Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Database and Information Systems
Fürstenallee 11, 33102 Paderborn

Incremental Unidirectional Model Transformation via Graph Transformation with eMoflon::IBeX

Master's Thesis

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Patrick Robrecht

submitted to

Jun.-Prof. Dr. Anthony Anjorin

and

Prof. Dr. Gregor Engels

Paderborn, July 12, 2018

Master's Thesis by Patrick Robrecht
Course of studies: Computer Science

Supervisor: Jun.-Prof. Dr. Anthony Anjorin

First examiner: Jun.-Prof. Dr. Anthony Anjorin
Second examiner: Prof. Dr. Gregor Engels

Date of submission: July 12, 2018

Abstract

Model transformations are used to transform models into other models. They are fundamental for model-driven engineering (MDE) which raises the abstraction level from programming to domain-specific languages. As models can be suitably represented as graphs, graph transformation (GT) is a frequently used formalism to realize model transformations. The approach is based on a rule set defining which graph patterns can be replaced with other graph structures.

This thesis presents eMoflon::IBeX-GT, a new interpreter-based graph transformation tool which supports model queries (i. e. finding patterns in the graph) as well as rule applications modifying graph structures. An incremental pattern matcher is used to find matches in the host graph. The support for incrementality is important to solve tasks which require permanent observation of all matches efficiently.

The graph transformation patterns and rules can be invoked from Java code via a typed API generated from the textual specification. Pattern refinement is a modularity concept to reduce duplications in pattern specifications.

There are many GT tools, but to the best of the author's knowledge none of them supports incrementality in combination with model queries and rule applications on attributed typed graphs via a typed API.

This thesis explores which tasks can be solved best via an incremental graph transformation tool and how the tool can be seamlessly integrated into Java code.

Contents

Declaration of Authorship

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Paderborn, July 12, 2018

Patrick Robrecht

1 Introduction

Model-driven engineering (MDE) is an approach to software development which focuses on the specification of software artifacts in the application domain. It raises the abstraction level from programming to domain-specific modeling languages by the use of *models* for the representation of knowledge. The MDE approach aims for a simplification of the specification process and improvement of the communication between different people and teams working on a system (cf. [SV06], p. 13; [HMS05]).

Model transformations [CH06] can be used to transform models into other models, e. g. modifying an existing model or generating code from a model. Other application areas are web applications [Koz16], handling XML documents [Kur05], interface design, and many others [CFH+08]. Because of the focus on models as primary artifacts, model transformations are a fundamental part of MDE.

1.1 Graphs and Graph Transformations

Since many models can be represented as graphs (e. g. UML[1] or XML documents), *graph transformations* (GT) are a frequently used formalism to implement model transformations. They consist of a set of *graph transformation rules* of the form $L \to R$. The left-hand side L (called pattern graph) specifies the context in which the rule may be applied to a given host graph, while the right-hand side R defines the elements with which the context is replaced during rule application.[2]

Using this declarative approach for the specification of model transformation, a rule engine can be used to find a match in the host graph conforming to the pattern graph L and apply the rule.

1.2 eMoflon::IBeX

The graph transformation tool we[3] shall work with in this thesis is the Eclipse-based tool *eMoflon*, which is currently being reimplemented in the *eMoflon::IBeX* project.[4] Unlike the code generation based *eMoflon::SDM/TiE*,[5] eMoflon::IBeX uses an interpreter which is completely based on incremental pattern matching.

[1] cp. UML 2.5 [Obj15], Annex E

[2] Formal foundations as needed for this thesis will be introduced in Chapter 2.

[3] Despite the use of the authorial we in this thesis, it has been written by its single author, as stated in the declaration of authorship. The authorial we serves to increase the readability of the thesis.

[4] see `https://github.com/eMoflon/emoflon-ibex`

[5] see `https://github.com/eMoflon/emoflon-tool`.
eMoflon::SDM refers to the story driven modeling part (graph transformations and control flow) of the latest eMoflon release using Enterprise Architect and code generation.
eMoflon::TiE (Tool Integration Environment) allows to specify TGG rules in textual syntax. It relies on code generation and a transformation to SDMs.

eMoflon::TiE and eMoflon::IBeX provide a textual editor for transformation rules with syntax highlighting and checks for compliance to the meta-models specified in Ecore, the UML-like meta-modeling language of the *Eclipse Modeling Framework* (EMF). A graphical visualization of specified rules generated from the textual syntax is provided for better understandability.

While eMoflon::SDM/TiE supports both unidirectional graph transformations and bidirectional transformations between two different meta-models based on *Triple Graph Grammars* (TGGs)[6] as an underlying formalism, only the transformations with TGGs have been ported to eMoflon::IBeX (herein after referred to as *eMoflon::IBeX-TGG*). The goal of this thesis is to extend eMoflon::IBeX by unidirectional model transformations modifying a model instance by applying graph transformation rules.

1.3 Running Example: She Remembered Caterpillars

The running example used in this thesis is based on a simplified version of the game *She Remembered Caterpillars*.[7] The use of graph transformation for this game has been suggested and applied by Zündorf et al. [PGH+16]. An example instance of this game is shown in Figure 1.1.[8] The meta-model is shown in Figure 1.2 as a class diagram.

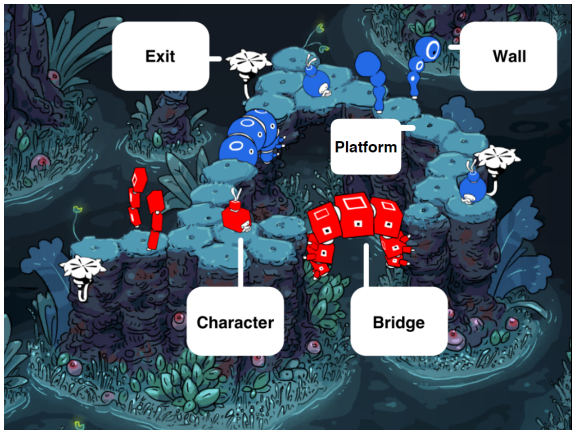


Figure 1.1: She Remembered Caterpillars Example

The She Remembered Caterpillars world contains simple and exit platforms which are connected to neighboring platforms. Simple platforms can be connected via bridges and walls as well.

The goal of the game is to move all characters to exit platforms. At the beginning, characters are placed on arbitrary platforms. Only one character may stand on each exit platform at the end.

[6] TGGs [Sch95] are a grammar-based approach to specify a consistency relation between two models. From the specification different operationalizations such as model generation, model synchronization, and checking the consistency of existing models can be derived.

[7] `http://caterpillar.solutions/` or `http://store.steampowered.com/app/470780/She_Remembered_Caterpillars/`. The rules describe only a limited subset of the actual game.

[8] Figure 1.1 is taken from the slides of the lecture "Fundamentals of Model-Driven Engineering" by Anthony Anjorin, whose examples are partly inspired by [PGH+16].

Characters, bridges and walls are colored blue, red or purple. The following rules define how characters can be moved:

1. All characters can walk from their current platform to any neighboring one.
2. Blue and red characters can cross bridges having the same color as the character and walk over walls which are not colored in the character color.
3. Purple characters can cross any bridge, but cannot walk through any wall.
4. If a red and a blue character stand on the same platform, they can be transformed into one purple character.
5. A purple character can be transformed into a blue and a red character, both standing on the same platform after the transformation.

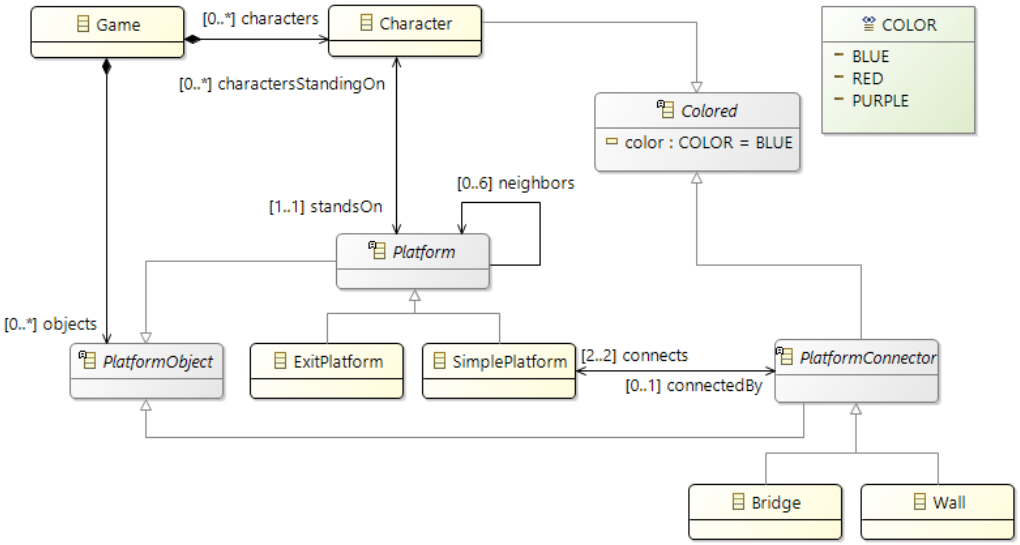


Figure 1.2: She Remembered Caterpillars Class Diagram

All examples in the following chapters will use the She Remembered Caterpillars metamodel. We will check constraints on models, create new models and write transformation rules for the different steps of the game.

1.4 Contribution

The goal of this thesis is to implement and evaluate a new tool for unidirectional graph transformations called *eMoflon::IBeX-GT*, which is based on an interpreter and incremental pattern matching. The rules can be invoked from Java code via a typed Java API generated from the textual rule specification.

eMoflon::IBeX-GT shall support simple patterns, rules, attribute assignments and conditions, pattern refinement, and application conditions. For each of the mentioned features the main steps for the implementation are:

1. the development of a textual editor for pattern and rule specifications and a graphical visualization of them,
2. the definition of an IBeX pattern model, i. e. a pattern network specification independent from a concrete pattern matching engine,
3. the implementation of the transformation from the editor specification into the IBeX pattern model (and from the IBeX pattern model into the Democles representation),
4. the design and the generation a typed Java API,
5. the extension of the interpreter used for API execution,
6. establishing a JUnit test suite to show the successful usage for a set of examples,
7. and providing documentation for end-users.

As we plan to integrate the new tool into the eMoflon::IBeX toolsuite, we want to explore and exploit the potential of Democles, the underlying incremental pattern matcher used in eMoflon::IBeX-TGG already. The following research questions shall be investigated in this thesis:

RQ 1 Which tasks are best solved using an incremental pattern matcher?

RQ 2 How can a GT and TGG tool be seamlessly integrated for developers and end users?

RQ 3 How can we integrate graph transformations seamlessly into Java code using the power of Java 8 features such as streams?

1.5 Structure of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 introduces the theory on graphs, graph transformations rules and their application. Our requirements specified in Chapter 3 will be evaluated for existing graph transformation tools in Chapter 4. Chapter 5 deals with the transformation from the specification of patterns in the editor into pattern networks. The generation of a typed API and the invocation from Java code via the API is presented in Chapter 6. Chapter 7 evaluates the new tool with respect to the requirements, correctness, and performance. Finally, we discuss future work in Chapter 8.

2 Fundamentals of Graph Transformations

This chapter introduces the formal definitions for graph transformations based on category theory. The definitions are based on Ehrig et al. [EEPT06, pp. 21-47 and pp. 65-71] in a slightly different notation used in the lecture "Fundamentals of Model-Driven Engineering" by Anthony Anjorin.

2.1 Typed Graphs

Our models are attributed typed graphs. Typed graphs are formally introduced in the following, while attributes are not formalized in this thesis. The interested reader is referred to [HP16] or [AVS12] for a formalization of attributed graphs.

