possibilities are available, the better; they allow to factor out and re-use common parts.

7.2.6. Extensibility, Meta Programming, and Runtime Flexibility

The extensibility of the languages and tools, their support for metaprogramming, and their flexibility concerning runtime changes are listed in Table F.18.

Extensions. "I have a subproblem the tool does not allow me to solve, what can I do?" is a question that arises if the tool is used outside a pure transformation setting (for which the tool and especially its languages were designed for). The tools introduced in this article for example typically do not natively support matrix or vector classes in the model and matrix-vector-multiplications in the attribute computations, or the filtering of matches by additional queries against a persistent database before they are applied.

The transformation may be extended with external program code in two ways:

- With externally-defined transformation operations that are callable in place of the built-in operations, or in place of operations specified in the language of the tool. They allow to fall back to a programmed model for tasks where this is more appropriate; in contrast to an API, which is used from outside the tool languages, they allow to utilize programmed operations from within the tool, e.g., calling an external operation from the rule control language of the tool.

- With externally-defined attribute types and attribute computations; e.g., for introducing a type matrix and a matrix multiplication as operation, which are opaque to the tool. Externally-defined attribute types and attribute computations are helpful when the attributes offered by the tool as given in Table F.13 and the operation available on them are not sufficient for the task at hand.

Meta-Programming. "Can I program my programs?" or in our setting "Can I transform my transformations?" is a question you are interested in getting an answer for in case your task at hand leads to repetitive specifications with a lot of boilerplate code.

When the transformations are available as models to be inspected and changed, the model transformation tools introduced here are able to synthesize the transformations with another user-written transformation, giving meta programming virtually "for free".

A reduced version of this functionality is available with the meta-iteration, which allows to iterate over all available types in the model, not naming the specific types. Tasks that require that a lot of types are treated in the same way can be specified concisely in a loop over the types, instead of being explicitly and statically enumerated in the code.

Meta-programming allows for the concise specification of tasks where a lot of similar computations arise, at the price of understandability. Tools that do not offer these capabilities require in that case either copy-n-paste programming or to program a tool-external code generator, which emits specifications in the tool's format.

Runtime Adaptability. "Can I adapt my metamodel or my computations at runtime?" is a question that arises when the tool has to adapt to environments not fully known at specification time in a flexible way. The capability to change the metamodel at runtime allows to build the target metamodel of a mapping transformation when executing the transformation. The capability to change the computation at runtime allows a rule based language to change the rules from the rule set while the transformation is running, depending on intermediate results (i.e. meta-programming at runtime).

7.3. The Languages and the User Interface Refined

"Does the user interface of the tool fit to my needs or preferences?" was given a first answer for in Section 6.4 by listing the languages offered by the tools and their form (distinguishing textual from graphical). Here we refine these points by comparing the support offered by the tools in developing transformations in Table F.19, and by a comparison of how easy it is to get acquainted with the tool and its languages.

7.3.1. Development Support

"What kind of support do I get by the tool when developing transformations?" is the question to be answered with the help of Table F.19. The domain specific languages introduced in this article *as such* are better suited to transformation problems, and the typically offered visual style of programming and debugging is often more adequate for transformation tasks – but the tools commonly fall short in advanced editing support like auto-completion or refactorings offered by the IDEs of general-purpose programming languages.

IDE and Editor Integration. "Can I use my favorite Integrated Development Environment or my favorite editor to develop my transformations?" If so, you can save the effort of learning a new IDE or editor. Have a look at Table F.19 to find out about the IDEs and editors offered.

Development Support. We focus on the assistance offered for debugging and editing your transformations. Regarding debugging we are especially interested in the abilities to visualize the model. Regarding editing we query for help in basic editing with auto completion, and for the ability to refactor the specifications, e.g., to consistently rename an entity definition; without refactoring support the changes must be carried out by hand, e.g., for the renaming with a search and replace in a textual editor, fixing the identifiers which were captured accidentally afterwards.

7.3.2. Learnability

"How easy is it to learn the tool and its languages?" Learning a new language is an investment that should pay off in the end, with a reduced total time needed for development and maintenance. Most tools offer own languages, which require considerable effort to catch up; the more expressive the language and the more powerful the tool (and thus the more to gain), the more effort is commonly required beforehand. But a documentation of high quality is of help here, as well as maybe already available knowledge.

User Documentation. The types of user documentation available are listed in Table F.20. They tell you where to look at first and give a hint on the overall documentation support you can expect.

Direct. Further on learning a tool may be supported by reusing already existing notations, from standardized programming or specification languages.

Concepts. Learning a tool may be fostered by conceptual reuse. So we ask which knowledge from what domains is helpful to understand the tool, reason by analogy. The more proficient you are in the domains specified, the easier should the transformation language be to learn for you.

7.4. The Environment and the Execution Refined

"In which environment can I use the tool?" was given a first answer for in Section 6.5. Here we refine it with Table F.21, listing what you have to set up to develop transformations, and what you have to set up on the computers of the user of your transformations.

Prerequisites Execution. "What must be installed on a computer besides the bare OS to run developed transformations?" The less the better. This is a much stronger requirement than the following prerequisites for the development, as each end user of the transformations is forced to install these prerequisites, or the transformation developer is forced to install them on the end user's computer with a setup routine together with the transformations as such.

Prerequisites Development. "What must be installed on a computer in addition to the execution prerequisites to develop transformations?" The less the better. The prerequisites listed give a hint on the effort required to get the tool running. While a long list for the development prerequisites is not much of a hindrance once the decision for a tool offering compelling features was taken, it is a large obstacle for prospective developers or reviewers in software engineering workshops, which just want to evaluate the tool. The pre-installed SHARE images of the tools are an important help in this case.

7.4.1. License and Maturity

"What is the price of the tool?" "Is the source code available?" "How mature is the tool?" You find answers to those questions in Table F.22.

License. The answers to the first two questions are linked to the license. When the source code is available you can fix bugs and extend the tool on your own, even after the tool runs out of support, so this defines a kind of insurance. Furthermore, the license has a strong influence on the price to pay, which may be measured in money or in code. Choose what you can afford (or what is compatible with the license of the transformation you intend to develop).

Maturity. Tool maturity gives hints on tool stability and usability, (typically the older and larger tools have an advantage here), but also flexibility in carrying out changes due to user requests (typically the younger and smaller tools have a higher degree of freedom).

7.5. Validation and Verification

"How can I ensure my transformation does what it should do?" is the primary question we answer in Table F.23, extended by answers to the question "How can I use the tool to ensure that things that I modeled are doing what

they should do?". The helpers for manual inspection (graph viewer, debugger) were already compared in Section 7.3.1, here we ask for the support for automatic checking. We begin by comparing the support for checking model integrity.

Constraints. Metamodel or type graph constraints may be enforced or may be checkable, regarding the allowed attributes, regarding the allowed edge types between certain node types, and regarding the allowed multiplicities of edges of a given type at specific nodes.

Facts of type checked can be violated during the transformation, this eases transformation writing. On the other hand it is easier to introduce errors as you must not forget to trigger the checking. The enforced checks may be in fact not checks at all, but consequences of the chosen model implementation: enforced attribute type checks are typically a consequence of model kind fixed as defined in Table F.13), commonly implemented by statically typed classes containing the attributes as member variables. Types and multiplicities being enforced is a hint at model-based tools, which implement edges by reference or container variables, whereas graph tools typically allow implementation-wise an arbitrary number of edges at each node. In case checks like these are not provided, they can be of course programmed.

Validation and Verification. Besides the basic checks for model integrity, OCL expressions may be used to validate the model; describing and checking constraints on object structures is in fact the primary reason for their existence. The tools may even offer a dedicated validation language extending OCL with e.g., refined user feedback. Furthermore, the tools may offer to clone a graph and rewrite it until an answer regarding a very complex constraint is reached.

Finally, the tool may be based on a correct-by-design approach: the user is not writing code to be executed, but describes precondition and postcondition formulas, which are then implemented by the code generator of the tool. Moreover, the tool may allow for formal verification of its transformations, with theorem proving support.

Until now we spoke of validating or verifying the developed transformations. A task some of the tools introduced here were created for is verifying other systems. To this end, they offer model checking by state space enumeration (SSE). A state space enumerator allows to enumerate a state space of graphs generated by applying a rule set, visualizing the space structure as

well as the single states. Unfolding the space of different executions is a help in manual inspection, too. But its primary use is in automated model checking: it is tested whether temporal logic formulas hold for all the enumerated states.

8. Related Work

A respectable number of tool comparisons was already carried out on *complex* cases with a *small* number of compared tools. This article in contrast summarizes the first (to the best of our knowledge) *large-scale* study based on an *instructive* case, consisting of several *simple* tasks.

In the invitation to the Model Transformations in Practice Workshop [5], a Class-to-RDBMS scenario was introduced, that defined one of the first and most influential cases for comparing model transformation tools. Classes containing attributes linked by associations in between them were to be transformed to tables containing columns with foreign key constraints in between them. In contrast to the Hello World case that comprises several very simple tasks, the Class-to-RDBMS case consists of one clearly more *complicated* core task (plus an advanced task, and several additional tasks). In Model Transformations by Graph Transformation [95], the transformation tools AGG, ATOM3, VIATRA, and VMTS were compared using the aforementioned object-to-relational transformation example in-depth, with QVT chosen as a 5th comparison partner; leading to the conclusion that there are a large number of commonalities, with the main differences to be found in the description of attribute computations and in the control of rule applications. The paper defined a first effort in comparing transformation tools, using up about 50 pages for a comparison of just 4 graph transformation based tools, highlighting one of the problems of comparisons alongside complex cases: a lot of time must be invested by a reader to understand the case alone, and even more to understand the solutions of the tools, before any conclusion can be drawn when searching for a tool to employ.

In the Comparison of Three Model Transformation Languages [27], the transformation languages CGT, AGG, and ATL are compared alongside "a fairly complicated refactoring of UML activity models", in the words of the authors. The paper attributes the more concise solution of CGT over AGG and ATL to the usage of a collection operator, which is a declarative language construct for processing breadth-splitting structures in our terminology (and

to the advantages of concrete syntax-based graph transformation as implemented by CGT).

The paper The Jury is still out: A Comparison of AGG, Fujaba, and PROGRES [18] comparing the three aforementioned tools in-depth is similar in its goal of helping a user to choose a tool, leaving it to the prospective user to draw the final decision.

Transformation Tool Contest Based Comparisons. Several cases of earlier editions of the Transformation Tool Contest have also resulted in the publication of tool comparisons, being the major source for those.

In the article Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools [100] originating from the AGTIVE 2007 Tool Contest, multiple tools are compared regarding a transformation from UML activity diagrams to formal CSP processes. The languages of UML activity diagrams and of CSP processes, as well as a non-trivial transformation in between the two languages have to be understood in order to be able to follow the solutions and the comparison of the eleven participating tools.

In the article Graph Transformation Tool Contest 2008 [83], and in A Case Study to Evaluate the Suitability of Graph Transformation Tools for Program Refactoring [78] are the different cases of the 2008 edition of the contest, its execution, and its results explained; concerning a program refactoring case (with the complexity of the program graphs and the transformations listed as key challenges by the authors), the simulation of an ant population with a focus on performance, a realistic transformation from BPMN to BPEL models, and a live contest case involving the scheduling of a conference. Several separate articles were published for the different transformation tools and their solutions to the cases, for AGG/EMF Tiger [6], Fujaba [21], GrGen.NET [46], Kermeta [76], MoTMoT [77], VIATRA2 [43], and VMTS [74]; as well as GROOVE [23], showcasing further tasks.

In Graph and Model Transformation Tools for Model Migration [86] are nine graph and model transformation tools compared with a focus on empirical evaluation alongside a migration of UML activity diagrams from version 1.4 to version 2.2, a more elaborate version of the prototypical Simple Migration task of the Hello World case. The decisive features for achieving good results were automatic copying of elements based on name equivalence, and retyping (cf. Section 7.2.4). The tools that scored best in voting were Epsilon Flock and COPE (the predecessor of Edapt), specifically designed for model migration, with GrGen.NET following as first general-purpose tool.