Definition 1 (*Category*)

A category $\mathbf{C} = (Ob, Arr, \mathbin{;}, id)$ consists of:

- a class Ob of *objects*,
- for each pair of objects $A, B \in Ob$, a class $Arr_{(A,B)}$ of *arrows*, where $f \in Arr_{(A,B)}$ is denoted by $f : A \to B$,
- for all objects $A, B, C \in Ob$, a binary operation (for composing arrows): $\mathbin{;} : Arr_{(A,B)} \times Arr_{(B,C)} \to Arr_{(A,C)}$,
- for each object $A \in Ob$, an identity arrow $id_A : A \to A$,

such that the following conditions hold:

- Associativity.
  $\forall A, B, C, D \in Ob.\ \forall f : A \to B, g : B \to C, h : C \to D.\ f \mathbin{;} (g \mathbin{;} h) = (f \mathbin{;} g) \mathbin{;} h.$
- Identity.
  $\forall A, B \in Ob.\ \forall f : A \to B.\ (id_A \mathbin{;} f = f) \wedge (f \mathbin{;} id_B = f).$

In general, a category is a mathematical structure which has objects and morphisms. A monomorphic arrow corresponds to an injective function.

Definition 2 (*Monomorphism*)

Let $\mathbf{C} = (Ob, Arr, \mathbin{;}, id)$ be a category.

An arrow $f : X \to Y$ is monomorphic, iff

$$\forall\, g : Z \to X\ \forall\, h : Z \to X\ [(h; f = g; f) \Rightarrow (g = h)].$$

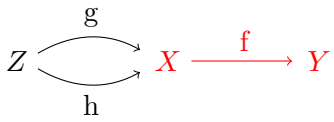


Figure 2.1: Monomorphism

Definition 3 (*Graphs and Graph Morphisms*)

A *graph* $G = (V, E, src, trg)$ consists of a set V of nodes (vertices), a set E of edges, and two functions $src, trg : E \to V$ that assign each edge a source node and a target node, respectively.

Given graphs $G = (V, E, src, trg)$ and $G' = (V', E', src', trg')$, a *graph morphism* $f : G \to G'$ consists of two functions $f_V : V \to V'$ and $f_E : E \to E'$ such that $src ; f_V = f_E ; src'$ and $trg ; f_V = f_E ; trg'$.

Graphs $= (Ob_{Graphs}, Arr_{Graphs}, ;_{Graphs}, id_{Graphs})$ consists of:

- graphs Ob_{Graphs},
- graph morphisms Arr_{Graphs},
- for $G, G', G'' \in Ob_{Graphs}$, $f = G \to G'$, $g = G' \to G'' \in Arr_{Graphs}$, $;_{Graphs}(f, g)$ is defined as $f ;_{Graphs} g := (f_V ; g_V, f_E ; g_E)$,
- for $G = (V, E, src, trg) \in Ob_{Graphs}$, $id_G : G \to G$ is defined as $id_G := (id_V, id_E)$.

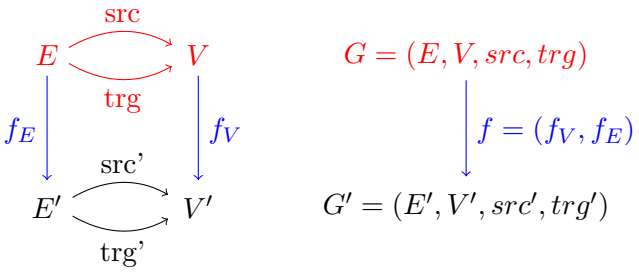


Figure 2.2: Graph Morphism

Definition 4 (*Typed Graphs and Typed Graph Morphisms*)

A *type graph* is a distinguished graph $TG = (V_{TG}, E_{TG}, src_{TG}, trg_{TG})$.

A *typed graph* is a pair $\hat{G} = (G, type)$ of a graph G together with a graph morphism $type : G \to TG$.

Given typed graphs $\hat{G} = (G, type)$ and $\hat{G}' = (G', type')$, a *typed graph morphism* $f : \hat{G} \to \hat{G}'$ is a graph morphism $f : G \to G'$ such that $f ; type' = type$.

TGraphs $= (Ob_{TGraphs}, Arr_{TGraphs}, ;_{TGraphs}, id_{TGraphs})$ consists of:

- typed graphs $Ob_{TGraphs}$,

- typed graph morphisms $Arr_{TGraphs}$,
- $;_{TGraphs} := ;_{Graphs}$,
- $id_{TGraphs} := id_{Graphs}$.

Note that **Graphs** and **TGraphs** are categories according to Definition 1.

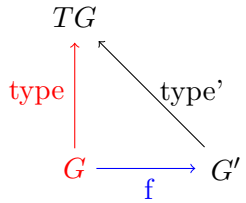


Figure 2.3: Typed Graph Morphism

Figure 2.4 shows the type graph for the She Remembered Caterpillars meta-model introduced in Section 1.3. For clarity only the types used in the following examples are included. A typed graph conforming to this type graph is shown in Figure 2.5. Each node is denoted by its name and its type, separated by a colon. The edge labels are omitted in the graphs as they are comprehensible without ambiguity based on the types of their source and target node.

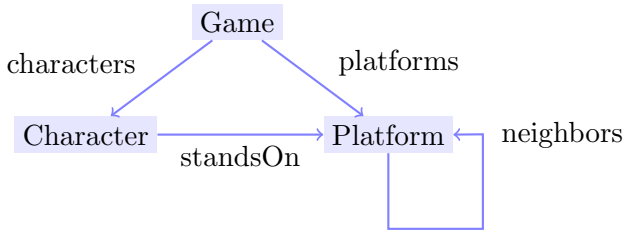


Figure 2.4: Type Graph for She Remembered Caterpillars (simplified)

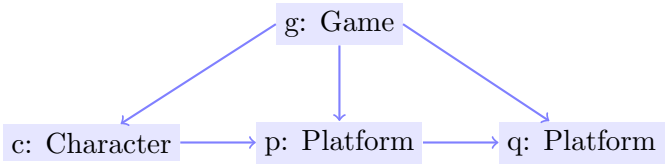


Figure 2.5: Typed Graph Instance

2.2 Rule Applications

A set of rules defines which model modifications are allowed in the graph transformation system. The semantics of rule applications is given by the following definitions.

Definition 5 (*Pushouts*)

Let $\mathbf{C} = (Ob, Arr, ;, id)$ be a category.

Given arrows $r \in Arr : L \to R$ and $m \in Arr : L \to G$, a *pushout* (G', r', m') over r and m is defined by a *pushout object* $G' \in Ob$, and morphisms $r' : G \to G'$, $m' : R \to G'$, where

1. the "pushout square" (Figure 2.6) commutes, i. e. $r \,;\, m' = m \,;\, r'$,
2. and the following universal property is fulfilled:
   $\forall\, G'' \in Ob\ \forall\, r'' : G \to G''\ \forall\, m'' : R \to G''$
   $[(r \,;\, m'' = m \,;\, r'') \Rightarrow (\exists!\, x : G' \to G''\ [(m' \,;\, x = m'') \wedge (r' \,;\, x = r'')])]$.

The category $\mathbf{C}$ is said to *have pushouts* if the pushout (G', r', m') always exists.

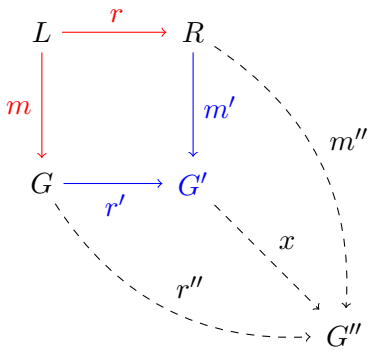


Figure 2.6: Pushout Diagram

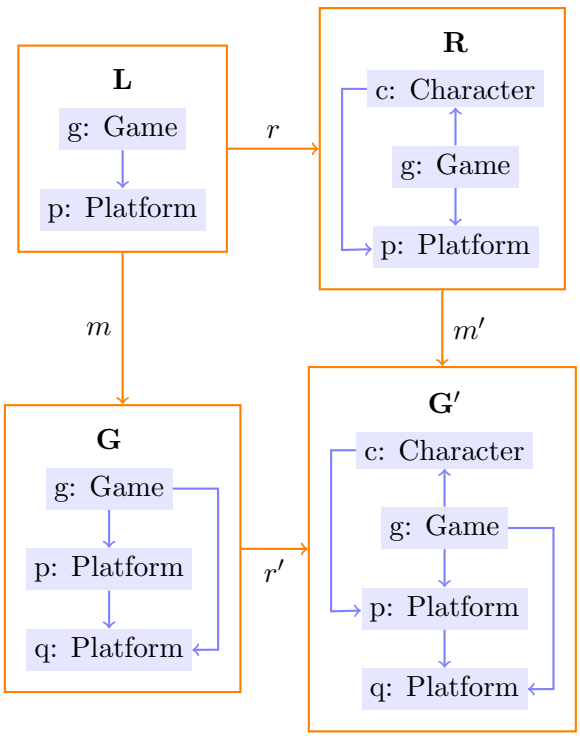


Figure 2.7: Application of the Monotonic Rule `createCharacter`

An application of a monotonic rule $r : L \to R$ requires finding a match in the host graph G (i. e. a monomorphic arrow $m : L \to G$) and constructing the resulting host graph G' by applying the changes (i. e. constructing the monomorphic arrow $m' : R \to G'$) such that the constraints above are fulfilled. m' is called *co-match*. The universal property ensures that no elements are glued unnecessarily, while the commutation in the pushout square rejects all results G' that do not glue enough.

Figure 2.7 illustrates the pushout construction for a rule `createCharacter` which creates a new character and connects the character to a platform and the game.

Definition 6 (*Graph Transformation Rules*)

A *(typed) graph transformation rule* $L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$ is a pair of monomorphic arrows in the category of (**T**)**Graphs** with common "gluing graph" K.

Monotonic rules can only handle creation, but not deletion of elements. That is why the formalism $L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$ is introduced in Definition 6: $l : K \hookrightarrow L$ describes the deletion, $r : K \hookrightarrow R$ the creation of elements. The elements in $L \setminus K$ are deleted by the rule, the elements in $R \setminus K$ are created. All elements in K remain unchanged.

In the following chapters of this thesis, graph transformation rules which do not have deleted or created elements (i. e. $L \setminus K = \emptyset = R \setminus K$) are referred to as *graph transformation patterns*.

Definition 7 (*Double-Pushout Graph Transformation Rule Application*)

Given a (typed) graph transformation rule $p = L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$, a *direct derivation* with p at a monomorphism $m : L \hookrightarrow G$, denoted by $G \overset{p@m}{\Longrightarrow} G'$ (or $G \overset{p}{\Rightarrow} G'$, or $G \Rightarrow G'$), is given by the double-pushout (DPO) diagram below, where (1) and (2) are pushouts in the category **(T)Graphs**.

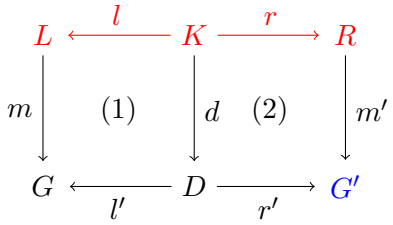


Figure 2.8: Double-Pushout Diagram

Definition 8 (*Applicability of Graph Transformation Rules*)

A (typed) graph transformation rule $p = L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$ is applicable to a (typed) graph G via the match $m : L \hookrightarrow G$, if the pushout complement D in the diagram below exists, such that (1) is a pushout in the category **(T)Graphs**.

For the deletion part, taking the host graph as pushout object and completing square (1) leads to the pushout complement D. After that the pushout is created in square (2) to obtain the resulting graph G'.

Figure 2.9 shows an application of the rule `moveCharacter` which deletes the edge between a character and its platform and creates a new edge between the character and another platform which must be neighboring to the previous one. The construction of the pushout complement D leads to the deletion of the edge between the character c and the platform p (so the rule is applicable according to Definition 8), while the pushout construction creates the new edge between the character c and the platform q, resulting in the graph G'.

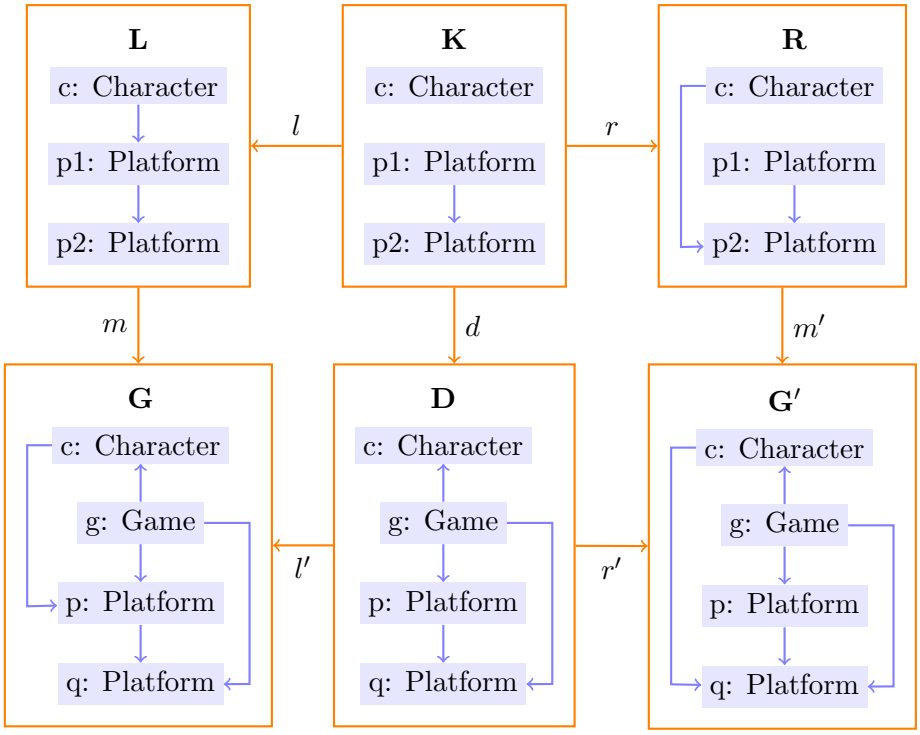


Figure 2.9: Application of the Graph Transformation Rule `moveCharacter`

Definition 9 (*Dangling Edge Condition*)

Given a (typed) graph transformation rule $p = L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$ and a match $m : L \hookrightarrow G$, the set of *dangling points* (DP) is defined as:

$$DP = \{v \in V_L \mid \exists e \in (E_G \setminus m_E(E_L)) \; [\, (s_G(e) = m_V(v)) \vee (t_G(e) = m_V(v)) \,]\}$$

The set of *gluing points* (GP) is defined as $GP = l_V(V_K)$.

The *Dangling Edge Condition* is fulfilled by p and m iff $DP \subseteq GP$.

Theorem 10 (*Existence and Uniqueness of Pushout Complements*)

Given a (typed) graph transformation rule $p = L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$ and a match $m : L \hookrightarrow G$, the pushout complement $\{D, d : K \hookrightarrow D, l' : D \hookrightarrow G\}$ of l and m exists and is unique up to isomorphism iff the dangling edge condition is fulfilled.

Dangling edges are edges whose source or target node is deleted during the application, without the edge itself is defined as a deleted edge by $l : K \hookrightarrow L$. Such edges must not be left over after rule application, as they lead to an invalid graph – that is why an approach to avoid dangling edges after rule application is necessary. Theorem 10 states that a rule is applicable according to DPO if and only if there are no dangling edges.[1] So dangling edges cannot be left over after rule application, as their existence prevents rule application.

Besides DPO rule application according to Definition 7, another pushout approach called *single-pushout* (SPO) exists.[2] SPO relaxes the strict precondition for rule application such that there may be dangling edges: If there are dangling edges, the rule is still applicable and the dangling edges are deleted by the rule application.

2.3 Application Conditions

Application conditions can be used to define additional constraints for rule applications. A rule with an application condition can only be applied if the rule is applicable according to the pushout approach (DPO as defined in Definition 8 or SPO) and the application conditions are fulfilled.

Definition 11 (*Graph Condition*)

Let $\mathbf{C} = (Ob, Arr, \mathbf{;}, id)$ be a category.

A *graph condition* over an object L is a pair $gc = (p : L \to P, \{c_i : P \to C_i \mid i \in I\})$, for some index set I.

$L \in Ob$ is referred to as the *context*, $p \in Arr$ the *premise*, and $\{c_i \in Arr \mid i \in I\}$ the *conclusions* of the graph condition gc.

Definition 12 (*Satisfaction of Graph Conditions*)

Let $\mathbf{C} = (Ob, Arr, \mathbf{;}, id)$ be a category, and gc a graph condition over $L \in Ob$ for some index set I, i.e., $gc = (p : L \to P, \{c_i : P \to C_i \mid i \in I\})$.

An arrow $m : L \to G$ *satisfies* gc, denoted by $m \models gc$, iff

$$\forall\, m_p : P \to G\ [(m = p \,\mathbf{;}\, m_p) \Rightarrow (\exists\, i \in I\, \exists\, m_{c_i} : C_i \to G\ [m_p = c_i \,\mathbf{;}\, m_{c_i}])],$$

where $m_p, (m_{c_i})_{i \in I}$ are monomorphisms.

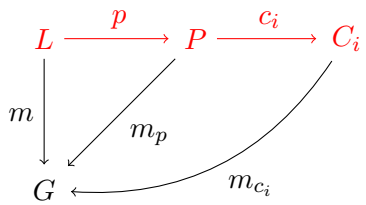


Figure 2.10: Graph Condition

[1] see [EEPT06, Fact 3.11, pp. 45-46] for a proof of Theorem 10
[2] see [EHK+97] for a formal definition of SPO

There are two possibilities to satisfy a graph condition: If there does not exist a monomorphic arrow $m_p : P \to G$, the condition is trivially satisfied (because the premise does not hold). In addition, the graph condition is satisfied if there is at least one $i \in I$ for which a monomorphic arrow $m_{c_i} : C_i \to G$ exists (one of the conclusion is fulfilled).

Graph conditions can be combined via logical expressions using negation ($\neg$), conjunction ($\wedge$) and disjunction ($\vee$) as defined in Definition 13.

Definition 13 (*Complex Graph Conditions*)

$true$ is a complex graph condition.

A graph condition gc is a complex graph condition.

$\neg c$ is a complex graph condition, where c is a complex graph condition.

$\wedge_{j \in J} c_j$ is a complex graph condition, where J is an index set and $(c_j)_{j \in J}$ are complex graph conditions.

$false$ is an abbreviation for $\neg true$.

$\vee_{j \in J} c_j$ is an abbreviation for $\neg(\wedge_{j \in J} \neg c_j)$.

$c \Rightarrow d$ is an abbreviation for $\neg c \vee d$.

Definition 14 (*Satisfaction of Complex Graph Conditions*)

Let $\mathbf{C} = (Ob, Arr, \mathbf{;}, id)$ be a category, and c a complex graph condition over $L \in Ob$.

An arrow $m \in Arr : L \to G$ *satisfies* the complex graph condition c, denoted by $m \models c$, iff one of the following holds:

- $c = true$
- $c = gc$, gc is a graph condition, and $m \models gc$
- $c = \neg c'$, and $m \not\models c'$ (m does not satisfy c')
- $c = \wedge_{j \in J} c_j$, and $\forall_{j \in J} [m \models c_j]$.

Given a rule $r : L \to R$, a graph condition using the left-hand side of the rule as context forms an application condition. A rule is only applicable for a match $m : L \to G$ if the application conditions of the rule are satisfied.

Definition 15 (*Application Conditions*)

Let $\mathbf{C} = (Ob, Arr, \mathbf{;}, id)$ be a category with pushouts.

Given a monotonic rule $r \in Arr : L \to R$, an *application condition* ac for r is a graph condition $(p : L \to P, \{c_i : P \to C_i \mid i \in I\})$ over L.

A direct derivation $d = G \overset{r@m}{\Longrightarrow} G'$ with r at match $m : L \to G$ satisfies ac, denoted by $d \models ac$, iff $m \models ac$ according to Definitions 12 and 14.

Definition 16 (*Negative Application Conditions (NAC)*)

Let $\mathbf{C} = (Ob, Arr, \mathbf{;}, id)$ be a category with pushouts.

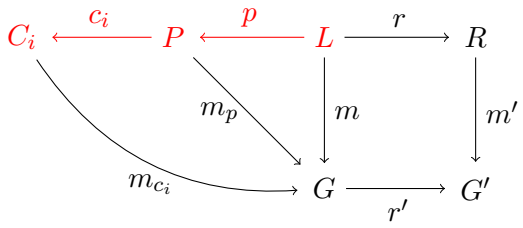


Figure 2.11: Application Condition for Monotonic Rules

Given a monotonic rule $r \in Arr : L \to R$, a *negative application condition (NAC)* for r is an application condition for r of the form $(n : L \to N, \{\})$.

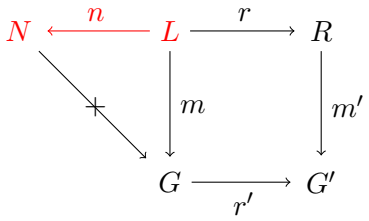


Figure 2.12: Negative Application Condition for Monotonic Rules

A NAC is violated if and only if a premise N exists as there are no conclusions, i. e. the NAC is satisfied if there is no N such that the arrow $N \to G$ can be constructed.

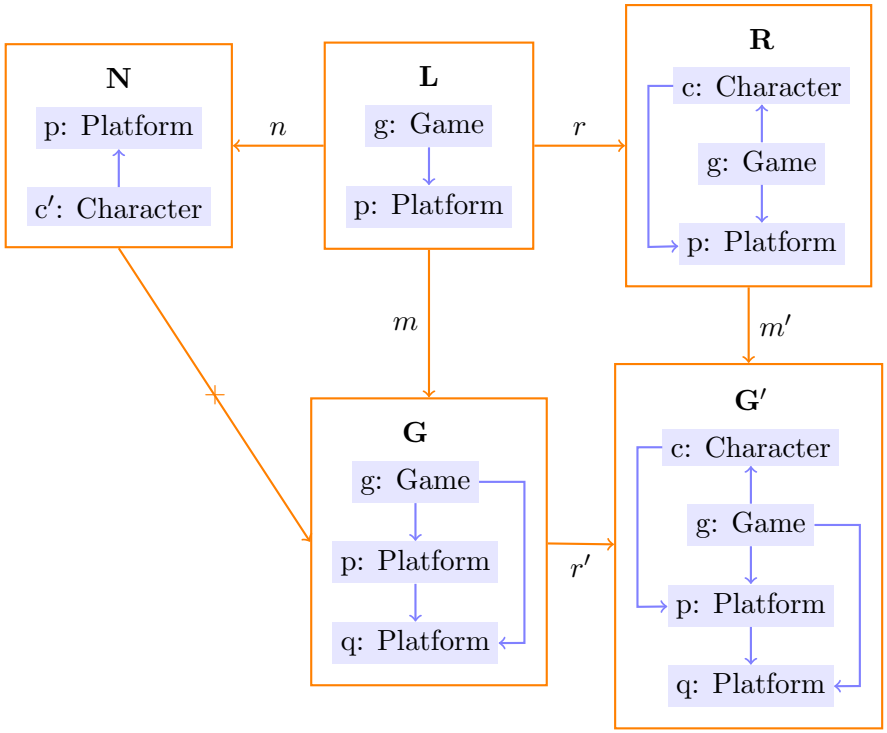


Figure 2.13: Negative Application Condition for the Monotonic Rule `createCharacter`

Figure 2.13 shows an example NAC for the rule `createCharacter`. The NAC ensures that there is no other character which stands already on the platform the new character is placed on. As the arrow $N \to G$ cannot be constructed, the rule is still applicable. Note that a second rule application with the same match m is not possible after the shown application to G because the arrow $N \to G'$ (G' from the first rule application is the new G for the next rule application) can be constructed for the match choosing the platform p. A new character could still be added if m chooses platform q, i. e. a the new character stands on q.

The definition of application conditions for monotonic rules $r : L \to R$ can be easily extended to graph transformation rules $p = L \hookleftarrow K \hookrightarrow R$ as only rule applicability is affected by the definition. Figures 2.14 and 2.15 show the modified diagrams.