The aforementioned comparisons concentrated on the languages of the tools and on development support. The performance of execution is typically investigated in dedicated benchmarks. The paper Benchmarking for Graph Transformation [103] proposes a benchmark for comparing the performance of graph transformation languages. On Improvements of the Varró Benchmark for Graph Transformation Tools [22] corrects and extends those initial measurements. In Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools [94] are multiple tools compared regarding their performance – execution time and memory consumption – in creating Sierpinski triangles. The results showed a huge difference in between the slowest and the fastest tools, but also highlighted that the fastest tool for the task (FUJABA) was coming near to a hand-coded solution maximally tuned for performance.

The Comparisons of the 2011 Edition of the TTC. The Hello World case aimed at introducing the tools and illustrating their approach was complemented by a Program Understanding and a Compiler Optimization case[23], which evaluate how well the tools are suited to complex tasks involving large workloads.

The Program Understanding case [40] required to extract a state machine model out of an abstract syntax graph (ASG) of a computer program, according to some patterns in the ASG. A prerequisite was the ability to import XMI. The case was designed to measure the ability of tools to capture non-local data; it required to follow chains of potentially unbounded length. So declarative depth support as given in Table F.15 was of high importance to achieve good results here, as underlined by the voting results, with GReTL offering regular path expressions scoring first, GrGen.NET offering recursive patterns scoring second, and MDELab SDI offering iterated containment paths third.

The Compiler Optimization case [10] required to carry out constant folding (which is trivial for data flow and complex for control flow) and instruction selection on a graph-based compiler intermediate representation. A prerequisite was the ability to import GXL. The case was designed to measure the performance of the competing tools. Achieving a good result was aided by the ability to store visited nodes in containers as given in Table F.14, this allowed to fold following the flow. Performance-oriented tools, which are

[23] and a Model Migration case, but this one received only two submissions

able to handle large data sets achieved good results here, as underlined by the voting results with GrGen.NET scoring first, GReTL second, and GROOVE third.

We want to note that in those two complex cases, as well as in Graph and Model Transformation Tools for Model Migration[86], the tools of the case proponents won. An in-depth case measures not only how well a tool is performing, but notably how well a tool is adapted to that specific case. If your task at hand is very similar to the case investigated, you find very good answers regarding the suitability of the compared tools. But regarding a different task, the results will be likely substantially different. Is is often difficult to *generalize* the results of complex cases. Multiple in-depth cases with their results accumulated would be ideal to assess general-purpose usability, saving users from learning a new transformation language and environment whenever they need to solve a new transformation task. Unfortunately, this would be excessively time consuming, for readers as well as tool providers[24]. For this reason is the article built on Hello World, a collection of prototypical tasks – the performance of the tools in solving those simple tasks gives rough hints at their performance when facing more complex problems of the same kind. But especially on a taxonomy with feature matrices telling about the usability of the tool for certain aspects and subtasks that a complex task can be broken into. This allows to *estimate* tool performance for a task at hand, for a wide range of many different tasks.

Taxonomies. The taxonomy that was applied to the tools in this article was developed based on the lessons learned from the cases, and inspired by the taxonomies introduced in [12] and [72] for classifying transformation tools, tasks, and solutions. The Feature-based survey of model transformation approaches [12] presents a classification model for transformation languages based on their technical properties. The Taxonomy of Model Transformations [72] introduces different applications of transformation languages and tools, and presents functional as well as non-functional requirements for transformation approaches; it was applied on AGG, Fujaba, GReAT, VIATRA in [73]. Our taxonomy is surpassing the aforementioned papers in its approach towards explaining expressiveness, but especially in its direct application to the large amount of tools that participated in the TTC.

[24] each single tool typically only appears in a small selection of the in-depth comparisons carried out for this reason

9. Summary and Conclusion

The Transformation Tool Contest 2011 was a research workshop for tool providers interested in the pros and cons of their tools regarding diverse applications. Their motivation was to understand the relative merits of different tool features in order to improve their tools and to advance the state of the art. This article is one in a series of articles in the history of this research event, casting the comparison results into paper form. But in contrast to all previous and many other in-depth comparisons carried out, do we not only target members of the tool building community, but also prospective users. The instructiveness of the Hello World case solved for the contest, and the ready availability of virtual machine images of the solutions allowed for an article also aimed at helping in choosing a tool – helping you.

We took you by the hand and assisted you on your way through the maze of the numerous available tools, in order to find a tool that is well-suited to your task at hand. First, we introduced the different tools with a calling card, that included an illustrative example solution of one of the tasks of the Hello World case. We further discussed the solutions, in order to balance wrong impressions you could get from only seeing this one example; especially explaining the importance that the support for processing structures splitting into breadth played for it.

We want to note that the task descriptions of the small, prototypical tasks of the Hello World case contain some semantic ambiguities. An upgraded version with more accurate descriptions would help in future tool comparisons, as would the inclusion of a task requiring to match structures extending into depth.

Then we gave an overview of the positions of the tools in the tool landscape, explaining core notions and discrimination points of the field on the way. The feature matrices employed there allowed you to quickly order and reduce the set of candidates. In the second, more detailed step based on the same setup of building a taxonomy and applying it directly to the tools did we answer multiple questions you typically ask when you have to choose a tool – or you would ask if you would know their importance, with a special focus on the topic of computational expressiveness.

This survey includes the largest number of tools compared up to now; we must note that it still does not include all of the state-of-the-art tools – a worthwhile endeavor would be an extension with the missing tools yielding a full market survey.

Besides helping in choosing a tool and painting a detailed picture of the tool landscape, this article gave an example for software engineering in the cloud[25]. We explained how cloud based virtual machines were used during the Transformation Tool Contest. SHARE gave rise to reproducible and strongly validated results, ameliorating the problems in reviewing the complex software toolkits employed. It allowed to review real solutions instead of only paper digests, and lowered the hurdle to learn from other tools.

The SHARE images did not only improve the review process, but they allow you now to investigate the tools in-depth, according to *your needs*. We must note, though, that the images are not yet available for anonymous access; this unfortunately builds a barrier that requires effort to be surpassed, it is not unsurmountable, though: we want to encourage you to write to the SHARE maintainer(s) in order to gain access.

Acknowledgments. We want to thank Stephan Hildebrandt for his contributions to this article, the organizers of the Transformation Tool Contest for rendering this comparison possible, and the SHARE maintainers fur supplying the virtual machines used in the contest. Furthermore, we want to thank the reviewers for their valuable comments. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

[25]Software engineering *in* the cloud is an application of cloud computing for the benefit of software engineering, as opposed to the older topic of software engineering *for* the cloud, which is concerned with building applications for the cloud

Appendix A. Problem Domain and Solution Domain

Appendix A.1. Problem Domain

In a nutshell, we are confronted with a graph rewriting or model transformation problem if the most adequate representation of the data at hand is a mesh of objects, and we need to change its structure or map it to another mesh of objects.

A *model* is an abstract representation of a system, which captures exactly the characteristics of the system that are of interest to the model designer [90]; a complete system may be covered by models at various abstraction levels described from different points of view. Models are typically built from entities and relationships between these entities, forming a network of interrelated objects. They conform to a metamodel (built from entities and relationships) to be defined by the designer while modeling. The metamodel in turn conforms to a fixed metametamodel.

The mathematical concept of a *graph* is very similar to the model notion: Here the entities are called *nodes*, and the relationships *edges*, forming a network of interconnected nodes. We regard both concepts to be equivalent and use them interchangeably. A programmer is facing a graph- or model-based problem if the domain of interest requires a representation of the system to be modeled as network of objects. Problems that can be adequately modeled with scalar values or lists do not benefit from any of the tools introduced in this article.

One domain in which the introduced tools are helpful is model-driven development according to the Model-Driven Architecture vision [75] of the OMG. In this approach, the central artifacts in the software development process are models. Software is developed by defining metamodels and implementing transformations between them, which finally yield an executable model or program code. In this context, we are typically confronted with *mapping* problems, which require to translate a higher-level program representation into a lower-level program representation. Other domains in which these tools are helpful include, among others, mechanical engineering [33], computer linguistics [3], and protocol verification [82]. There, we are typically confronted with *rewrite* problems, which require modifying one graph by a sequence of transformation steps. In each of these steps, a graph pattern needs to be matched and replaced by another pattern until a goal state is reached.

Appendix A.2. Solution Domain

Given a problem as described above, what are the options in solving it? You could either

1. code it by hand in a high-level programming language of your own choice, or

2. use one of the transformation tools introduced in this article.

The question that now naturally arises is: What are the benefits of using one of the tools compared to manual coding? The direct, simple answer is *reuse*. A tool offers a proven and tested implementation, which was already debugged and is ready to be used. Reusing an existing tool allows a developer to achieve his goals quicker and with less effort.

The languages offered by the tools are typically of a much higher *expressiveness* for tasks of the domains of model transformation or graph rewriting than general-purpose programming languages. This holds because subproblems of the domain, which required explicit imperative code before were solved and made *declarative* by the tool providers through implementing a code generator or a runtime library (that emits or contains the imperative code the user would have had to write on his own before).

Solutions specified in these special-purpose languages are *concise* compared to solutions coded in general-purpose programming languages. This leads to a decrease in the development and maintenance costs as the developer needs to write less code during the development and read or change less code during maintenance. Additionally, graph and model transformation tools may offer a more user-friendly graphical access to models and model transformations. Often, these tools offer a *visual* style of specification and debugging (or simulation).

Appendix A.3. Potential Benefits and Drawbacks

We now discuss in more detail the potential benefits and drawbacks of using a transformation tool instead of a manually implemented solution.

Graph and Model. In the object-oriented paradigm, node types would be typically implemented by classes, nodes by objects, and edges by references stored in the source object, pointing to the target object. In contrast, a transformation tool or a special-purpose language may offer:

- A more concise specification. For the easiest manually coded solution the benefits would not be compelling. However, optimizing the implementation to achieve a high performance (especially for pattern matching) is technically challenging.
- A more elaborate model, e.g., featuring attributed edges.
- Run-time adaptability of the metamodel.
- Reuse of importer/exporter code.
- A graph viewer that can be used to visually inspect the model, instead of being forced to chase chains of references in the debugger of the programming language used for a manual implementation.
- A graphical metamodel editor.
- An explicit model interface that abstracts from the underlying modeling, enabling the reuse of computations on different modeling technologies.

Rewriting and Transformation. The most simple manually implemented solution would navigate the object graph with loops, explicitly manipulating the processing state. In contrast, a transformation tool or a special-purpose language may offer:

- A more concise specification, with e.g., navigational expressions.
- A declarative specification with rules, which specify the graph patterns to be matched and modified.
- An already implemented rule execution engine, which takes care of the matching of rules, or a concise special-purpose language for this task.
- Declarative pattern matching and rewriting, instead of nested loops iterating incident edges or adjacent nodes.
- A graphical debugger, which allows the developer to stepwise follow the rule executions, highlighting the currently matched rule in the graph.
- A graphical rule editor.

- An already implemented and tested library for predefined high-level tasks.
- Higher performance than an unoptimized manually implemented solution.

Potential Problems. There are also possible drawbacks in using a transformation tool compared to a manually implemented solution:

- The time needed to learn the tool and its languages. The higher productivity achieved when using a tool is offset by the initial effort to learn the tool.
- Reduced flexibility w.r.t. transformations implemented in general purpose programming languages in general. Especially missing flexibility regarding tasks the tool was not designed for; they might lead to complicated solutions. This holds in particular for the case that the requirements change after the project start.
- Increased deployment effort due to the transformation engine and its prerequisites.
- Potential performance problems because of an unoptimized engine. This might be the case with a manually coded solution as well, but optimizing a tool supplied by others is a more difficult task.
- Maintenance problems due to the dependency on a third-party component; esp. vendor lock-in – the developer might be forced to abandon a solution in case the tool is not maintained anymore, esp. if it is closed source.
- Lack of advanced IDE features commonly offered for general-purpose programming languages, such as refactorings.

Appendix B. Hello World Case, the other Tasks

The *Constant Transformation* task ("Constant Transformation and Model-to-Text Transformation") and the *Deletion* task ("Deletion of Model Components") were already introduced in Section 2. Here we give digests of the other tasks of the Hello World Case. Note that certain subtasks had been

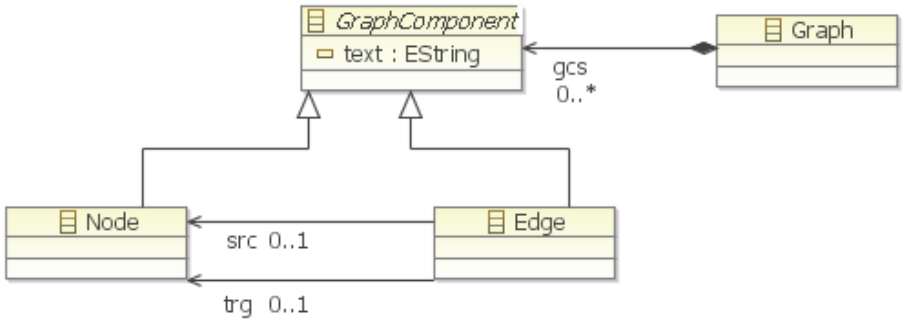

Figure B.18: The evolved graph metamodel

marked as optional, i.e., those have not been required to solve the case but can be considered only as extensions. The original descriptions can be found in [70].

Appendix B.1. Pattern Matching

The *querying* tasks asks for model queries counting certain elements in the graph conforming to the metamodel given in Figure 3. Numbers wrapped into an object of a result type have to be returned.

Asked-for are the number of nodes, the number of looping edges, the number of isolated nodes, the number of matches of a circle consisting of three nodes in a graph, and optionally the number of dangling edges.

Appendix B.2. Simple Replacement

The *update* task requires to provide a transformation reversing all edges in a graph (conforming to Figure 3).

Appendix B.3. Simple Migration

The *migration* task asks for a transformation migrating a graph conforming to the metamodel given in Figure 3 to a graph conforming to the metamodel given in Figure B.18. (The `name` of a node becomes its `text`. The `text` of a migrated edge has to be set to the empty string.)

An optional task is to provide a topology-changing migration into graphs as defined by the metamodel in Figure B.19.

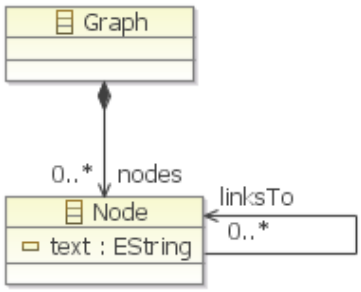

Figure B.19: The even more evolved graph metamodel

Sub-task	CRUD	Transf.	Compl.	Flow	Language	Abstraction
Constant transformation	C	creation	simple	output	endogenous	horizontal
Pattern matching	R	query	simple	input	exogenous	vertical
Simple replacement	RU	turnover	simple	input	endogenous	horizontal
Simple migration	RC	turnover	simple	input	exogenous	horizontal
Deletion	RD	spot	simple	input	endogenous	horizontal
Transitive edges	RC	turnover	simple	input	endogenous	horizontal

Table C.4: Categorization of "Hello World" case sub-tasks.

Appendix B.4. Transitive Edges

The optional transitive edges task asked for the insertion of an edge `e3` in between the nodes `n1` and `n3`, in case there was a node `n2` existing, with an edge `e1` in between `n1` and `n2` and an edge `e2` in between `n2` and `n3`.

Appendix C. Discussion of Hello World

To provide an insight in the coverage of the "Hello World" case, Table C.4 lists a number of transformation-related properties covered by each sub-task of the case. These properties are based on experiences from the TTC workshop series, as well as transformation tool surveys by [72] and [12], as typical distinguishing factors between the different transformation languages/tools. The properties are *CRUD*, *transformation* kind, transformation *complexity*, control *flow*, *language*, and *abstraction*. The *CRUD* property stands for *Create/Read/Update/Delete*, and refers to the kind of model manipulations required by the transformation problem. For the *transformation* kind we distinguish *creation* for a write-only transformation, from *query* for a read-only transformation, from *turnover* for a transformation that rolls around most parts of the model, from *spot* for a transformation that changes only

a small part of the model, which means an *in-place* transformation carried out by a *rewriting* tool is better suited to it. *Complexity* refers to the kind of transformation that is required, and can be either *simple* or *heavy*. *Simple* problems allow the transformation to be described as a relatively straightforward relationship between input and output elements, or *mapping*. *Heavy* problems are more complex, and require intermediate model states that are transformed in a stepwise fashion until the desired output is achieved (favoring *rewriting* tools). All transformation sub-tasks were *simple* problems. The contol *flow* refers to the kind of navigation required to generate the desired output. It can be either *input-driven*, where the transformation output can be fully quantified by the input (i.e., n output elements for each input element), or it is *output-driven*, where a specific output template structure is required, regardless of the kind and amount of input elements. The *language* property refers to either *endogenous* transformations, or *exogenous* transformations (i.e., either within the same language, or between different languages). The *abstraction* property refers to either *horizontal* (same abstraction level) or *vertical* (different abstraction levels) transformation problems.

Appendix D. Votes and Discussion

Here, we publish the votes for the Hello World case, which were cast along the dimensions completeness, understandability, and conciseness in steps of 1 to 5 points, with 5 being the highest score. Additionally, we give an interpretation and discussion comparing the solutions and tools in the order of voting results. The discussion is based on the opponent statements, enriching them with further post-workshop insights of the authors. Only the tools that were presented at the TTC were voted; this is why the EMFTVM, the metatools, and QVTR-XSLT are missing from Table D.5. PETE, which was presented at the workshop, is not included in this article.

Completeness was of low impact compared to conciseness and understandability, with the worst solution in this regard scoring at 95% of the maximum value (compared to 56% and 57% regarding the other dimensions). This high rate of success is not surprising taking into account how basic the tasks were; in fact it is rather surprising that a third of the tools was not able to give a complete/correct solution in the first place. So the matter was decided alongside understandability and conciseness. Regarding understandability, three points played a role: (i) the distinction into graphical versus textual languages, with a general bonus for graphical tools, (ii) the concepts the

Criteria	GReTL	UML-RSDS	MDE Lab	Groove	EMF Henshin	Epsilon	Edapt	MOLA	Via-tra2	GrGen .NET	PETE
Compl.	5,00	4,83	4,77	5,00	5,00	4,93	5,00	4,86	5,00	5,00	5,00
Underst.	2,92	2,85	4,08	3,67	3,69	4,33	3,47	4,00	3,86	3,77	2,92
Concisen.	4,23	3,54	3,46	4,54	3,38	4,33	3,47	3,57	2,79	3,77	3,62
Average	4,05	3,74	4,10	4,40	4,03	4,53	3,98	4,14	3,88	4,18	3,85
Rank	6	11	5	2	7	**1**	8	4	9	3	10

Table D.5: Outcome of the voting for the tool solutions

tools are built upon, constructs from formal logics received a malus, and (iii) whether the tool offers a syntax similar to well-known programming languages, which was preferred. Regarding conciseness, the availability of (i) lightweight means for simple CRUD tasks played a role, as offered by imperative solutions, but even more so (ii) the general expressiveness of the tools, as expressed by the availability of the features referenced in the feature matrices; they had not to be used into great depth, but their general availability already lead to more compact solutions compared to competing tools.

Appendix D.1. Epsilon

The Epsilon solution was able to employ a language matching the task at hand for nearly each task of the Hello World case. So the solution scored highly regarding conciseness and understandability, rendering it the clear winner of this challenge. But there is one exception to the understandability and conciseness: as no declarative pattern matching is offered, the match a cycle of nodes task had to be coded imperatively with 3 nested loops. Offering multiple languages has another side effect, which was also complained about by the opponents: learning becomes harder.

Appendix D.2. GROOVE

GROOVE was able to solve each task with one rule, which resulted in the highest conciseness vote of all competing solutions. This was made possible by the use of quantified nodes – but while yielding the top vote regarding conciseness, they lead to only a midfield place and complaints from the reviewers regarding understandability. A further point criticized was the lack of XMI import and export.

Appendix D.3. GrGen.NET

GrGen.NET was as balanced as Epsilon regarding conciseness and understandability, but at a clearly lower level. Strong points regarding conciseness were the retyping (cf. Table F.16) for the migration task, which earned it kudos from the opponents, and the subpattern iteration. Complaints were fielded regarding the lack of an exporter for XMI, which was manually coded, and an imperatively formulated transitive edges solution.

Appendix D.4. MOLA

The MOLA solution received the 3rd highest vote regarding understandability, but only a midfield result regarding conciseness. It can be expected that the graphical language for both rules and control leads to the high vote for understandability while reducing the conciseness score.

Appendix D.5. MDELab SDI

The MDELab SDI solution received the 2nd highest vote regarding understandability, which most likely must be attributed to its clean graphical syntax. But this was offset by the 3rd lowest vote regarding conciseness, with the opponents complaining about the lack of non-isomorphic matching (cf. Table F.16) and the unavailability of negative patterns (cf. Table F.14), which would have been useful for this case.

Appendix D.6. GReTL

The GReTL solution scored 3rd highest regarding conciseness based on the highly compact language for the transformation of query result sets. But on the other hand it scored 2nd worst regarding understandability, which must be attributed to a lot of special tokens not known from common programming languages, which seem to stem from formal specification languages the audience was not trained in.

Appendix D.7. Henshin

Henshin scored in the midfield for understandability, which is rather surprising for a graphical rule-based language, especially compared to MDELab SDI and MOLA – it seems that the non-activity-diagram-based control language gave a minus in this regard. It scored 2nd worst regarding conciseness, with an opponent criticizing it for the lack of a pure apply-rule-for-all-matches construct not requiring a kernel rule.

Appendix D.8. Edapt

Edapt offered a strong solution for the model migration task for which the tool is designed, only reusing some predefined operations. For the other tasks, hand-coded solutions against an API were offered, which gave complaints by the opponents regarding a lack of declarativeness and conciseness and a reduced vote in this areas.

Appendix D.9. VIATRA2

VIATRA2 was seen by the voters to belong to the top third regarding understandability, but received a devastating vote regarding conciseness. While clearly being one of the most verbose languages, the effect seems disproportional, which most likely is to be attributed to the solution introduction paper that presented several possible solutions for each of the tasks; while being in accordance with the goals of the Hello World case it seems that people just glimpsing over paper got the impression that there is a huge amount of code needed.

Appendix D.10. PETE

Pete received kudos for its declarative and potentially bidirectional style for specifying transformations with Prolog terms. On the other hand, this most likely gave it a bad vote regarding understandability with an audience not trained in this language. In ended in the midfield regarding conciseness, with an opponent praising it for a concise solutions regarding the counting task and denouncing it for a verbose solution regarding the model-to-text task.

Appendix D.11. UML-RSDS

UML-RSDS scored in the midfield regarding conciseness but worst regarding understandability. The opponents termed the solution interesting regarding its basic approach while at the same time asking about the use of mathematical formulas for the simple Hello World task; it seems the voters did not appreciate the predicate logic style of specification either.

Appendix D.12. Validity and Full Solutions

The Hello World task descriptions, despite being simple, contain several ugly corner cases and ambiguities (e.g., reversing dangling edges), that showed when they were to be implemented. Some tool providers delivered a correct solution offering exactly the required features, at the price of a more complex and thus less concise and readable solution, while others were flexible in the interpretation of the requirements, concentrating on the pragmatics of the case, i.e., introducing their tools. Some voters took this into account and gave lower votes regarding completeness based on correctness, while others just ignored this issue or were even surprised that some participants voted them in this regard.

The votes cast can be seen as good indication on the overall performance of the tools regarding understandability and conciseness, but should be taken with a grain of salt. On the one hand, they are tweaked by the presentational skill of the tool provider before the audience, and on the other hand by the inquisitional skill of the opponents. For while in principle every tool provider could evaluate all tools and then give an all encompassing vote, this rarely happened with more than a dozen competing tools and solutions; the SHARE images of the solutions typically were investigated by the opponents and a handful of interested, but not all participants.

Regarding understandability one should take care of the fact that this criterion is highly dependent on the experience and knowledge of the voters. Tools offering a notation based on general programming constructs fared better than tools based on constructs from formal specification languages or formal logic. While this gives a good indication on the mass market compatibility of a tool and the appeal to the average software engineer, it might give a wrong impression on the understandability of a tool regarding the knowledge available to a prospective user.