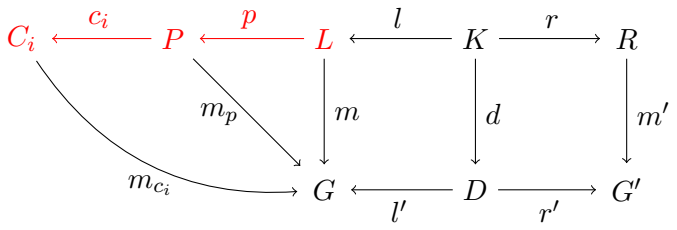


Figure 2.14: Application Condition for Graph Transformation Rules

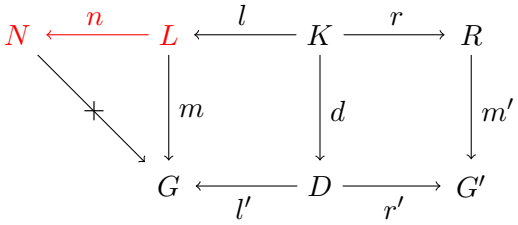


Figure 2.15: Negative Application Condition for Graph Transformation Rules

3 Requirements

We are interested in a *graph transformation tool* for endogenous model transformations of attributed typed graphs. The tool shall be usable for teaching and can be seamlessly integrated into a general purpose language (Java). In the following, we list the features we regard as desirable for the intended use cases in an order ranging from essential to less significant aspects. For each requirement we shall give a label, a short explanation and argue why the feature is useful.

Incrementality

Matching a pattern in a host graph basically requires solving the subgraph isomorphism problem which is known to be NP-complete in general. Incremental pattern matchers store the set of matches of a rule and incrementally maintain it as the model changes (cp. [VVS06], [BHRV08]).

With support for incrementality some features can be implemented more easily, e. g. notifications if a certain pattern is found for the first time or if a certain pattern cannot be matched anymore (cp. Section 6.4 for a description of tasks which are best solved with incremental pattern matching). Without incremental pattern matching one would need to check for new matches for the observed patterns after every change to be able to report new matches as soon as they appear.

Interpreter

The graph transformations are realized with an interpreter,[1] so no code generation is necessary to perform pattern matching and model manipulation.[2] The main advantage of an interpreter is that patterns can be found for all possible bindings. For code generation all bindings (free or bound) have to be fixed at specification time. The graph transformation engine should support setting parameters to fixed values before application.

For developers of graph transformation rules interpreters have the advantage that the rule can be directly tested without generating code. As code generation may take a long time if the rule set is large and everything needs to be regenerated, this can help to speed up the specification process.

In addition, the interpreter-based execution appears to be best suited for support of incrementality as we do not know any pattern matcher that is not an interpreter.

[1] Compared to generated code, an interpreter will have reduced performance in general, but as the tool is mainly intended for teaching, we consider performance to be of secondary importance.

[2] Generated code for the EMF meta-models as well as for the typed interface provided for Java integration can of course be used. In fact, eMoflon::IBeX-GT requires generated code for the meta-models and for the generated API.

Integration with a TGG tool

The graph transformation tool shall be integrated with a TGG tool such that the implementation can use shared libraries. From a developer's perspective the integration reduces the effort for maintaining both a GT and TGG tool.

The same meta-models can be used for GT and TGG specifications. This makes it possible to combine GT and TGG (e. g. GT for preprocessing of a model transformed via TGG). A similar syntax for rule specification simplifies learning the domain-specific language for rule specification and allows even to reuse parts of an existing specification.

Mature integration with a general purpose language

A seamless integration with Java 8 via an API allows to invoke model queries and transformations from standard Java source code. Thereby a developer can implement a method by just calling a graph transformation rule instead of writing code for pattern matching and performing changes in the model. This way queries for complicated graph structures and operations on them do not need to be programmed in Java, but can be specified in the GT editor and invoked by a program.

As the specification of patterns and rules does not require advanced programming skills this part can be understood much easier by people without programming experience. Ideally even domain experts instead of programmers could specify the rules.

The editor specification focuses on graph structures and does not include control flow. Instead the control flow structures of Java will be used in a program using the API. In this way the benefits of the textual pattern specification for complex graph patterns are combined with the full power of Java control flow structures.

Dedicated support for model queries

In addition to model transformations (i. e. applying transformation rules on matches in a given host graph), we are interested in querying models, i. e. just finding matches for the left-hand side of a rule. In the API this feature can be used to check constraints specified as patterns on the host graph or check rule applicability without actually applying any changes.

DPO or SPO semantics

The implementation shall be based on the algebraic approach to graph transformations and therefore be well-grounded on category theory, e. g. supporting the *double-pushout* (DPO) and *single-pushout* (SPO) semantics. With support for both approaches, one can flexibly switch between them, choosing the most suitable one for a specific use case.

Modularity on rule level

Pattern refinement[3] as a modularity concept allows to inherit and overwrite graph structures from super patterns. This way common parts can be shared between different patterns, such that they do not have to be copied into similar or more specific patterns.

[3] see Section 5.2.4 for a definition of pattern refinement

Application conditions

Graph conditions, especially *Negative Application Condition*s (NACs), can be specified to restrict rule applicability.[4] Simple application conditions shall be combinable via logical expressions.

Attribute manipulation

The editor shall support the specification of attribute constraints. New attribute constraints can be implemented in Java and integrated into the specification.

Textual concrete syntax and visualization

Graph transformation rules shall be specified textually in a DSL via an editor with syntax-highlighting. The textual syntax allows easy versioning with any version control software. A visualization is generated from the textual syntax to help users to understand large specifications, especially when using pattern refinement.

Modeling standard

The tool shall support a modeling standard (such as EMF) to provide interoperability of models with other tools.

End-user documentation

Detailed documentation for end-users shall be available such that people who are not familiar with the development of the tool can easily install and use it. For teaching purposes this is necessary because students should be able to quickly and easily learn to work with the tool.

[4] see Section 2.3 for a formal definition of application conditions

4 Related Work

This chapter presents existing graph transformation tools. Section 4.1 analyzes them with respect to our requirements from Chapter 3, while Section 4.2 summarizes the comparison results.

4.1 Graph Transformation Tools

There are already some tools available supporting the modification of attributed typed graphs via graph transformation. Many of them are used for academic purposes, while others are also applied in industry.

AGG ("The Attributed Graph Grammar System") [Run17], developed by Olga Runge (TU Berlin), interprets graph transformation rules with NACs and control structures based on the SPO semantics. While the latest version was released in 2017, the manual [Run06] is outdated (already ten years old). AGG offers an API for integration with Java programs, for which detailed documentation could not be found. The examples available on the website indicate that the API is not typed, thus requires casting.

AToMPM ("A Tool for Multi-formalism and Meta-Modelling") [SVM+16] is an interpreter and code generator for model queries and unidirectional transformations (both endogenous and exogenous) with a web-based user interface. The development of the tool is a joint project of universities in Montreal, Antwerp and Alabama. An integration into a general purpose language is not directly supported. AGG and AtoMPM do not support modularity for rules and parameterized rules.

eMoflon [AKK+17b] is a GT and TGG tool developed at TU Darmstadt and Paderborn University. Only eMoflon::SDM (the GT part) is checked for compliance with our requirements. The EMF-based Eclipse plugin relies on code generation. An end-user handbook [AKK+17a] is available, introducing eMoflon based on examples.

EMorF [Sol12], developed by Solunar GmbH, is an EMF-based Eclipse plugin with support for model queries and model transformations (model modification as well as bidirectional transformations with TGGs). It offers an API for integration into Java programs, but no detailed API documentation is available. The parameters of the API are not typed; the same holds for objects in matches so that casts are required. The rule specification is just introduced based on one example. The last release 0.4.2 was published in 2012, since then, the development has been discontinued.

GRAPE ("Graph Rewriting And Persistence Engine") [Web17], a GT interpreter by Jens Weber, can be integrated into Clojure programs. The GRAPE interpreter uses SPO semantics by default, but rules may specifiy that they are applied according to DPO semantics. Rules can be parameterized, but cannot refine other rules. Documentation of the textual syntax [Web16] with many examples is available online.

GrGen.NET ("Graph Rewrite Generator") [JBG17a] offers declarative rule specification for transformations, applied by a code generator. Rules may share subpatterns, but cannot

refine other rules. Their application follows SPO semantics. A type-safe API enables integration into C# code. Detailed documentation [JBG17b] is available.

GROOVE ("GRaphs for Object-Oriented Verification") [RdMZ17] provides graph transformations based on SPO rules. Graphs can be queried with Prolog, but graphs cannot be transformed via a general purpose language. Groove also lacks a modularity concept in the rule specification language.

Henshin EMF [BHK+16a] is an Eclipse-based graph transformaton tool providing an interpreter API for the usage in Java code. When using the API, rules must be loaded into a so-called unit which can be applied to the model.[1] The interaction with the API classes relies on rule and node names. The interface is not typed such that casting from `Object` to the concrete type is necessary. The interpreter supports both SPO and DPO which can be arbitrarily mixed. The tool is documented in a wiki [BHK+16b] accompanied by examples.

VIATRA [HVRU17], part of the Eclipse Modeling Project, is based on an incremental query backend and focuses on pattern matching and not on graph transformation. A Java API is available for integration into Java applications. A typed interface is generated for the API.[2] Even if the focus is not on algebraic graph transformation, VIATRA offers major support for many of our requirements, but lacks support for rule applications and integration with a TGG tool. From the tools presented in this chapter, it is the only one that supports incrementality. VIATRA offers composable and reusable patterns, but there is no modularity concept on rule level. No visualization is provided in the textual editor. The website offers tutorials and documentation.

4.2 Comparison of Existing Graph Transformation Tools

The comparison of the tools introduced in the previous section is summarized in Figure 4.1.[3] This chapter focuses on tools with at least one release since 2015 or special support for one of the requirements. It is partly based on the comparison of model transformation tools published by Kahani and Cordy [KC15a, KC15b]. Please note that the comparison takes solely our requirements into consideration and ignores other aspects completely. The tools have been analyzed in the version stated in Figure 4.1 according to their official documentation.

The symbol ✓ means that the requirement is fully fulfilled and ✗ states that the requirement is not fulfilled. The notation (✓) indicates that the requirement is only partially fulfilled, as stated in the previous section or in the footnotes.

VIATRA is the only tool with support for incrementality, while only eMoflon and eMorF combine GT and TGG within one tool. Most tools support integration into a general purpose language, but not all APIs are typed and well documented (see previous section).

No tool supports modularity in the sense of pattern refinement, only GrGen.NET and VIATRA allow to share the specification of graph structures between patterns. Only GrGen.NET and Henshin EMF support application conditions which are more complex than NACs. Except AtoMPM, all tools support one of the common modeling standards

[1] `https://wiki.eclipse.org/Henshin/Interpreter`

[2] `https://www.eclipse.org/viatra/documentation/query-api.html`

[3] Horizontal and vertical lines in the table are just for easier reading, the aspects are ordered as in the previous section.

EMF (Eclipse Modeling Framework), GXL (Graph eXchange Language), or Neo4J (a graph database management system).

Tool, Version → / Feature ↓	*AGG* 2.1	*AtoMPM* 0.6.1	*eMoflon::SDM* 2.32.0	*EMorF* 0.4.2	*GRAPE* 0.1.1	*GrGen.NET* 4.5.2	*GROOVE* 5.7.2	*Henshin EMF* 1.4.0	*VIATRA* 1.6.1
Incrementality	✗	✗	✗	✗	✗	✗	✗	✗	✓
Interpreter	✓	✓	✗	✓	✓	✗	✓	✓	✓
Integration with a TGG tool	✗	✗	✓	✓	✗	✗	✗	✗	✗
Mature integration with a general purpose language[4]	✓	✗	✗	✓	✓	✓	(✓)	✓	✓
Dedicated support for model queries	✗	✓	✗	✓	✗	✗	✗	✗	✓
DPO or SPO semantics	SPO	?[5]	SPO	SPO	both	SPO	both	both	–
Modularity on rule level	✗	✗	✗	✗	✗	(✓)	✗	✗	(✓)
Application conditions[6]	NACs	NACs	NACs	OCL[7]	NACs	✓	NACs	✓	NACs
Attribute manipulation[8]	(✓)	(✓)	✓	(✓)	(✓)	✓	(✓)	(✓)	(✓)
Textual concrete syntax and visualization[9]	✗	✗	✗	✗	✓	✓	✗	✗	✗
Modeling standard	GXL	✗	EMF	EMF	Neo4J	EMF	GXL	EMF	EMF
End-user documentation	(✓)	✓	✓	✗	✓	✓	✓	✓	✓

Table 4.1: Comparison of Graph Transformation Tools

[4] ✓indicates that an API for integration into Java, C#, or Clojure is available. GROOVE allows model queries via Prolog.
[5] The documentation does not describe the pushout semantics.
[6] For application conditions, a ✓indicates support for more complex application conditions than NACs.
[7] EMorF one can use constraints defined in the Object Constraint Language (OCL) to restrict the applicability of a rule.
[8] Most tools only support a limited set of attribute conditions, indicated by (✓). A ✓indicates that the user can define custom attribute relations.
[9] AGG, AtoMPM, EMorF, eMoflon::SDM, GROOVE, and Henshin EMF come with a graphical editor, while GRAPE and GrGen.NET provide a textual editor and a generated visualization. VIATRA makes use of a textual syntax, but does not provide any visualization.