Conciseness was judged subjectively, too, based on the presentation and the paper. Presenting only the concise parts could have lead to a better vote then deserved regarding overall performance. Some help in this regard and especially regarding the question of correctness might come from a transformation judge [71], a computer program under development judging objectively.

Appendix E. Feature Matrices for Section 6

Appendix E.1. Table E.6

Domain. We note down

graph for graph-based tools.

model for model-based tools.

Of high importance in distinguishing in between the two notions are the supported input formats and output formats defining the technical space [59] the tools operate in: model transformation tools are normally built on UML/-MOF/XMI/Ecore, whereas graph transformation tools are typically built on GXL or custom formats. Graph-based tools are typically more general purpose, geared towards all domains that operate on graph like representations.

Tool	Domain	Kind	Traceability
Edapt	model (EMF)	rewriting	preserve
EMFTVM	model (EMF)	both	implicit
Epsilon	multi (EMF, ...)	both	implicit, explicit
GReTL	graph	mapping	implicit
GrGen.NET	graph	rewriting	explicit, preserve
GROOVE	graph	rewriting	explicit
Henshin	model (EMF)	rewriting	explicit
MDELab SDI	model (EMF)	rewriting	explicit
metatools	model	both	explicit
MOLA	model (EMF, ...)	both	explicit
QVTR-XSLT	model	mapping	implicit
UML-RSDS	model	both	explicit
VIATRA2	mixed	rewriting	explicit

Table E.6: Domain and Kind

In contrast, model tools are typically centered on program representations. Graph transformation tools tend to view edges as first level constructs, which might be attributed or endowed with an inheritance hierarchy, while model transformation tools tend to view edges as references between nodes without own attributes. The world of graphs, which is older and more mature, enjoys a higher level of mathematical sophistication compared to the more practically minded world of models (sometimes at the price of regarding practical usability an inferior topic).

In addition to these two opposite values, there are two refinements available:

mixed is used for tools that are located in the middle of the two domains.

multi is used for tools that offer to switch the metametamodel.

Tools normally provide one fixed metametamodel based on which the users can define the metamodel to work with. Tools classified as multi allow to switch the modeling technology to work with different metametamodels. They may show a different behavior depending on the chosen metametamodel. In cases where this influences the features we compare, we hint at it with a table footnote.

The modeling technology for tools that are not using a custom model, but are built on the API of a common technology, like EMF for the Eclipse

Modeling Framework [92], is given in parenthesis. The advantage of such a setup is that other tools built on that technology can be easily (re)used; a disadvantage may arise from the fact that it is not possible to adapt the model implementation to the computations, esp. that it is not possible to tweak it for more efficient matching.

Kind. We note down:

mapping for mapping based tools (operating on several models).

rewriting for rewriting based tools (operating on one model).

both for tools that support both kinds.

Traceability. The support for traceability links can be distinguished into:

implicit Traceability is built-in, source and target elements are automatically linked by the transformation engine.

explicit Traceability must be explicitly coded by assignments to variables of map type.

preserve Traceability can be modeled by keeping the identity of the transformed entity in a rewrite-based tool supporting retyping, cf. Table F.16.

Appendix E.2. Table E.7
Languages. We distinguish:

DDL A data definition language to specify the metamodel (see Section 6.2).

PL A programming language (as explained in Section 6.3).

QU A query language (as explained in Section 6.3).

RL A rule language (as explained in Section 6.3).

CL A control language (as explained in Section 6.3).

SPs stands for tools that are horizontally split into several special-purpose languages.

sp is noted down for tools that offer a user interface that can be seen as a domain specific language, but one that does not allow for general transformation programming.

Tool	Languages offered	Attributes	External Languages
Edapt	sp for migration		Extensions : Java
EMFTVM	RL	OCL	API : Java
Epsilon	PL and SPs, one: RL	OCL	API : Java
GReTL	QU		Main : Java
GrGen.NET	DDL, RL, CL(/PL)	Java-like	API(RL,CL), Events : C#
GROOVE	DDL, RL, CL		
Henshin	RL, CL	JavaScript	API, Events: Java/EMF
MDELab SDI	RL, CL	OCL	API, Events: Java/EMF
metatools	SPs: DDLs, RL	Java	Main: Java
MOLA	DDL, RL, CL	OCL-like	Extensions, API : Java, C++
QVTR-XSLT	RL	XSLT	API : XSLT
UML-RSDS	RL, CL	OCL	
VIATRA2	DDL, RL, PL(/CL)		

Table E.7: Offered languages and extensions

Attributes. In addition to the languages concerned with structural transformation, the sub-language offered for attribute computations is given.

External Languages. We distinguish:

Main The computations are meant to be programmed in a general-purpose programming language.

Extensions The computations that are exceeding the functionality of a supplied library are to be programmed in a general-purpose programming language.

API A user program written in a general-purpose programming language can manipulate the model or call the computations via an API.

Events A user program may hook into events that are fired when the model is manipulated or a rule is matched.

For all points the programming language that needs to be employed is of interest.

Events fired on model changes finally allow for an even tighter integration of tool supplied data and computations into an externally written program.

Tool	**M.-M.**	**Computations**	MM	RE	CE
Edapt	G	G (T for extensions)	×	×	×
EMFTVM	G or T	T, T	×	×	×
Epsilon	G or T	T	×	×	×
GReTL	G or T	T	×	×	×
GrGen.NET	T	T, T	×	×	×
GROOVE	G	G, T	✔	✔	×
Henshin	G	G, G	×[1]	✔	✔
MDELab SDI	G or T	G, G	×[1]	✔	✔
metatools	T	T	×	×	×
MOLA	G	G, G	✔	✔	✔
QVTR-XSLT	G	G, T	✔	✔	×
UML-RSDS	G	T, T	✔	✔	×
VIATRA2	T	T, T	×	×	×

[1] third-party editors for EMF-based metamodels are T or G

Table E.8: Form of specification language

Appendix E.3. Table E.8

Form of Specification Language. We note down:

G for graphical languages like UML class or activity diagrams.

T for textual languages.

The form is applied to the:

Metamodel for the metamodel specification.

Computations for the computation specification.

The computation specification comprises one value for tools of non-rule kind, and two values for tools of the rule and control kind, first the rule, then the rule control part.

Graphical Editors. In addition, we list the availability of graphical editors, differentiated in between the different language parts:

MM stands for a graphical metamodel editor, which allows to edit the metamodel in a notation similar to UML class diagrams.

Tool	Execution host	Operating System		
		Windows	Linux	Mac OS X
Edapt	EXE, IDE(Eclipse), APP	✔	✔	✔
EMFTVM	IDE(Eclipse), APP(JVM)	✔	✔	✔
Epsilon	IDE(Eclipse), EXE, APP(JVM)	✔	✔	✔
GReTL	EXE, APP(JVM)	✔	✔	✔
GrGen.NET	EXE, APP(.NET)	✔	✔	✔
GROOVE	EXE, APP(JVM)	✔	✔	✔
Henshin	APP(JVM), IDE(Eclipse)	✔	✔	✔
MDELab SDI	APP(JVM), IDE(Eclipse)	✔	✔	✔
metatools	APP(JVM), EXE	✔	✔	✔
MOLA	IDE(Eclipse), APP(JVM, Win)[1]	✔	×	×
QVTR-XSLT	EXE, XSLT	✔	✔	✔
UML-RSDS	API(JVM)	✔	✔	×
VIATRA2	IDE(Eclipse), EXE, APP(JVM)	✔	✔	✔

[1] Target environment dependent (Java or C++)

Table E.9: Execution host and supported OS

RE stands for a graphical rule editor, which allows to edit the left hand side and right hand side patterns of the rules in a notation similar to UML object diagrams.

CE stands for a graphical rule control editor, which allows to edit the control flow in a notation similar to UML activity diagrams.

Graphical editors bring graphical languages to life; while a graphical language without an own editor would be unusable, a textual language, esp. a pattern-based one might come with an additional graphical editor. Graphical editors offer the benefits of visual programming, but bind the user to that editor (in addition to the underlying execution engine, as is the case for the textual languages).

Appendix E.4. Table E.9

Execution Host. We distinguish:

EXE The tool suite contains an external executable, which allows to run the transformation, e.g., a shell application or a simulator.

Tool	M.-M.	Comp.	Engine	Memory
Edapt	I	I and C	UP	48bytes, 40bytes
EMFTVM	C	C, C	SB(stat), UP	not tool defined[2]
Epsilon	C or I[1]	I	UP	not tool defined[1]
GReTL	C or I	I	SB(dyn)	36bytes, 60bytes
GrGen.NET	C	C, C or I	SB(dyn)	44bytes, 56bytes, 4bytes
GROOVE	I	I, I	SB(stat) or INC	40bytes, 48bytes, 48bytes
Henshin	I	I, I	SB(dyn), UP	not tool defined[2]
MDELab SDI	C or I	I, I	SB(dyn)	not tool defined[2]
metatools	C	C		
MOLA	C	C, C	SB(stat)	not tool defined[1]
QVTR-XSLT	C	C, C	SB(dyn), UP	
UML-RSDS	C	C, C		
VIATRA2	I	I, I	SB or INC	1kbyte, 1kbyte, 3kbytes[3]

[1] depends on underlying modeling technology
[2] depends on EMF implementation (approx. 48bytes, 32bytes)
[3] due to canonical model representation

Table E.10: Execution model

IDE The tool suite contains a plugin for an IDE that is able to execute and debug the code.

APP The transformations are executed from an user application, which is accessing the transformation via an API. (On what kind of machine?)

Operating System. We distinguish:

Win Windows,

Lin Linux or Unix,

Mac Mac OS X.

Appendix E.5. Table E.10
Tool Execution.

Execution Model. We note down:

C for compilation or code generation.

I for interpretation.

The execution model is applied to the:

Metamodel for the metamodel specification.

Computations for the computation specification.

The computation specification comprises one value for tools of non-rule kind, and two values for tools of the rule and control kind, first the rule, then the rule control part.

Engine. We distinguish the following matching engine approaches:

SB for search-based – queries are answered or patterns are matched by a search in the graph, based on a plan scheduled by the search engine. A discrimination point within this approach is the time when the search plan is computed, a static one is computed at specification time, a dynamic one can be (re-)computed at runtime to better fit to a currently given model.

INC for incremental – all matches of all rules are stored in a Rete network, model changes percolate through this network to yield all matches of all rules after the model change. This allows for very fast queries at the price of slow updates and increased memory consumption. The less rules, the better the performance.

UP for user programmed – patterns are sought by nested navigational expressions or nested loops; here the user can easily optimize the matching order, but on the other hand, the user is always forced to define the matching order.

Memory. We give the amount of memory required by a plain node without attributes, followed by the amount of memory required by a plain edge without attributes, followed by the additional amount of memory required by an 4-byte integer attribute of a node.

Appendix F. Feature Matrices for Section 7

Appendix F.1. Table F.11

The formats listed are:

Tool	Input formats	Output formats
Edapt	XMI	XMI
EMFTVM	XMI, TXT	XMI, TXT
Epsilon	XMI, XMI(1.x), XML	XMI, XMI(1.x), XML
GReTL	CST, XMI, GXL	CST, XMI, GXL, VIZ
GrGen.NET	CST, GXL, XMI	CST, GXL, TXT, VIZ
GROOVE	GXL, XMI	GXL, XMI, VIZ, VER
Henshin	XMI	XMI, VIZ, VER
MDELab SDI	XMI	XMI
metatools	TXT, XML(dtd)	TXT, XML(dtd)
MOLA	XMI, CST	XMI, CST, TXT
QVTR-XSLT	XML, XMI, GXL, CST	XML, XMI, GXL, CST
UML-RSDS	CST	CST, XMI
VIATRA2	XMI, CST	XMI, CST, TXT

Table F.11: Input and Output

GXL: an XML dialect, the de facto standard for graph transformation tools, specifying the graph as well as the type graph.

XMI: an XML dialect, the de facto standard for model transformation tools, specifying the model. The metamodel for an .xmi is typically given in .ecore format (version 2.0).

XMI(1): in case the tool supports XMI 1.x.

XML: general support for XML.

CST: custom formats, which are not helpful in tool interoperability (unless adopted by other tools thus defining a de facto standard), but unleash the full potential of a tool.

TXT: means the tool has support for text output into a file, which allows to emulate an arbitrary file format (at the price of explicit coding). If this is specified for input, the tool offers integration with a parser generator, e.g., EMFText [32].

VIZ: visualization formats emitted by the tool, which are used as input for a graph visualization tool (e.g., `.dot` or `.vcg`).

VER: file formats for verification, e.g., `.aut`.

Tool	Nodes	Edges	Modifier					
			+M	+U	+O	+RC	+I	+B
Edapt	M, A	P	✔	×	✔	✔	✔	×
EMFTVM	M, A	P	✔	×	✔	✔	✔	×
Epsilon	S[1]	[1]	✔	×	✔[1]	×	×	×
GReTL	M, A	M, A	✔	×	✔	✔	×	✔
GrGen.NET	M, A	M, A	✔	✔	×	×	×	✔
GROOVE	M, A	S	×	×	×	×	×	✔
Henshin	M, A	P	×	×	×	✔	×	×
MDELab SDI	M, A	P	✔	✔	✔	✔	✔	×
metatools	M, A		✔	×	✔	×	×	×
MOLA	M, A	P	×	✔	✔	✔	✔	✔
QVTR-XSLT	M, A	P	✔	×	✔	✔	×	×
UML-RSDS	S, A	P	×	×	✔	✔	✔	×
VIATRA2	M, A	M	✔	×	×	✔	×	×

[1] depends on the underlying modeling technology, values given for EMF

Table F.12: Nodes and Edges

Appendix F.2. Table F.12

The ways in which the nodes or edges may be typed are:

U: means the elements are untyped.

P: for plain means the elements are equipped with disjoint types.

S: for single inheritance means the elements are equipped with a type hierarchy specifying a can-be-substituted relation with one single parent for each type.

M: for multiple inheritance means the elements are equipped with a type hierarchy specifying a can-be-substituted relation with potentially multiple parents for each type.

Additionally, an element may bear attributes, denoted with **A**.

The modifiers for how edges (resp. references) are exactly realized are:

+M: If the edges have an identity, i.e., multiple edges of the same type between two nodes are allowed, we speak of a Multi-Graph; we note

down +M in this case. This is an extension of a simple graph, which allows only one edge of the same type in between two nodes. The simple graph is the natural choice for problems, which are strictly built on relations, where multiple edges between same nodes do not have any meaning.

+U: Edges are normally directed, with one node distinguished as source and the other as target. There might be additionally undirected edges. In this case, we hint at them by noting down +U.

+O: Edges in a graph are typically unordered (in contrast to an array of references). In case the tool is able to query for and assign the position of edges, this is denoted by +O for ordered.

+RC: Model transformation tools typically distinguish edges into references and containment, and e.g., implement a cascading delete of children if the parent is deleted. It is indicated by +RC that this is the case.

+I: The tool supports inverse edges if creating or deleting an edge automatically creates or deletes a co-edge in the opposite direction.

+B: The tools supports traversing edges in both directions (bi-directional navigationability). This is common for graph tools, the co-edge handling can be seen as a way to achieve this behavior in a model-based tool.

Appendix F.3. Table F.13

The available attribute kinds are:

Liberal: means each element can bear arbitrary attributes.

Fixed: means a type defines uniquely which attributes an element may bear.

The types available for typing attributes are: **b** for boolean, **i** for integer numbers, **f** for floating point numbers, **s** for strings, **e** for enumerations, **o** for objects (user-defined types, which are opaque to the tool language, they make only sense when the tool can be employed from an API embedded in a host program), **C** for collection types (e.g., set, map, bag, array, list).

Tool	Attribute Kind	Attributes						
		b	i	f	s	e	o	C
Edapt	Fixed	✔	✔	✔	✔	✔	×	✔
EMFTVM	Fixed	✔	✔	✔	✔	✔	✔	✔
Epsilon	Fixed	✔	✔	✔	✔	✔	×	✔
GReTL	Fixed	✔	✔	✔	✔	✔	×	✔
GrGen.NET	Fixed	✔	✔	✔	✔	✔	✔	✔
GROOVE	Fixed or liberal[1]	✔	✔	✔	✔	×	×	×
Henshin	Fixed	✔	✔	✔	✔	✔	✔	×
MDELab SDI	Fixed	✔	✔	✔	✔	✔	✔	✔
metatools	Fixed, Liberal	✔	✔	✔	✔	✔	✔	✔
MOLA	Fixed	✔	✔	×	✔	✔	×	×
QVTR-XSLT	Fixed	✔	✔	✔	✔	✔	×	×
UML-RSDS	Fixed	✔	✔	✔	✔	✔	×	×
Viatra2	Liberal	✔	✔	✔	✔	✔	×	×

[1] The user can either set a type graph fixing the allowed attributes, or leave the model untyped

Table F.13: Attributes

Tool	Data collection	Result storing
Edapt	PL, OCL	container
EMFTVM	P, NP, PP, AC or	container
	E, OCL	none
Epsilon OL	PL, OCL	container
Epsilon TL	E, OCL	none
GReTL	Query Language: GReQL	compound container
GrGen.NET	P, NNP, NPP, AC	container
GROOVE	P, NNP, NPP, AC	none
Henshin	P, NNP, NPP, AC	variable
MDELab SDI	P, NE, PE, PP, AC, OCL	container
metatools	P, PL, NNP, NPP, AC	container
MOLA	P, NE, PE, AC	variable
QVTR-XSLT	P, PP, AC	none
UML-RSDS	OCL	none
Viatra2	P, NNP, NPP, AC	container

Table F.14: Data collection

Appendix F.4. Table F.14

The available abbreviations for data collection are:

P: for the main pattern.

E: for the single element from the source to be mapped.

NP: for a negative pattern.

PP: for a positive pattern (used as application condition).

NE/PE: for negative/positive elements.

NNP/NPP: for nested negative/positive patterns.

AC: for attribute condition.

OCL: for conditions formulated in the Object Constraint Language.

PL: for data collection programmed in a programming language.

The different ways to store results of the data collection:

none: it is not possible to store results

variable: for single valued variables.

container: for container variables.

compound container: for compound containers (capable of storing records, similar to the table-valued result of a SQL query).

Appendix F.5. Table F.15

The abbreviations employed in explaining the depth and breadth matching capabilities are:

Imper: for an imperative solutions with loops or calls, implemented in a programming language or a rule control language.

Reg: for regular path expressions.

Rec: for recursive patterns.

Pat: for a pattern.

Tool	Depth	Breadth
Edapt	Imper	OCL, Imper
EMFTVM	Imper	OCL, Multi-nodes/RHS, Imper/RHS
Epsilon	Imper/RHS	OCL, Imper/RHS
GReTL	Reg-Path	Result-Set/RHS
GrGen.NET	Rec-Pat/RHS, Imper/RHS	Iterated-Pat/RHS, F-All, Imper/RHS
GROOVE	Reg-Path	Quantified-Pat/RHS
Henshin	Imper/RHS	Nested rules/RHS, Imper/RHS
MDELab SDI	Imper/RHS	OCL, F-All, Imper/RHS
metatools	Reg-Path, Imper	Reg-Path, Imper
MOLA	Imper/RHS	Loop-with-header/RHS, Imper/RHS
QVTR-XSLT	Rec-Pat	Iterated-Pat
UML-RSDS	Imper	OCL/RHS, Imper/RHS?
VIATRA2	Rec-Pat	F-All, Imper/RHS

Table F.15: Depth and breadth support

F-All: for rules applied with for-all semantics.

RHS: abbreviates right-hand-side; this tells that it is possible to modify the elements matched.

We add the suffix **RHS** to the solutions listed in Table F.15 if we can modify the items matched into depth, e.g., reversing all edges visited.

Appendix F.6. Table F.16

The values for retyping are:

cross: in case it is supported without restrictions.

down: if only downcasts are allowed (being type safe but less powerful).

none: if retyping is not supported.

The values for transaction support are:

simple: if transactions are supported.

nested: if they may be nested.

none: if transactions are not supported.

The values for the matching modifier are:

Tool	Retype	Transaction	Mod. Mat.	Mod. Rew.
Edapt	cross	none	-	-
EMFTVM	none	none	iso, hom	DPO
Epsilon	cross[1]	simple	-	-
GReTL	none	none	-	-
GrGen.NET	cross	nested	iso, hom	SPO, DPO
GROOVE	down	simple	hom, iso	SPO, DPO
Henshin	none	simple	iso, hom	SPO, DPO
MDELab SDI	none	none[3]	iso	SPO
metatools	none	none	hom	-
MOLA	none	none	hom, iso	SPO
QVTR-XSLT	none	none	hom	SPO
UML-RSDS	none	none	hom, iso	SPO
VIATRA2	cross[2]	nested	iso, hom	SPO

[1] depends on the underlying modeling technology, values given for EMF
[2] only supported by the rule control language
[3] transactions are supported in the debugger

Table F.16: Retyping, transactions, and pattern modifiers

iso if a single graph element cannot get matched by multiple pattern elements.

hom if a graph element can get matched by multiple pattern elements.

The default is specified first.

The values for the rewriting modifier are:

SPO if the rule is applied and the edges that would dangle are deleted.

DPO if the rule is prevented from matching and nothing happens.

The default is given first.

Appendix F.7. Table F.17

A check mark in the model column (Mod.) tells that the metamodel specification can be combined from different files or views; a second check mark denotes the availability of separate namespaces.

A check mark in the computations column (Comp.) tells that the computation specification can be combined from different files or views; a second check mark denotes the availability of separate namespaces.

Tool	Mod.	Com.	Units of abstraction and reuse
Edapt	✔/✔	✔/×	P(I)
EMFTVM	✔/✔	✔/×	C(I,O), R(I,O), S(I,O)
Epsilon OL	✔/✔	✔/×	P
Epsilon TL	✔/✔	✔/×	R, S
GReTL	×/×	✔/×	Q(I,O), P(I,O)
GrGen.NET	✔/×	✔/×	C(I,O), R(I,O), S(I,O)
GROOVE	✔/×	✔/✔	C, R(I,O)
Henshin	✔/✔	✔/✔	C(I,O), R(I,O)
MDELab SDI	✔/✔	✔/✔	C(I,O), R(I,O), S(I,O)
metatools	partly		P(I,O)
MOLA	✔/✔	✔/×	C(I,O)
QVTR-XSLT			R(I,O)
UML-RSDS	✔/×	✔/×	C(I), R(I)
VIATRA2	✔/✔	✔/✔	C(I,O), R(I,O), S(I,O)

Table F.17: Transformation organization and reuse

Abstraction and Parameterization. In the program-based approach, a program is the basic unit of use. If it is in addition possible to factor out common program computations and reuse them from other program parts, we note down **P** for a subprogram. The subprogram subsumes all the other means of reuse offered by alternative approaches.

In the library-based approach, library functions are the basic unit of use. As it is not possible to abstract them further we omit them here.

In the query-update-based approach (as introduced in Section 6.3), a query is the basic unit of use. When it is in addition possible to factor out common queries and reuse them from other queries, we note down **Q** for subqueries.