5 Patterns in eMoflon::IBeX-GT

This chapter provides a short introduction to the architecture of the eMoflon::IBeX tool suite (Section 5.1) and describes the implementation of the graph transformation part in detail. Section 5.2 explains the language features supported by the textual editor and the transformation into patterns for the pattern matcher. The generated pattern networks are summarized in Section 5.3.

5.1 eMoflon::IBeX Architecture

The eMoflon::IBeX tool suite is implemented as a set of plugins for the Eclipse IDE and supports both unidirectional and bidirectional model transformations. Some functions such as code generation for meta-models and utilities for handling the interaction with the Eclipse framework are shared with eMoflon::SDM/TiE via eMoflon::Core.

Figure 5.1 gives an overview of the high-level architecture of eMoflon::IBeX. The tool can be divided into the editor (green), its core functionality (blue) and the Democles adapter (orange). The GT and TGG rules are defined in textual editors based on the Xtext[1] framework which provide features such as syntax highlighting, auto-completion, and validation. Graphical visualizations are implemented with PlantUML.[2] Utilities needed for both the GT and TGG editor have been moved to a shared project ("Editor Utils") or eMoflon::Core.

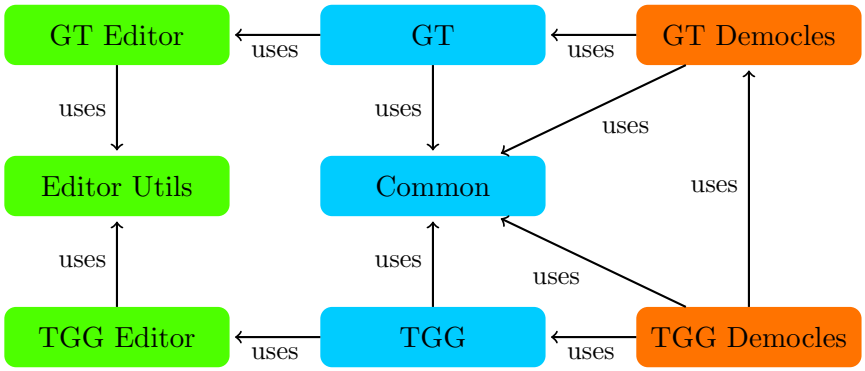


Figure 5.1: Component Diagram for eMoflon::IBeX (see Appendix A for details)

The Common project defines the meta-model for IBeX pattern model, a pattern network which is independent from a concrete pattern matcher. Currently IBeX patterns are used for GT only, but shall be shared with the TGG part later. In addition, some utilities for dealing with EMF models are provided.

[1] `https://www.eclipse.org/Xtext/`
[2] `http://plantuml.com/`

The GT and TGG projects provide Eclipse integration and runtime code which is independent from a concrete pattern matcher. To use eMoflon::IBeX, an adapter for a concrete incremental pattern matching engine is necessary. Currently only Democles [VAS12] is fully supported, although prototypes for Viatra and Drools exist.

5.2 Transformation of Graph Transformation Rules into IBeX Patterns

This section explains the transformation from the textual specification in the editor into IBeX patterns which can be understood by a pattern matching engine after another transformation into the pattern format of the engine. In the following, patterns and rules will be given by their textual syntax and graphical visualization. The interested reader is referred to the handbook [AR18] for more details about the textual syntax.

Figure 5.2 illustrates the use of model transformations in the eMoflon::IBeX implementation. The user specifies patterns and rules in text files with the file extension `gt` using an Xtext-based editor. Xtext automatically parses the file and transforms the textual specification into an editor model when a `gt` file is loaded.

The visualizations of the editor model, the IBeX patterns, and the Democles patterns are realized via transformations to PlantUML code, which is interpreted and displayed by the PlantUML Eclipse plugin.

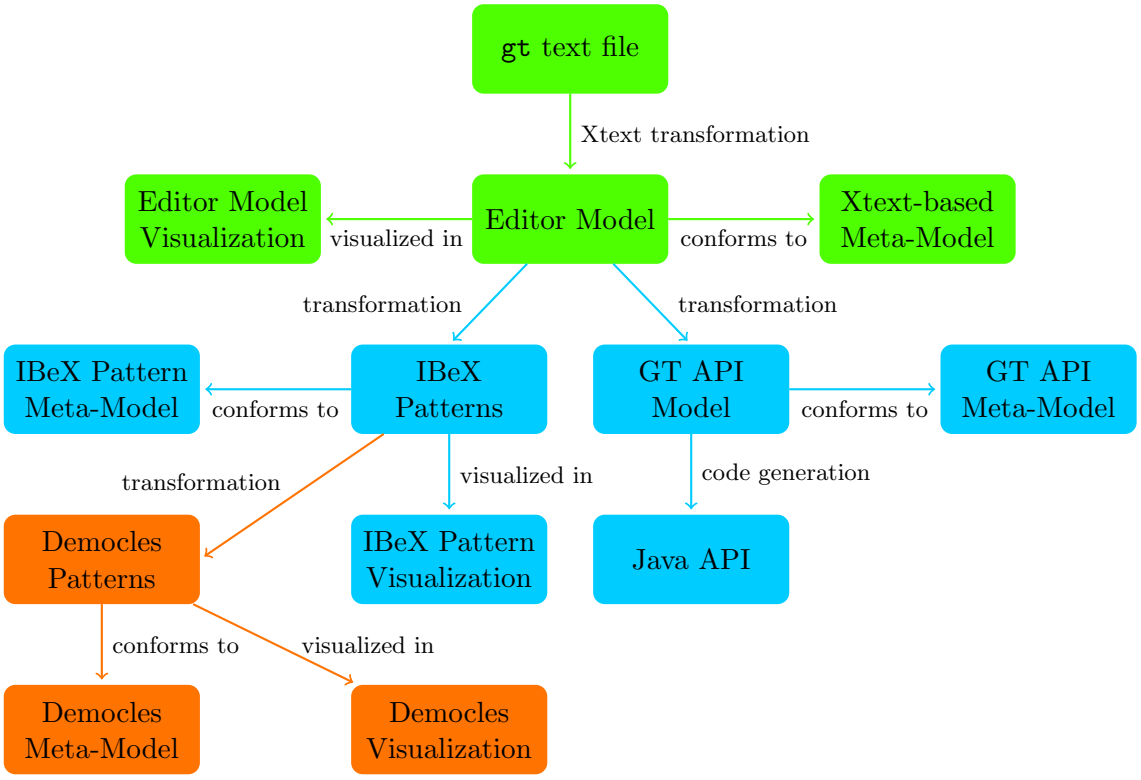


Figure 5.2: Model transformations for eMoflon::IBeX-GT

Whenever a project with `gt` files is built, the editor models of all `gt` files in a package are transformed into the GT API model and the IBeX pattern model. In addition, code for a typed Java API is generated as described in Section 6.1.

For each pattern and each rule a context pattern containing all context elements, deleted elements, and attribute conditions is generated. In addition, for each rule a create and a delete pattern is generated to define which elements must be created or deleted when the rule is applied. The create pattern contains all created nodes and references, together with any attribute assignments, while a delete pattern contains just deleted nodes and references. Figure 5.3 summarizes which parts of the editor model are included in which kind of the generated IBeX patterns.

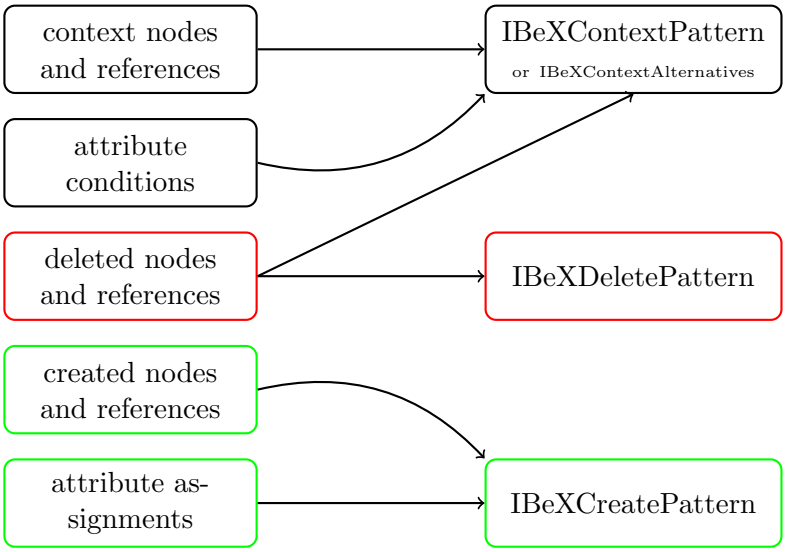


Figure 5.3: Editor model to IBeX patterns

The transformation from IBeX context patterns to the pattern representation used by a concrete incremental pattern matching engine (e. g. Democles) happens at runtime. If one wants to use eMoflon::IBeX with another pattern matcher, just the orange part in Figure 5.2 (the Democles adapter) needs to be implemented for the other engine, as just this part depends on the meta-model for Democles patterns.

5.2.1 Nodes and References

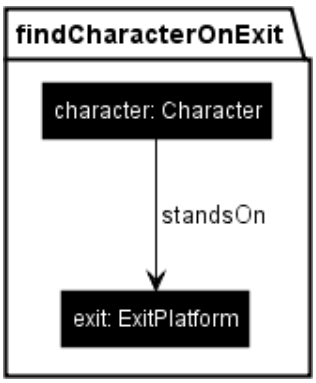
A pattern/rule consists of nodes and references between them. Each node and reference must have a type from an Ecore meta-model. The type of a node can be abstract – except if the node is created and the rule is not abstract (cp. Section 5.2.4). This ensures that all created nodes in applicable rules have a concrete type such that the node can be actually created, which would not be possible for an abstract type.

For each node and reference an operator defines whether it is context (shown with black background in the visualization), created (green) or deleted (red). Note that some combinations of node and reference operator do not make sense and are forbidden, e. g. there must not be a context reference in a created node as there cannot be a reference in a node which does not exist yet.