In the rule-based approach, a rule is the basic unit of use. The explicit control sequence or an implicit control engine is an accompanying basic unit of use. When it is in addition possible to factor out a control sequence of a rule control language (or a program when the rule control language includes a programming language), and reuse that unit as a compound control unit, we note down **C** for a sub-control program. When it is in addition possible to factor out a rule and reuse it from another rule, we note down **R** for a subrule. When it is in addition possible to factor out a computation below rule level and reuse it from another rule, we note down **S** for subrule com-

Tool	Extensions	M-I	M-P	Gen.	Model	Comp.
Edapt	EOP(Java)	✔	×	×	✔	✔
EMFTVM	EOP(Java), EAC(Java)	✔[1]	✔	×	×	✔[2]
Epsilon	EOP(Java)	✔	×	✔	✔	×
GReTL	EOP(Java)	✔	×	×	✔	✔
GrGen.NET	EOP(C#), EAC(C#)	×	×	×	×	✔
GROOVE	-	✔	✔	×	×	×
Henshin	EAC(JavaScript)	×	✔	✔	✔	✔
MDELab SDI	EOP(Java), EAC(Java)	×	✔	✔	×	✔
metatools	Comp. *are* external	✔	✔	×	×	✔[2]
MOLA	EOP(Java, C++)	×	✔	×	×	×
QVTR-XSLT	EOP(XSLT)	×	✔	×	×	×
UML-RSDS	EOP(Java), EAC(Java)	×	×	×	×	×
VIATRA2	EOP(Java), EAC(Java)	✔	✔	✔	✔	×

[1] only supported by the rule control language
[2] self-reflective adaptation of the currently running transformation is *not* supported (for tools not supporting M-P this holds directly)

Table F.18: Extensions, metaprogramming, and runtime flexibility

putations; this includes e.g., application conditions and subpatterns. **C** is helpful when complicated transformations need to be programmed. **R** saves you from returning output parameters to the control language to be used as input parameters for following rules, for tools of the rule and explicit control approach, which is a cumbersome practice, and allows implicit control tools to use other rules as a reusable entity. **S** is helpful when there are common structures to be processed in the same way or when there are non-trivial attribute computations to be carried out from several rules.

If these units of reuse support parameterization, we add:

I if parameters passed in are supported.

O if parameters passed out after applying the part are supported.

Appendix F.8. Table F.18

The following values are available for the Extensions column:

EOP Externally-defined transformation operations can be called in place of the built-in operations, or in place of operations specified in the language of the tool.

EAC Externally-defined attribute types and attribute computations can be defined; e.g., for introducing a type matrix and a matrix multiplication as operation.

The programming languages in which these extensions may be programmed is appended.

A check mark in the M-I column denotes the capability to program with meta-iterations. A check mark in the M-P column denotes tools in which the computations are available as models to be inspected and changed at runtime by meta programs.

A check mark in the Gen. column denotes the capability for generic programming in the sense that metamodels are passed as parameters at runtime, i.e., the metamodels are not statically known (as supported e.g. by Dynamic EMF). A check mark in the Model column denotes the capability to change the metamodel at runtime. A check mark in the Comp. column denotes the capability to change the computations at runtime.

Appendix F.9. Table F.19

The IDE and editor column lists the IDEs for which a tool integration is available; an editor integration must consist of at least syntax highlighting.

The following columns under developer support are checked when the corresponding functionality is available:

GV Graph viewer – a graph viewer allows to view the graph before the transformation and after the transformation.

HG Hierarchical graph support – the graph viewer allows to view a nested graph, where nodes are capable of containing further graphs. This feature is important for large graphs, which are not consumable by the human mind if presented in a flat layout.

GD Graphical debugger – a graphical debugger allows to execute a transformation stepwise, rule match by rule match, and to view the matching pattern highlighted in the graph.

DB Textual debugger – a debugger for a textual transformation language is available.

CD Change difference – the tool offers a viewer to highlight changes in a model.

Tool	**IDE & Editor**	**Developer support**						
		GV	HG	GD	DB	CD	RF	AC
Edapt	Eclipse	✔	×	×	×	✔[3]	✔	×
EMFTVM	Eclipse	×	×	✔	✔	×	×	✔
Epsilon	Eclipse	×	×	×	✔	×	×	×
GReTL	Eclipse, Emacs	✔	×	×	×[2]	×	×	✔
GrGen.NET	N++;[1] Vim, Emacs	✔	✔	✔	×	×	×	×
GROOVE		✔	×	✔	×	×	×	✔
Henshin	Eclipse	×	×	×	×	✔[3]	×	×
MDELab SDI	Eclipse	✔	×	✔	×	✔[3]	×	×
metatools		✔	×	×	✔[2]	×	×	×
MOLA	Eclipse	✔	×	×	×	✔[3]	✔	✔
QVTR-XSLT	MagicDraw UML	×	×	×	×	×	×	×
UML-RSDS		×	×	×	×	×	✔	×
VIATRA2	Eclipse	✔	×	×	×	×	×	×

[1] N++ abbreviates the editor Notepad++.
[2] The debugger of the general-purpose programming language is available.
[3] EMFCompare can be used to compare models and visualize differences.

Table F.19: Developer support

Tool	User documentation	Direct	Concepts
Edapt	Tut, API, Expl		PL, OO
EMFTVM	UM, Tut, Wiki	OCL	PL, OO
Epsilon	UM(238), Tut, LR	OCL	PL, OO, RP
GReTL	API, Pap, Expl		SL, DQ, PL, OO
GrGen.NET	UM(250 incl. LR), Expl, API	Java	PL, OO, RP, FLA
GROOVE	Tut, Expl, UM(25)		FLO, PL, GL, RP
Henshin	Tut, Wiki, Expl	JavaScript	OO, GL, RP
MDELab SDI	IPHelp, Expl, API	OCL	OO, GL, RP
metatools	Pap, UM(200), API	Java	FLA, PL, SL, OO
MOLA	Web, UM(28), LR(60)	OCL	PL, OO, GL, RP
QVTR-XSLT	Web, Expl, Pap	OCL	GL, RP
UML-RSDS	UM(20), Expl, Pap	OCL,UML	FLO, SL, OO, GL
VIATRA2	Wiki(42), UM(115), LR(88)		PL, RP, DQ

Table F.20: User documentation and learnability

RF Refactoring – an editor that allows for some refactorings of the tool language is available.

AC Auto completion – an editor that helps in editing by suggesting completions for the current editing operation is available.

Appendix F.10. Table F.20

The following types of user documentation may be listed in Table F.20: **UM** for user manual, **Tut** for a tutorial, **Wiki** for a wiki, **LR** for a language reference, **API** for an API documentation, **Expl** for examples, **IPHelp** for an in-program help, e.g., Eclipse help, **Pap** for scientific papers, and **Web** for a web site. Numbers appended in parenthesis specify the amount of A4 pages when printed. The three most important pieces of documentation are given in descending order.

The column direct lists the standardized programming or specification languages whose notations are reused in the tool, for attribute evaluation and assignment, or application conditions, as the computations as such.

Concepts. The concepts column may list one (or multiple) of the following abbreviations in case knowledge from the corresponding domain is helpful in learning the tool:

FLO Formal logic. For predicates, quantifiers, proofs.

SL Specification languages, e.g., Z, B. For set comprehension and set operations, logic operations, map operations, and their specific syntax.

FLA Formal languages. Grammar rules of nonterminals as placeholders for things yet to be derived/parsed (subpatterns) are employed, and terminals (nodes and edges). Programming with recursion.

DQ Database queries. A query language returning a set of tuples and an update language.

PL Programming languages are characterized by: block nesting, keywords, definition and use of identifiers, statement sequences, expressions for attribute evaluation, and procedure calls.

OO Attributed classes with inheritance hierarchies over types and method call notation.

GL Graphical languages: UML class diagrams, UML object diagrams, UML activity diagrams.

RP Rule-based programming and pattern matching. Specification by matching of inputs, generation of outputs.

Appendix F.11. Table F.21

The Prerequisites Execution column lists the third-party software an end user is forced to install in order to get a transformation developed with the tool running (copying a few runtime libraries supplied by the tool, which can be shipped with a solution do not count into this).

The Prerequisites Development column lists the third-party software the transformation developer needs to install additionally in order to develop transformations with the tool (besides the software components directly shipped with the tool).

Appendix F.12. Table F.22

The License column lists a combined value from

open the source code is available to the public.

closed the source code is not available, only binaries are supplied.

and a price value:

Tool	Prerequisites Execution	Prereq. Development
Edapt	Java	Java, Eclipse (ME)
EMFTVM	Eclipse, EMF, ATL core	ADT[1]+ SimpleGT
Epsilon	Java	Eclipse
GReTL	Java	Java
GrGen.NET	.NET 2.0 (or mono)	Java 1.5 RE
GROOVE	Java 1.6 RE	
Henshin	Java, EMF	Eclipse, EMF
MDELab SDI	Java, EMF	Eclipse, EMF
metatools	Java RE	Java SDK, ANTLR, GNU make, coreutils
MOLA	Java, EMF or Windows(C++)	
QVTR-XSLT	XSLT processor	MagicDraw UML
UML-RSDS	Java	Java
Viatra2	Java	Eclipse

[1] ADT abbreviates the ATL development tools.

Table F.21: Prerequisites

Tool	License	Year	Ver.	Code size
Edapt	EPL	2007		36k Java, 31k comment
EMFTVM	EPL	2011[2]	3.3.0	50k Java
Epsilon	EPL	2005	0.9.1	397k Java
GReTL	GPL	2009		28k Java, 19k comment
GrGen.NET	LGPL	2003	3.1	78k Java, 65k C#
GROOVE	Apache	2003	4.6.0	133k Java, 63k comment
Henshin	EPL	2009	EIP	57k Java, 41k comment
MDELab SDI	EPL	2008	2.3.5	104k Java, 49k comment
metatools	closed/money[1]	2001		103k Java
MOLA	closed/free	2005	1.3.5	82k C++
QVTR-XSLT	closed/free	2009		7.5k XSLT, 1.2k Java, 10k XML
UML-RSDS	open/free	1996	1.3	75k Java
Viatra2	EPL	2002	3.2	144k Java, 104k comment

[1] academic licenses available
[2] the values are given for the EMFTVM extension, ATL is older

Table F.22: License and Maturity

money using the tool requires payment.

strong using the tool requires publishing the using code.

weak changing the tool requires publishing the changes.

free none of the above.

Alternatively, the licenses may be specified directly:

Apache the Apache License, of type free (mostly).

GPL the GNU General Public License, of type strong.

LGPL the Lesser GPL, of type weak.

EPL the Eclipse Public License, of type weak.

The column Year specifies when development was started. The column Version defines the version number reached. EIP instead of a version number stands for Eclipse Incubation Project. The column Code gives the number of lines of code written, by programming language, according to cloc.sourceforge.net (excluding third-party libraries and examples).

Appendix F.13. Table F.23

The classification for the support of metamodel or type graph constraints in the Constraints column is combined from two groups. The first is the fact to be checked, being one of:

A: only certain attributes are allowed (depending on the type).

T: only certain edge types are allowed between certain node types.

M: only certain multiplicities of incoming/outgoing edges of a given type are allowed at specific nodes.

The second is the strictness of the check, being one of:

F: this is enforced.

C: this can be checked.

The abbreviation **SSE** in the Validation and Verification column stands for state space enumeration.

Tool	Constraints	Validation and Verification
Edapt	T:C, M:C	
EMFTVM	A:F, T:F, M:C	
Epsilon	A:F, T:F, M:F [1]	dedicated validation language
GReTL	A:F, T:F, M:C	
GrGen.NET	A:F, T:C, M:C	rewrite-clone, programmed SSE
GROOVE	A:F, T:F, M:C+F	SSE for model checking
Henshin	T:C+F, M:C+F	SSE for model checking
MDELab SDI	T:C+F, A:C, M:C	
metatools	A:F, T:F, M:F	
MOLA	A:F, T:F, M:F	
QVTR-XSLT	A:C, T:C	OCL checking
UML-RSDS	A:F, M:F, T:F	correct-by-design, export to theorem-prover
VIATRA2	T:C, M:C	stochastic SSE

[1] depends on the underlying modeling technology, values given for EMF

Table F.23: Validation and verification support

References

[1] Arendt, T., Bierman, E., Jurack, S., Krause, C., Taentzer, G., 2010. HENSHIN: Advanced concepts and tools for in-place EMF model transformations. In: Model Driven Engineering Languages and Systems (MODELS). Vol. 6394 of Lecture Notes in Computer Science. Springer, pp. 121–135.

[2] Barmpis, K., Kolovos, D., 2012. Comparative analysis of data persistence technologies for large-scale models. In: Extreme Modeling Workshop, ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems.

[3] Bedaride, P., Gardent, C., 2009. Semantic normalisation: a framework and an experiment. In: Proceedings of the Eighth International Conference on Computational Semantics. IWCS-8 '09. Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 359–370.

[4] Bergmann, G., Ráth, I., Varró, G., Varró, D., 2012. Change-driven model transformations. change (in) the rule to rule the change. Software and Systems Modeling (SoSyM) 11, 431–461.

[5] Bézivin, J., Rumpe, B., Schürr, A., Tratt, L., 2005. Model transformations in practice workshop.
URL http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf

[6] Biermann, E., Ermel, C., Lambers, L., Prange, U., Runge, O., Taentzer, G., 2010. Introduction to AGG and EMF Tiger by modeling a conference scheduling system. International Journal on Software Tools for Technology Transfer (STTT) 12, 245–261.

[7] Blomer, J., Geiß, R., Jakumeit, E., Jul. 2011. The GrGen.NET user manual.
URL http://www.grgen.net

[8] Börger, E., Stärk, R., 2003. Abstract State Machines. A method for High-Level System Design and Analysis. Springer.

[9] Brambilla, M., Cabot, J., Wimmer, M., 2012. Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers.

[10] Buchwald, S., Jakumeit, E., 2011. Compiler optimization: A case for the transformation tool contest. In: Van Gorp, P., Mazanek, S., Rose, L. (Eds.), Proceedings Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011. Vol. 74 of Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 6–16.

[11] Clowes, D., Kolovos, D. S., Holmes, C., Rose, L. M., Paige, R. F., Johnson, J., Dawson, R., Probets, S. G., 2010. A reflective approach to model-driven web engineering. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (Eds.), Modelling Foundations and Applications (ECMFA). Vol. 6138 of Lecture Notes in Computer Science. Springer, pp. 62–73.

[12] Czarnecki, K., Helsen, S., Jul. 2006. Feature-based survey of model transformation approaches. IBM Systems Journal 45 (3), 621–645.
URL http://dx.doi.org/10.1147/sj.453.0621

[13] Dan, L., 2011. SHARE image of the QVTR XSLT solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_TTC11_QVTR-XSLT.vdi

[14] Ebert, J., Bildhauer, D., 2010. Reverse engineering using graph queries. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (Eds.), Graph Transformations and Model-Driven Engineering. Vol. 5765 of Lecture Notes in Computer Science. Springer, pp. 335–362.

[15] Ebert, J., Horn, T., 2012. GReTL: an extensible, operational, graph-based transformation language. Software and Systems Modeling (SoSyM), 1–21.
URL http://dx.doi.org/10.1007/s10270-012-0250-3

[16] Edgar Jakumeit, S. B., 2011. SHARE image of the GrGen.NET solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_GrGen_v2.vdi

[17] Fischer, T., Niere, J., Torunski, L., Zündorf, A., 2000. Story diagrams: A new graph rewrite language based on the unified modeling language and Java. In: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations (TAGT). Vol. 1764 of Lecture Notes in Computer Science. Springer, pp. 296–309.

[18] Fuss, C., Mosler, C., Ranger, U., Schultchen, E., 2007. The jury is still out: A comparison of AGG, Fujaba, and PROGRES. ECEASST 6.
URL http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/65

[19] Gabrysiak, G., Giese, H., Seibel, A., Sep. 2010. Deriving behavior of multi-user processes from interactive requirements validation. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE'10. ACM, Antwerp, Belgium, pp. 355–356.

[20] Galvao Lourenco da Silva, I., Zambon, E., Rensink, A., Wevers, L., Akşit, M., 2011. Knowledge-based graph exploration analysis. In: Schürr, A., Varró, D., Varró, G. (Eds.), Fourth International Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE). Vol. 7233 of Lecture Notes in Computer Science. Springer, pp. 105–120.

[21] Geiger, L., Zündorf, A., 2010. Fujaba case studies for GraBaTs 2008: lessons learned. International Journal on Software Tools for Technology Transfer (STTT) 12, 287–304.

[22] Geiß, R., Kroll, M., Dec. 2007. On improvements of the Varró benchmark for graph transformation tools. Tech. Rep. 2007-7, Universität Karlsruhe, IPD Goos.
URL http://www.info.uni-karlsruhe.de/papers/TR_2007_7.pdf

[23] Ghamarian, A. H., de Mol, M. J., Rensink, A., Zambon, E., Zimakova, M. V., 2012. Modelling and analysis using GROOVE. International Journal on Software Tools for Technology Transfer (STTT) 14 (1), 15–40.

[24] Giese, H., Hildebrandt, S., Seibel, A., 2009. Improved flexibility and scalability by interpreting story diagrams. In: Magaria, T., Padberg, J., Taentzer, G. (Eds.), Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT). Vol. 18 of Electronic Communications of the EASST.

[25] Gorp, P. V., Mazanek, S., 2011. Share: a web portal for creating and sharing executable research papers. Procedia CS 4, 589–597.

[26] Grasl, T., Economou, A., 2011. Grape: using graph grammars to implement shape grammars. In: Proceedings of the 2011 Symposium on Simulation for Architecture and Urban Design. SimAUD '11. Society for Computer Simulation International, San Diego, CA, USA, pp. 21–28.
URL http://dl.acm.org/citation.cfm?id=2048536.2048539

[27] Grønmo, R., Møller-Pedersen, B., Olsen, G., 2009. Comparison of three model transformation languages. In: Paige, R. F., Hartman, A., Rensink, A. (Eds.), European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA). Vol. 5562 of Lecture Notes in Computer Science. Springer, pp. 2–17.

[28] Heckel, R., Feb. 2006. Graph transformation in a nutshell. Electron. Notes Theor. Comput. Sci. 148 (1), 187–198.
URL http://dx.doi.org/10.1016/j.entcs.2005.12.018

[29] Hegedüs, Á., 2011. SHARE image of the VIATRA2 solution. URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu-11_TTC11_VIATRA.vdi

[30] Hegedüs, Á., Horváth, Á., Ráth, I., Varró, D., Nov. 2011. A model-driven framework for guided design space exploration. In: Alexander, P., Pasareanu, C. S., Hosking, J. G. (Eds.), 26th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, Lawrence, Kansas, USA, pp. 173–182.

[31] Hegedüs, Á., Ujhelyi, Z., Ráth, I., Horváth, Á., 2011. Visualization of traceability models with domain-specific layouting. Electronic Communications of the EASST, Proceedings of the Fourth International Workshop on Graph-Based Tools 32.

[32] Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C., 2009. Derivation and refinement of textual syntax for models. In: Paige, R. F., Hartman, A., Rensink, A. (Eds.), Proceedings of the 5th European Conference on Model Driven Architecture — Foundations and Applications (ECMDA-FA). Vol. 5562 of Lecture Notes in Computer Science. Springer, pp. 114–129.

[33] Helms, B., Shea, K., Hoisl, F., 2009. A framework for computational design synthesis based on graph-grammars and function-behavior-structure. ASME Conference Proceedings 2009 (49057), 841–851.

[34] Herrmannsdoerfer, M., 2011. GMF: A model migration case for the transformation tool contest. In: TTC. pp. 1–5.

[35] Herrmannsdoerfer, M., 2011. SHARE image of the Edapt solution. URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_EMFEdapt.vdi

[36] Herrmannsdoerfer, M., Benz, S., Juergens, E., 2008. Automatability of coupled evolution of metamodels and models in practice. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (Eds.), Model Driven Engineering Languages and Systems (MODELS). Vol. 5301 of Lecture Notes in Computer Science. Springer, pp. 645–659.

[37] Herrmannsdoerfer, M., Benz, S., Juergens, E., 2008. COPE: A language for the coupled evolution of metamodels and models. In: 1st International Workshop on Model Co-Evolution and Consistency Management (MCC).

[38] Herrmannsdoerfer, M., Benz, S., Juergens, E., 2009. COPE - automating coupled evolution of metamodels and models. In: Drossopoulou, S. (Ed.), Object-Oriented Programming, 23rd European Conference (ECOOP). Vol. 5653 of Lecture Notes in Computer Science. Springer, pp. 52–76.

[39] Herrmannsdoerfer, M., Vermolen, S., Wachsmuth, G., 2010. An extensive catalog of operators for the coupled evolution of metamodels and models. In: Malloy, B. A., Staab, S., van den Brand, M. (Eds.), Software Language Engineering (SLE). Vol. 6563 of Lecture Notes in Computer Science. Springer, pp. 163–182.

[40] Horn, T., 2011. Program understanding: A reengineering case for the transformation tool contest. In: Van Gorp, P., Mazanek, S., Rose, L. (Eds.), Proceedings Fifth Transformation Tool Contest (TTC). Vol. 74 of Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 17–21.

[41] Horn, T., 2011. SHARE image of the GReTL solution. URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_gretl-cases.vdi

[42] Horn, T., Ebert, J., 2011. The GReTL transformation language. In: Theory and Practice of Model Transformations - 4th International Conference (ICMT). Vol. 6707 of Lecture Notes in Computer Science. Springer, pp. 183–197.

[43] Horváth, Á., Bergmann, G., Ráth, I., Varró, D., 2010. Experimental assessment of combining pattern matching strategies with VIATRA2. International Journal on Software Tools for Technology Transfer (STTT) 12, 211–230.

[44] Interdisciplinary Centre for Security, Reliability and Trust (SnT), 2011. Annual report 2011.

URL http://www.uni.lu/content/download/52106/624943/version/1/file/SnT_AR2011_final_web.pdf

[45] Jakumeit, E., 2011. ENBF and SDT for GrGen.NET. Tech. rep.
URL http://www.grgen.net

[46] Jakumeit, E., Buchwald, S., Kroll, M., Jul. 2010. GrGen.NET. International Journal on Software Tools for Technology Transfer (STTT) 12 (3), 263–271.
URL http://dx.doi.org/10.1007/s10009-010-0148-8

[47] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Jun. 2008. ATL: A model transformation tool. Science of Computer Programming 72 (1-2), 31–39.

[48] Jurack, S., Tietje, J., 2011. Saying hello world with Henshin - a solution to the TTC 2011 instructive case. In: Van Gorp, P., Mazanek, S., Rose, L. (Eds.), Proceedings Fifth Transformation Tool Contest (TTC). Vol. 74 of Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 253–280.

[49] Kalnina, E., 2011. SHARE image of the Mola solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_MOLA.vdi

[50] Kalnina, E., Kalnins, A., Celms, E., Sostaks, A., 2010. Graphical template language for transformation synthesis. In: van den Brand, M., Gasevic, D., Gray, J. (Eds.), Software Language Engineering (SLE). Vol. 5969 of Lecture Notes in Computer Science. Springer, pp. 244–253.

[51] Kalnins, A., Barzdins, J., Celms, E., 2004. Model transformation language MOLA. In: Aßmann, U., Aksit, M., Rensink, A. (Eds.), Model Driven Architecture: Foundations and Applications (MDAFA). Vol. 3599 of Lecture Notes in Computer Science. Springer, pp. 14–28.

[52] Kalnins, A., Vilitis, O., Celms, E., Kalnina, E., Sostaks, A., Barzdins, J., 2007. Building tools by model transformations in Eclipse. In: Proceedings of DSM'07 Workshop of OOPSLA 2007. Jyvaskyla University Printing House, Montreal, Canada, pp. 194–207.

[53] Kastenberg, H., Rensink, A., 2006. Model checking dynamic states in GROOVE. In: Valmari, A. (Ed.), Model Checking Software (SPIN). Vol. 3925 of Lecture Notes in Computer Science. Springer, pp. 299–305.

[54] Kernighan, B. W., 1988. The C Programming Language, 2nd Edition. Prentice Hall Professional Technical Reference.

[55] Kleppe, A., Warmer, J., Bast, W., 2003. MDA Explained — The Model Driven Architecture: Practice and Promise. Addison-Wesley.

[56] Kolovos, D., Paige, R., Polack, F., 2006. The epsilon object language (EOL). In: Rensink, A., Warmer, J. (Eds.), Model Driven Architecture — Foundations and Applications (ECMDA-FA). Vol. 4066 of Lecture Notes in Computer Science. Springer, pp. 128–142.

[57] Krause, C., 2011. SHARE image of the Henshin solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_Henshin_v2.vdi

[58] Krause, C., Giese, H., 2012. Probabilistic graph transformation systems. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (Eds.), International Conference on Graph Transformation (ICGT). Vol. 7562 of Lecture Notes in Computer Science. Springer, pp. 311–325.