The pattern `findCharacterOnExit` (Figure 5.4) will match any characters standing on an exit platform. The rule `createCharacter` (Figure 5.5) which creates a new character and references from the character to an existing game and to an existing platform. The rule `moveCharacterToNeighboringPlatform` (Figure 5.6) deletes the `standsOn` reference between a character to its current platform and creates a new `standsOn` reference to another platform which must be a neighbor of the previous one.

```
pattern findCharacterOnExit {
    character: Character {
        -standsOn -> exit
    }

    exit: ExitPlatform
}
```

(a) Textual Syntax

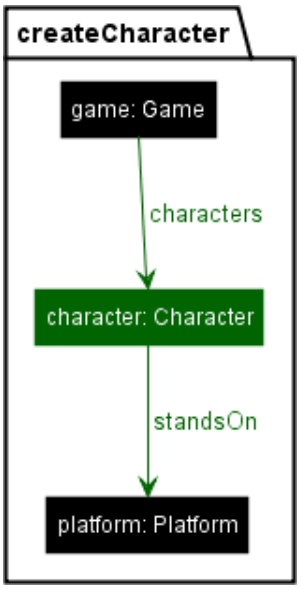


(b) Visualization

Figure 5.4: Pattern `findCharacterOnExit`

```
rule createCharacter {
    game: Game {
        ++ -characters -> character
    }

    ++ character: Character {
        ++ -standsOn -> platform
    }

    platform: Platform
}
```

(a) Textual Syntax



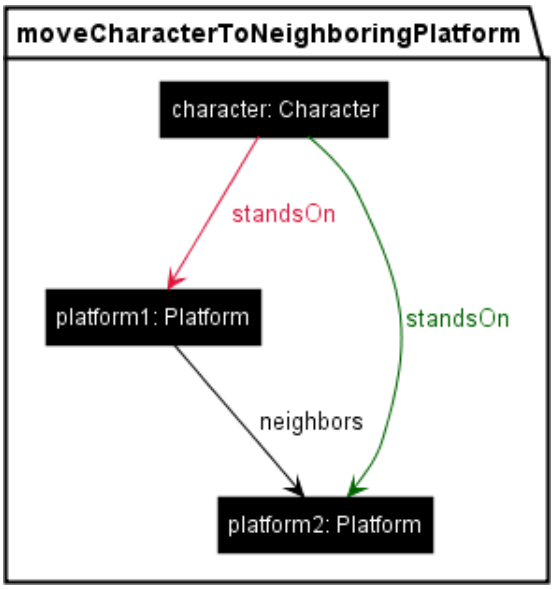
(b) Visualization

Figure 5.5: Rule `createCharacter`

In eMoflon::IBeX matches must be injective, i. e. nodes with different names must be matched to different objects in a match. For example, the two platforms in the rule `moveCharacterToNeighboringPlatform` must be different platforms. So a platform which has a `neighbors` edge to itself would not be a valid match. The decision for injectivity

```
rule moveCharacterToNeighboringPlatform {
    character: Character {
        -- -standsOn -> platform1
        ++ -standsOn -> platform2
    }

    platform1: Platform {
        -neighbors -> platform2
    }

    platform2: Platform
}
```

(a) Textual Syntax



(b) Visualization

Figure 5.6: Rule `moveCharacterToNeighboringPlatform`

per default has been made as this is what the user intuitively expects when looking at the visualization. If one does not want to have injectivity for a pattern, one has to specify different variants of the pattern.

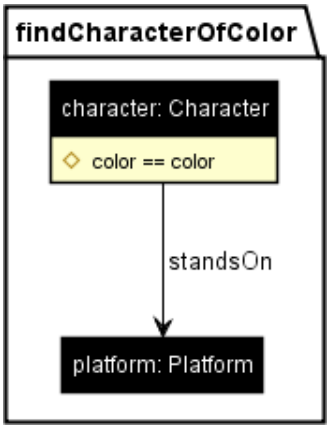
A match for a pattern contains all nodes of that pattern – except so-called local nodes. Per convention in eMoflon::IBeX-GT, a node is local if and only if its name starts with an underscore. If one was not interested in the platforms of Figure 5.6, one could name them `_platform1` and `_platform2` to omit them from in the matches for the pattern. So local nodes can be used to get smaller matches, as less elements are contained in the match. As the binding of local nodes is not relevant for the match, specifying nodes as local can reduce the number of matches because different bindings for the local nodes do not result in different matches.

5.2.2 Attribute Assignments and Conditions

Attribute assignments define new attribute values to be set when the rule is applied, while attribute conditions are a match filter based on a comparison of attribute values according to the defined relation (==, !=, <=, <, >, or >=). eMoflon::IBeX-GT supports constants, the attributes of other nodes, and parameters as attribute values as long as the type of the value fits to the one of the attribute. For example, a String attribute can only have a String value.

```
pattern findCharacterOfColor(color: COLOR) {
    character: Character {
        .color == param::color
        -standsOn -> platform
    }

    platform: Platform
}
```

(a) Textual Syntax

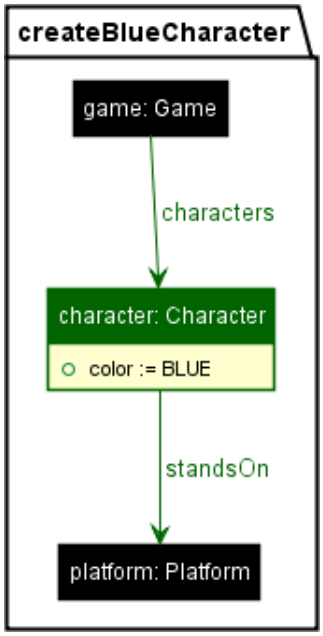


(b) Visualization

Figure 5.7: Pattern `findCharacterOfColor(color: COLOR)`

```
rule createBlueCharacter {
    game: Game {
        ++ -characters -> character
    }

    ++ character: Character {
        .color := enum::BLUE
        ++ -standsOn -> platform
    }

    platform: Platform
}
```

(a) Textual Syntax



(b) Visualization

Figure 5.8: Rule `createBlueCharacter`

The pattern `findColoredCharacter` (Figure 5.7) searches for a character which stands on a platform and has a certain color. The attribute condition defines that only characters whose color attribute equals the value of the parameter `color` of the type `COLOR` (an enum defined in the meta-model) can be matched. The rule `createBlueCharacter` (Figure 5.8) creates a new character whose color attribute is set to `BLUE`, given by an enum literal. Attributes which are not specified for created nodes are set to the default value defined in the Ecore meta-model.

Parameters for primitive data types and enums must be declared in the signature of a pattern or a rule. At run time they are replaced with a concrete value of the correct type.[3] As parameter values are only bound at run time, parameterized attribute conditions cannot be transformed to Democles, but are handled by the graph transformation interpreter.

Note that there are some logical restrictions when using attribute assignments and conditions: Attribute assignments must not be placed in deleted nodes (if the node was deleted, its attributes cannot be changed anymore) and conditions cannot occur within created nodes (the node does not exist yet and does not have any attribute values).

5.2.3 Applications Conditions

Application conditions are additional constraints on graph structures to check when finding matches for a certain pattern: The application conditions must be fulfilled for the match according to Definitions 12 and 14, otherwise the match is not valid. eMoflon::IBeX-GT allows to specify positive and negative application conditions and combine them via logical expressions `&&` (logical conjunction) and `||` (logical disjunction).

By convention, nodes of the same name in a pattern and the patterns used in its application conditions must be matched to the same node. The arrows in the visualization illustrate which nodes of the different patterns must be equal. Only the nodes of the main pattern are included in matches, any bindings of the nodes in the patterns of conditions will not be available in matches.

5.2.3.1 Negative Application Conditions

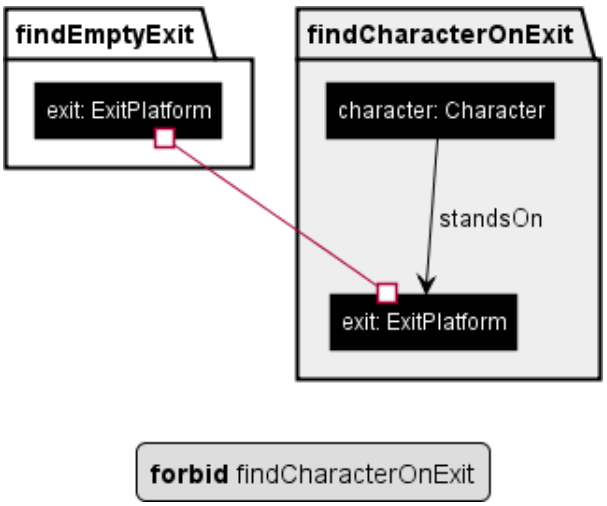
Via `forbid` a negative application condition (NAC) can be defined. A NAC invalidates matches with a certain pattern structure. For example, the pattern `findEmptyExit` (Figure 5.9) contains a NAC to ensure that no character stands on the exit platform of the main pattern. Otherwise any exit platform could be mapped for the `exit` node regardless of whether characters stand on it or not.

```
pattern findEmptyExit {
    exit: ExitPlatform
}
when noCharacterOnExit

condition noCharacterOnExit = forbid findCharacterOnExit
```

(a) Textual Syntax

Figure 5.9: Pattern `findEmptyExit`

[3] Section 6.3.5 explains how parameters can be passed via the API.



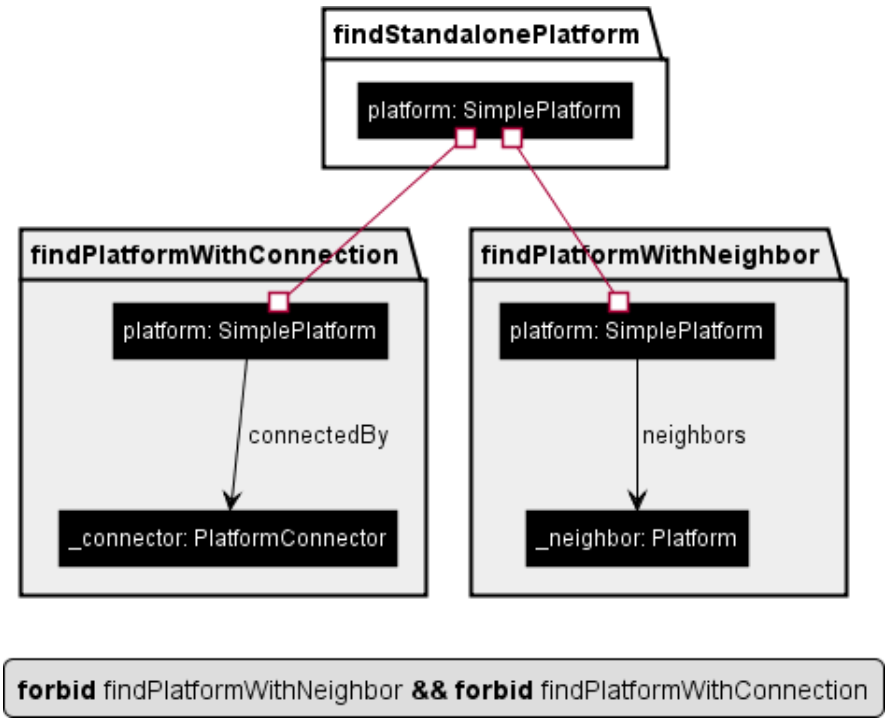
(b) Visualization

Figure 5.9: Pattern `findEmptyExit` (cont.)

```
pattern findStandalonePlatform {
    platform: SimplePlatform
}
when platformIsStandalone

condition platformIsStandalone =
    platformHasNoNeighbors && platformHasNoConnections

condition platformHasNoNeighbors = forbid findPlatformWithNeighbor

pattern findPlatformWithNeighbor {
    platform: SimplePlatform {
        -neighbors -> _neighbor
    }

    _neighbor: Platform
}

condition platformHasNoConnections = forbid findPlatformWithConnection

pattern findPlatformWithConnection {
    platform: SimplePlatform {
        -connectedBy -> _connector
    }

    _connector: PlatformConnector
}
```

(a) Textual Syntax

Figure 5.10: Pattern `findStandalonePlatform`



(b) Visualization

Figure 5.10: Pattern `findStandalonePlatform` (cont.)

Multiple application conditions can be combined via `&&` as illustrated in the pattern `findStandalonePlatform` (Figure 5.10). The two NACs restrict the matches for the simple platforms such that all platforms which are connected to another platform via a bridge or a wall are excluded as well as any platforms with a neighboring platform. Platforms matched by the pattern `findStandalonePlatform` cannot be left by a character standing on it. When a She Remembered Caterpillars world is created, such a standalone platform with a character standing on it should be avoided, otherwise the game cannot be finished.

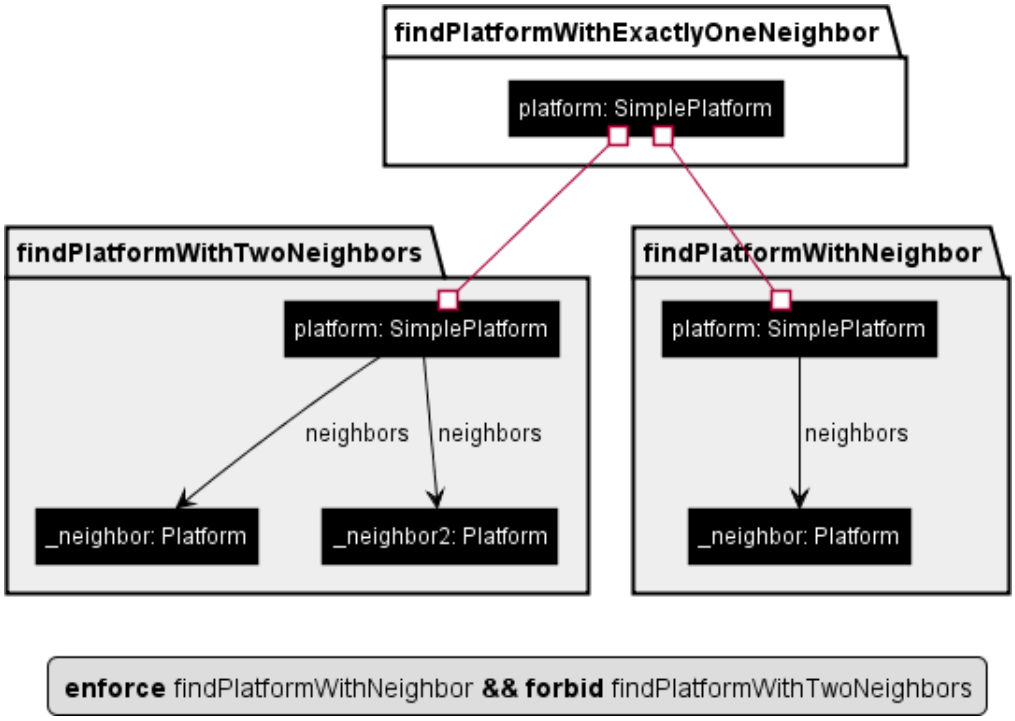
5.2.3.2 Positive Application Conditions

The pattern `findPlatformWithExactlyOneNeighbor` (Figure 5.11) uses a positive application condition (PAC) via `enforce` in combination with a NAC. The PAC ensures that the platform must have at least one neighboring platform and the NAC excludes any platforms which have at least two neighbors. Finally, only the platforms with exactly one neighbor remain as matches.

In this example, the PAC could be simply integrated into the main pattern without changing its meaning. However, in general PACs are necessary to express certain constraints: For example, multiple PACs combined via disjunction cannot be expressed with the integration into the main pattern (cp. example in Section 5.2.3.3). In addition, the nodes used in application conditions are not contained in the matches.

```
pattern findPlatformWithExactlyOneNeighbor {
    platform: SimplePlatform
}
when platformHasExactlyOneNeighbor

condition platformHasExactlyOneNeighbor =
    platformHasAtLeastOneNeighbor && platformHasAtMostOneNeighbor

condition platformHasAtLeastOneNeighbor =
    enforce findPlatformWithNeighbor

condition platformHasAtMostOneNeighbor =
    forbid findPlatformWithTwoNeighbors

pattern findPlatformWithTwoNeighbors {
    platform: SimplePlatform {
        -neighbors -> _neighbor
        -neighbors -> _neighbor2
    }
    _neighbor: Platform
    _neighbor2: Platform
}
```

(a) Textual Syntax



(b) Visualization

Figure 5.11: Pattern `findPlatformWithExactlyOneNeighbor`

5.2.3.3 Disjunctions

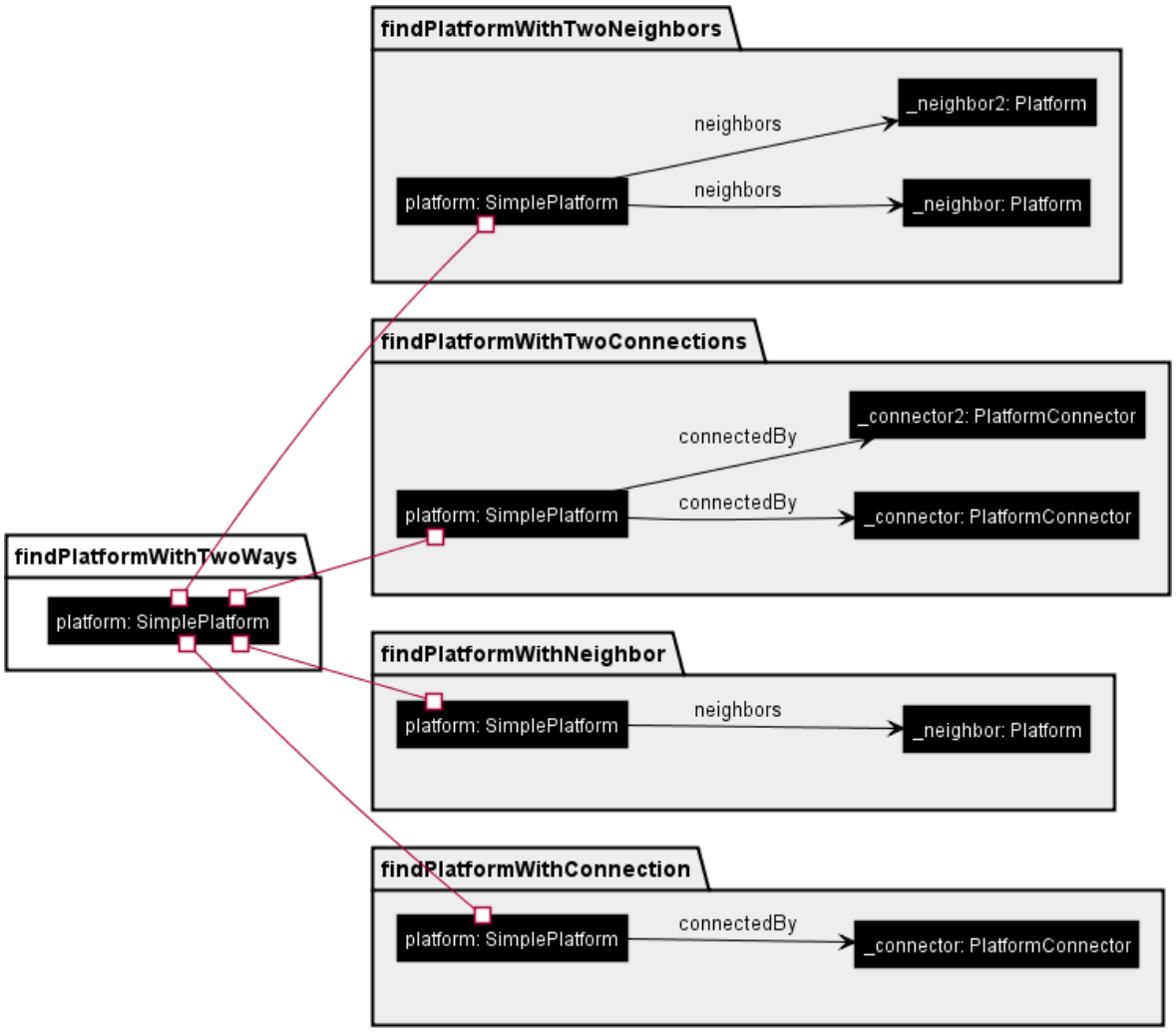
Application conditions can be used to express alternatives as well: For example, the pattern `findPlatformWithTwoWays` (Figure 5.12) searches for platforms which have at least two ways to enter or leave the platform (either via neighboring platforms or via bridges/walls). To express this, three application conditions need to be combined via `||` to deal with the possibilities that the platform (1) has two neighboring platforms, (2) has two connections to bridges or walls, or (3) one neighboring platform and one connection to a bridge or a wall.

```
pattern findPlatformWithTwoWays {
    platform: SimplePlatform
}
when platformHasAtLeastTwoNeighbors
  || platformHasAtLeastTwoConnections
  || platformHasNeighborAndConnection

condition platformHasAtLeastTwoNeighbors =
    enforce findPlatformWithTwoNeighbors

condition platformHasAtLeastTwoConnections =
    enforce findPlatformWithTwoConnections

condition platformHasNeighborAndConnection =
    platformHasExactlyOneNeighbor && platformHasExactlyOneConnection

condition platformHasExactlyOneConnection =
    platformHasAtLeastOneConnection && platformHasAtMostOneConnection

condition platformHasAtLeastOneConnection =
    enforce findPlatformWithConnection

condition platformHasAtMostOneConnection =
    forbid findPlatformWithTwoConnections

pattern findPlatformWithTwoConnections {
    platform: SimplePlatform {
        -connectedBy -> _connector
        -connectedBy -> _connector2
    }

    _connector: PlatformConnector

    _connector2: PlatformConnector
}
```

(a) Textual Syntax

Figure 5.12: Pattern `findPlatformWithTwoWays`

In the IBeX pattern model, patterns with multiple disjunctions are represented as a set of alternative patterns. A match for the pattern is defined as a match for any of the alternative patterns. For each alternative pattern a separate pattern is generated which



enforce findPlatformWithTwoNeighbors ||
enforce findPlatformWithTwoConnections ||
(**enforce** findPlatformWithNeighbor **&& forbid** findPlatformWithTwoConnections
&& enforce findPlatformWithConnection **&& forbid** findPlatformWithTwoNeighbors)

(b) Visualization

Figure 5.12: Pattern `findPlatformWithTwoWays` (cont.)

is given to the pattern matching engine because Democles cannot deal with alternatives directly (although this feature is planned for the future). As usual, Democles reports the matches for all patterns to the graph transformation interpreter, which has to collect and combine the matches of all alternative patterns, excluding duplicate matches. The removal of duplicates is necessary as Democles may report the same match for multiple alternative patterns.

The three alternatives for the pattern `findPlatformWithTwoWays` are disjoint. So no matches will be removed when combining the matches of the three alternatives because no duplicates can be found. Without the NACs in the third alternative, a match for a platform with two neighbors and a bridge connection (as the platform `p1` shown in Figure 5.13) would be reported by the first and the third alternative. The duplicate check would remove one of the matches, so the interpreter would still return the same set of matches. Without the removal of matches, a match reported by two alternatives would be returned twice.

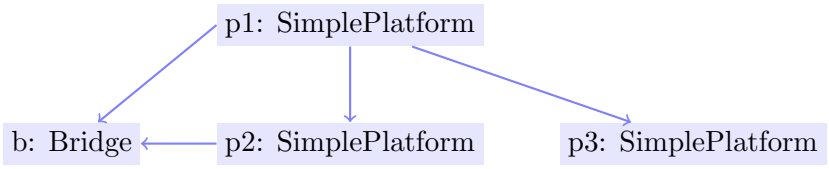


Figure 5.13: Example Model with Matches Reported for Two Alternatives

Note that logical expressions combining application conditions must be given in disjunctive normal form (DNF), i. e. the clauses combined via `||` must only contain PACs and NACs combined via `&&`. This is necessary to easily create the alternative patterns which can be given to the pattern matcher, as for each clause an alternative pattern is generated, containing the main pattern with the PACs and NACs from the clause.

5.2.4 Pattern Refinement

Pattern refinement is a modularity concept on the level of patterns which allows to share common subparts of the pattern with one or multiple super patterns. Similar to inheritance in object-oriented programming, this avoids declaring the same graph structures in multiple patterns. So pattern refinement helps to reduce copy and paste and leads to a specification which can be maintained more easily.

Patterns with super patterns are flattened, i. e. transformed into an equivalent version without pattern refinement. After that the flattened pattern is transformed into IBeX patterns as shown in Figure 5.3. The semantics of pattern refinement (how a pattern with super patterns can be flattened) is given by Definition 17.[4] Note that application conditions are not inherited to avoid additional complexity in situations in which the application conditions of super patterns come into conflict with each other. Due to this decision the user cannot lose track of the application conditions of a concrete pattern.

Patterns can be abstract: Abstract means that the pattern cannot be applied directly, but only exists to be used as a super pattern.

[4]The definition of rule refinement for Triple Graph Grammars by Stolte [Sto17] is generalized for graph transformations.

Definition 17 (*Refined Pattern*)

A pattern with one or more super patterns is called a *refined pattern*. The semantics of such a pattern is defined by the following constraints:

1. A pattern contains all nodes from its super patterns.
2. Equivalent nodes (identified by their name) are merged into one node.
3. Equivalent references (identified by their type, source and target node) are merged into one edge.
4. Equivalent attribute assignments (identified by their node, attribute type and value) are merged into one attribute assignment. There must not be different attribute assignments for the same attribute inherited from different patterns.
5. Equivalent attribute conditions (identified by their node, attribute type, relation and value) are merged into one attribute condition.
6. Equivalent parameters (identified by their name) are merged into one parameter. There must not be different type definitions for parameters of the same name.
7. When overriding a node/reference, created/deleted elements can be overwritten by context elements. Context elements must not be overwritten by created or deleted elements. Created elements must not be overwritten by deleted elements, deleted elements must not be overwritten by created elements.
8. The type of a node is the lowest of the types of all declarations of a node within a pattern and its super patterns. The type of a node can only be the same type or a lower type as declared for a node of the same name in a super pattern.