[59] Kurtev, I., Bézivin, J., Aksit, M., 2002. Technological spaces: An initial appraisal. In: CoopIS, DOA Federated Conferences, Industrial track.

[60] Lano, K., Apr. 2008. Constraint-driven development. Inf. Softw. Technol. 50 (5), 406–423.
URL http://dx.doi.org/10.1016/j.infsof.2007.04.003

[61] Lano, K., 2011. Share image of the UML RSDS.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC10_uml-rsds-livecontest_TTC11.vdi

[62] Lano, K., Kolahdouz-Rahimi, S., 2011. Specification of the GMF migration case study. In: Transformation Tool Contest (TTC).

[63] Lano, K., Kolahdouz-Rahimi, S., 2012. Constraint-based specification of model transformations. Journal of Systems and Software.

[64] Lano, K., Kolahdouz-Rahimi, S., Clark, T., 2012. Comparing model transformation verification approaches. In: Modevva workshop, MODELS 2012.

[65] Lepper, M., 2011. SHARE image of the metatools solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_metatools.vdi

[66] Lepper, M., Trancón y Widemann, B., 2011. Optimization of visitor performance by reflection-based analysis. In: Cabot, J., Visser, E. (Eds.), International Conference on Theory and Practice of Model Transformations (ICMT). Vol. 6707 of Lecture Notes in Computer Science. Springer, pp. 15–30.

[67] Lepper, M., Trancón y Widemann, B., 2011. Solving the TTC 2011 compiler optimization task with metatools. In: Gorp, P. V., Mazanek, S., Rose, L. (Eds.), TTC. Vol. 74 of EPTCS. pp. 70–115.

[68] Li, D., Li, X., Stolz, V., Jan. 2011. QVT-based model transformation using XSLT. SIGSOFT Softw. Eng. Notes 36 (1), 1–8.
URL http://dx.doi.org/10.1145/1921532.1921563

[69] Liepiņš, R., 2011. lQuery: A model query and transformation library. Scientific Papers, University of Latvia 770, 27–45.

[70] Mazanek, S., 2011. Helloworld! An instructive case for the transformation tool contest. In: Van Gorp, P., Mazanek, S., Rose, L. (Eds.), Proceedings Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011. Vol. 74 of Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 22–26.
URL http://sites.google.com/site/helloworldcase/

[71] Mazanek, S., Rutetzki, C., Minas, M., 2011. Tool demonstration of the transformation judge. In: Schürr, A., Varró, D., Varró, G. (Eds.), Applications of Graph Transformations with Industrial Relevance (AGTIVE). Vol. 7233 of Lecture Notes in Computer Science. Springer, pp. 97–104.

[72] Mens, T., Gorp, P. V., 2006. A taxonomy of model transformation. Electr. Notes Theor. Comput. Sci. 152, 125–142.

[73] Mens, T., Gorp, P. V., Varró, D., Karsai, G., 2006. Applying a model transformation taxonomy to graph transformation technology. Electr. Notes Theor. Comput. Sci. 152, 143–159.
URL http://dx.doi.org/10.1016/j.entcs.2005.10.022

[74] Mészáros, T., Mezei, G., Levendovszky, T., Asztalos, M., Jul. 2010. Manual and automated performance optimization of model transformation systems. International Journal on Software Tools for Technology Transfer (STTT) 12 (3-4), 231–243.

[75] Miller, J., Mukerji, J. (Eds.), 2003. MDA Guide. Object Management Group.

[76] Moha, N., Sen, S., Faucher, C., Barais, O., Jézéquel, J.-M., 2010. Evaluation of Kermeta for solving graph-based problems. International Journal on Software Tools for Technology Transfer (STTT) 12, 273–285.

[77] Muliawan, O., Janssens, D., 2010. Model refactoring using MoT-MoT. International Journal on Software Tools for Technology Transfer (STTT) 12, 201–209.

[78] Pérez, J., Crespo, Y., Hoffmann, B., Mens, T., 2010. A case study to evaluate the suitability of graph transformation tools for program refactoring. International Journal on Software Tools for Technology Transfer (STTT) 12, 183–199.

[79] Ráth, I., Bergmann, G., Ökrös, A., Varró, D., 2008. Live model transformations driven by incremental pattern matching. In: Vallecillo, A., Gray, J., Pierantonio, A. (Eds.), Proc. First International Conference on the Theory and Practice of Model Transformations (ICMT). Vol. 5063/2008 of Lecture Notes in Computer Science. Springer, pp. 107–121.

[80] Ráth, I., Vágó, D., Varró, D., 2008. Design-time simulation of domain-specific models by incremental pattern matching. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008,

Herrsching am Ammersee, Germany, 15-19 September 2008, Proceedings. IEEE, pp. 219–222.

[81] Rensink, A., 2011. SHARE image of the GROOVE solution. URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_groove-helloworld.vdi

[82] Rensink, A., Heckel, R., König, B., Jul. 2007. Graph transformation for concurrency and verification - preface. In: Proceedings of the Workshop on Graph Transformation for Concurrency and Verification (GT-VC). Vol. 175 of Electronic Notes in Theoretical Computer Science 175. Elsevier, Amsterdam, pp. 1–2.
URL http://doc.utwente.nl/61779/

[83] Rensink, A., Van Gorp, P., 2010. Graph transformation tool contest 2008. International Journal on Software Tools for Technology Transfer (STTT) 12, 171–181.

[84] Rose, L., 2011. SHARE image of the Epsilon solution. URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_epsilon-helloworld.vdi

[85] Rose, L., Kolovos, D., Paige, R., Polack, F., 2010. Model migration with Epsilon Flock. In: Tratt, L., Gogolla, M. (Eds.), Theory and Practice of Model Transformations, Third International Conference (ICMT). Vol. 6142 of Lecture Notes in Computer Science. Springer, pp. 184–198.

[86] Rose, L. M., Herrmannsdörfer, M., Mazanek, S., Gorp, P. V., Buchwald, S., Horn, T., Kalnina, E., Koch, A., Lano, K., Schätz, B., Wimmer, M., 2012. Graph and model transformation tools for model migration. Software and Systems Modeling (SoSyM).

[87] Rose, L. M., Paige, R. F., Kolovos, D. S., Polack, F., 2008. The epsilon generation language. In: Schieferdecker, I., Hartman, A. (Eds.), Model Driven Architecture - Foundations and Applications (ECMDA-FA). Vol. 5095 of Lecture Notes in Computer Science. Springer, pp. 1–16.

[88] Schimmel, J., Gelhausen, T., Schaefer, C. A., 2009. Gene expression with general purpose graph rewriting systems. In: Proceedings of the 8th GT-VMT Workshop. Vol. 18 of Electronic Communications of the EASST.
URL /Tichy/uploads/publikationen/207/GT-VMT09.pdf

[89] Schösser, A., Geiß, R., 2008. Graph rewriting for hardware dependent program optimizations. In: Schürr, A., Nagl, M., Zündorf, A. (Eds.), Applications of Graph Transformation with Industrial Relevance (AGTIVE). Vol. 5088 of Lecture Notes in Computer Science. Springer, pp. 233–248.

[90] Seidewitz, E., Sep. 2003. What models mean. IEEE Softw. 20 (5), 26–32.
URL http://dx.doi.org/10.1109/MS.2003.1231147

[91] Śmiałek, M., Kalnins, A., Kalnina, E., Ambroziewicz, A., Straszak, T., Wolter, K., 2010. Comprehensive system for systematic case-driven software reuse. In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorn'y, J., Rumpe, B. (Eds.), SOFSEM 2010: Theory and Practice of Computer Science. Vol. 5901 of Lecture Notes in Computer Science. Springer, pp. 697–708.

[92] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2009. EMF: Eclipse Modeling Framework 2.0, 2nd Edition. Addison-Wesley Professional.

[93] Taentzer, G., 2004. AGG: A graph transformation environment for modeling and validation of software. In: Pfaltz, J. L., Nagl, M., Böhlen, B. (Eds.), Applications of Graph Transformations with Industrial Relevance (AGTIVE). Vol. 3062 of Lecture Notes in Computer Science. Springer, pp. 446–453.

[94] Taentzer, G., Biermann, E., Bisztray, D., Bohnet, B., Boneva, I., Boronat, A., Geiger, L., Geiß, R., Horvath, Á., Kniemeyer, O., Mens, T., Ness, B., Plump, D., Vajk, T., 2007. Generation of Sierpinski triangles: A case study for graph transformation tools. In: AGTIVE. pp. 514–539.

[95] Taentzer, G., Ehrig, K., Guerra, E., Lara, J. D., Levendovszky, T., Prange, U., Varró, D., 2005. Model transformations by graph transformations: A comparative study. In: Model Transformations in Practice Workshop at MODELS 2005, Montego. p. 05.

[96] Trancón y Widemann, B., Lepper, M., 2012. Paisley: Pattern matching à la carte. In: Hu, Z., de Lara, J. (Eds.), Theory and Practice of Model Transformations — 5th International Conference (ICMT). Vol. 7307 of Lecture Notes in Computer Science. Springer, pp. 240–247.

[97] Trancón y Widemann, B., Lepper, M., Wieland, J., 2003. Automatic construction of XML-based tools seen as meta-programming. Automated Software Engineering 10, 23–38.

[98] Ujhelyi, Z., Horváth, Á., Varró, D., 2012. Dynamic backward slicing of model transformations. In: Antoniol, G., Bertolino, A., Labiche, Y. (Eds.), International Conference on Software Testing and Validation (ICST). IEEE, pp. 1–10.

[99] Van Gorp, P., Mazanek, S., Rose, L. (Eds.), 2011. TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011, Post-Proceedings. EPTCS.

[100] Varró, D., Asztalos, M., Bisztray, D., Boronat, A., Dang, D.-H., Geiß, R., Greenyer, J., Van Gorp, P., Kniemeyer, O., Narayanan, A., Rencis, E., Weinell, E., 2008. Transformation of UML models to CSP: A case study for graph transformation tools. In: Schürr, A., Nagl, M., Zündorf, A. (Eds.), International Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE). Vol. 5088 of Lecture Notes in Computer Science. Springer, pp. 540–565.

[101] Varró, D., Balogh, A., Oct. 2007. The model transformation language of the VIATRA2 framework. Science of Computer Programming 68 (3), 214–234.

[102] Varró, D., Pataricza, A., Oct. 2003. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Software and Systems Modeling (SoSyM) 2 (3), 187–210.

[103] Varró, G., Schürr, A., Varró, D., 2005. Benchmarking for graph transformation. In: VL/HCC. pp. 79–88.

[104] VIATRA2 Framework: An Eclipse GMT Subproject, 2012.
URL http://viatra.inf.mit.bme.hu

[105] von Detten, M., Heinzemann, C., Platenius, M., Rieke, J., Travkin, D., Hildebrandt, S., 2012. Story diagrams - syntax and semantics. Tech. rep., Software Engineering Group, Heinz Nixdorf Institute.

[106] Wagelaar, D., 2011. SHARE image of the EMFTVM solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu-11_TTC11_livecontest-dennis_EMFTVM-HelloWorld.vdi

[107] Wagelaar, D., Iovino, L., Di Ruscio, D., Pierantonio, A., 2012. Translational semantics of a co-evolution specific language with the EMF transformation virtual machine. In: Hu, Z., de Lara, J. (Eds.), Theory and Practice of Model Transformations - 5th International Conference (ICMT). Vol. 7307 of Lecture Notes in Computer Science. Springer, pp. 192–207.

[108] Wagelaar, D., Tisi, M., Cabot, J., Jouault, F., 2011. Towards a general composition semantics for rule-based model transformation. In: Whittle, J., Clark, T., Kühne, T. (Eds.), Model Driven Engineering Languages and Systems (MODELS). Vol. 6981 of Lecture Notes in Computer Science. Springer, pp. 623–637.

[109] Warmer, J., Kleppe, A., 2003. The Object Constraint Language: Getting Your Models Ready for MDA, 2nd Edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[110] Wätzold, S., 2011. SHARE image of the MDE Lab solution.
URL http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_sdm-hpi.vdi