Table 5.1 shows some combinations of allowed and forbidden node type declarations in refined patterns and their super patterns according to the last constraint. Specifications without a lowest type and type not conforming to the type in the super pattern lead to an error message in the editor.

Node types in super patterns	Node type in pattern	Final type
Platform	(none)	Platform
Platform, SimplePlatform	(none)	SimplePlatform
ExitPlatform, SimplePlatform	(none)	ERROR (no lowest type)
ExitPlatform	Platform	ERROR (no subtype)
Platform	SimplePlatform	SimplePlatform
ExitPlatform	SimplePlatform	ERROR (no subtype)

Table 5.1: Allowed and Forbidden Node Type Changes in Refined Patterns

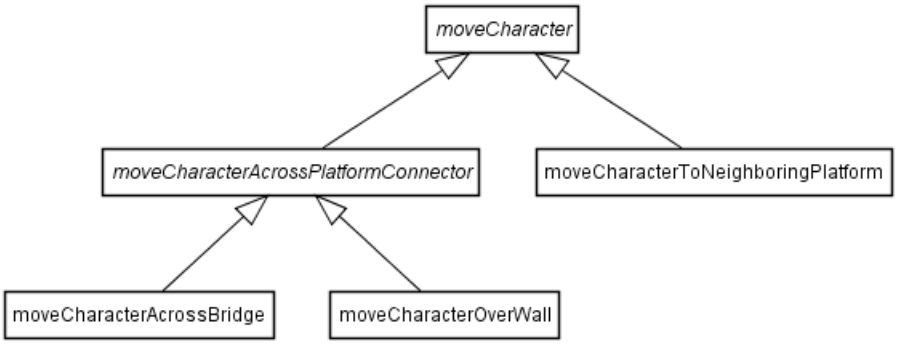
```
abstract rule moveCharacter {
    character: Character {
        -- -standsOn -> platform1
        ++ -standsOn -> platform2
    }

    platform1: Platform

    platform2: Platform
}

rule moveCharacterToNeighboringPlatform
refines moveCharacter {
    platform1: Platform {
        -neighbors -> platform2
    }
}

abstract rule moveCharacterAcrossPlatformConnector
refines moveCharacter {
    platform1: SimplePlatform

    platform2: SimplePlatform
}

rule moveCharacterAcrossBridge
refines moveCharacterAcrossPlatformConnector {
    bridge: Bridge {
        -connects -> platform1
        -connects -> platform2
    }
}

rule moveCharacterOverWall
refines moveCharacterAcrossPlatformConnector {
    wall: Wall {
        -connects -> platform1
        -connects -> platform2
    }
}
```

(a) Textual Syntax

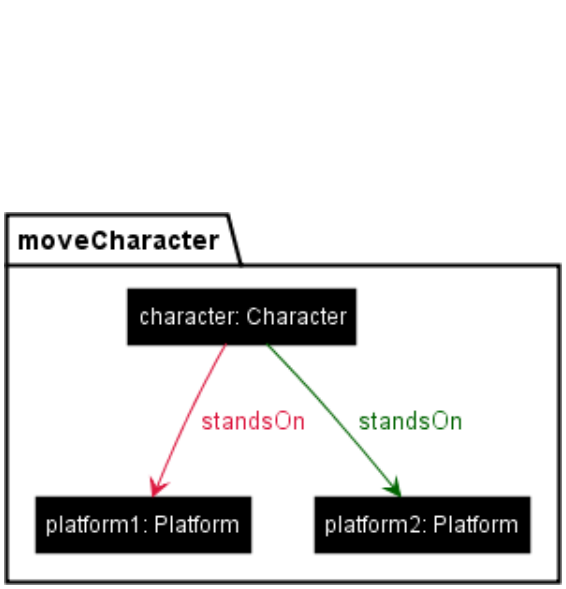
Figure 5.14: Rules with Refinements

Figure 5.14 shows some rules having a lot of elements in common. Using the pattern refinement hierarchy shown in Figure 5.14b, duplications in the textual specification can be avoided or at least significantly reduced as only nodes with additional references or another type need to be redeclared in the refined pattern. Nodes which are already defined in a super pattern are highlighted in bold. The visualization shows the refined rules after the flattening according to Definition 17.

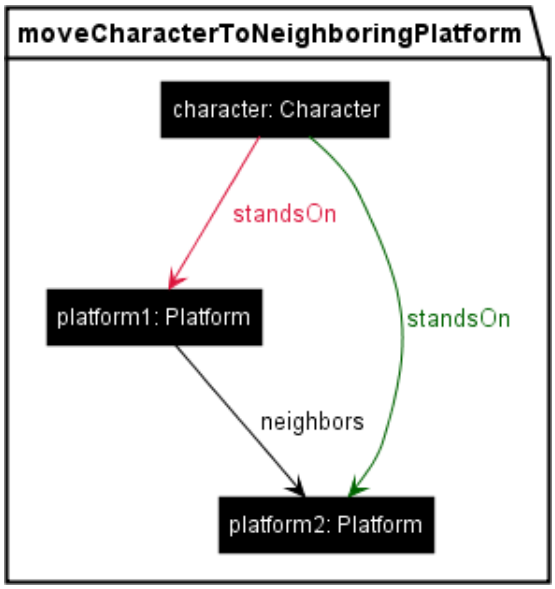
The rule `moveCharacterToNeighboringPlatform` just specifies the `neighbors` edge



(b) Refinement Hierarchy



(c) Abstract Rule `moveCharacter`



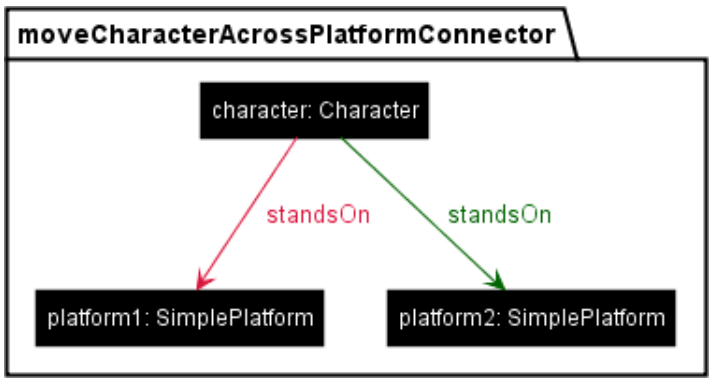
(d) Rule `moveCharacterToNeighboringPlatform`

Figure 5.14: Rules with Refinements (cont.)

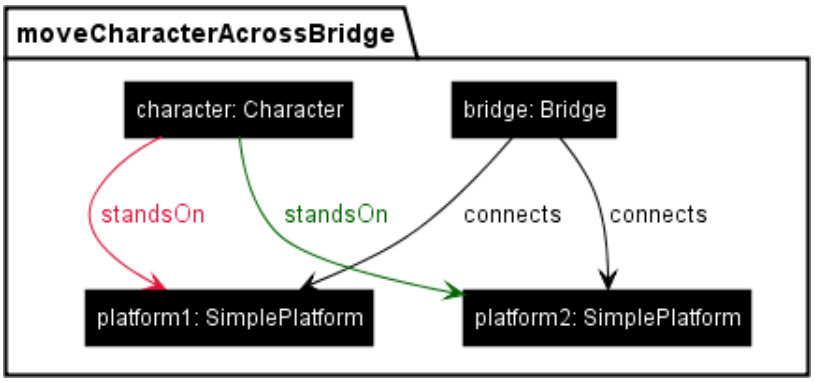
between the two platforms, all objects and the other edges are inherited from the super rule `moveCharacter`. The abstract rule `moveCharacterAcrossPlatformConnector` overrides the type of the platform nodes to a subtype, `SimplePlatform`. The rules `moveCharacterAcrossBridge` and `moveCharacterOverWall` define that the two platforms must be connected via a bridge or a wall, respectively.

The algorithm for the flattening of an editor pattern collects all nodes of the pattern and its super patterns and combines them into one large pattern (so-called co-product). After that parts which are equivalent according to Definition 17 are merged as described in the following:

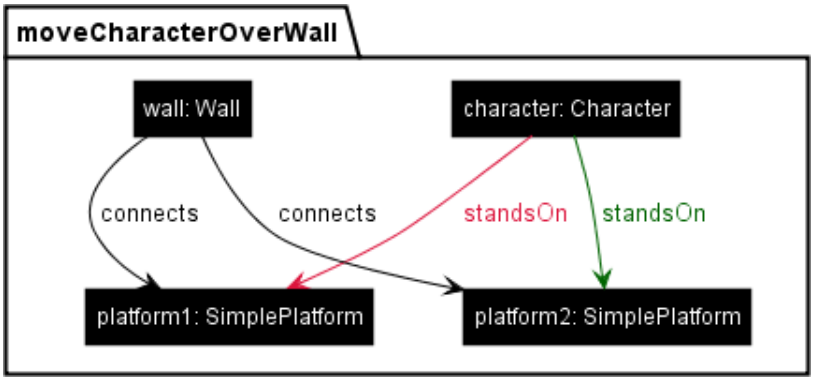
1. When merging two nodes, the operator is given by Table 5.2 (see constraint 7 in the definition) and the type of the node is the lower type of the two merged nodes. In the case that two nodes of the same name have different types and none of the types is a subtype of the other, the specification is invalid.



(e) Abstract Rule `moveCharacterAcrossPlatformConnector`



(f) Rule `moveCharacterAcrossBridge`



(g) Rule `moveCharacterOverWall`

Figure 5.14: Rules with Refinements (cont.)

2. When merging two equivalent references, attribute assignments, or attribute conditions only one of them remains and the other one is removed.

   a) The operator of merged references is given by Table 5.2.

   b) If there are multiple attribute assignments for the same attribute within a node, their values must be equal. Otherwise the specification is invalid, as one cannot decide which attribute value shall be assigned.

3. When merging parameters, the parameters of the super patterns are appended to the parameter list of the refined pattern. If parameters of the same name have different types, the specification is invalid.

	++	--	context
++	++	ERROR	context
--	ERROR	--	context
context	context	context	context

Table 5.2: Operators of Merged Nodes and References in Refined Patterns

For the flattening algorithm it is irrelevant whether a node is specified in the refined pattern or one of its super patterns. Only for parameters the order is relevant (because the parameters of the pattern specification become parameters for the constructor of the pattern). By repeating all parameters in the refined pattern, the user can influence the order of all parameters.

The user gets feedback on invalid specifications by error markers shown in the editor. When a project with invalid specifications is built, the problems are written on the console.

5.3 Pattern Networks

The previous sections introduced the features of the pattern language in the textual editor: nodes and references, attributes, application conditions, and pattern refinement. This section gives a summary how they are represented in the editor model, the IBeX pattern model, and in Democles. In addition, details on the transformations are provided.

The parser of the Xtext framework parses `gt` files into an editor model file containing `EditorPatterns` and `EditorConditions`. Figure 5.15 shows a simplified class diagram of the editor model. `EditorPatterns` have a type, either `PATTERN` or `RULE`. By using the same object for patterns and rules, both can easily be used together in a refinement hierarchy. They consist of a set of `EditorNodes`, which can have `EditorAttributes` and `EditorReferences`. An `EditorAttribute` has a relation (assignment or a comparison such as equals) and an attribute value. An `EditorReference` has a target node.

Both `EditorNodes` and `EditorReferences` have an operator, either `CONTEXT` (default value), `CREATE` (keyword `++` in the editor), or `DELETE` (`--`).

`EditorPatterns` have a set of `EditorConditions` (disjunction). `EditorConditions` represent a conjunction of conditions (a clause in the DNF), which can be a reference to another `EditorCondition` or a positive or negative `EditorApplicationCondition` (`enforce` or `forbid`). `EditorConditionReference` reference another condition.

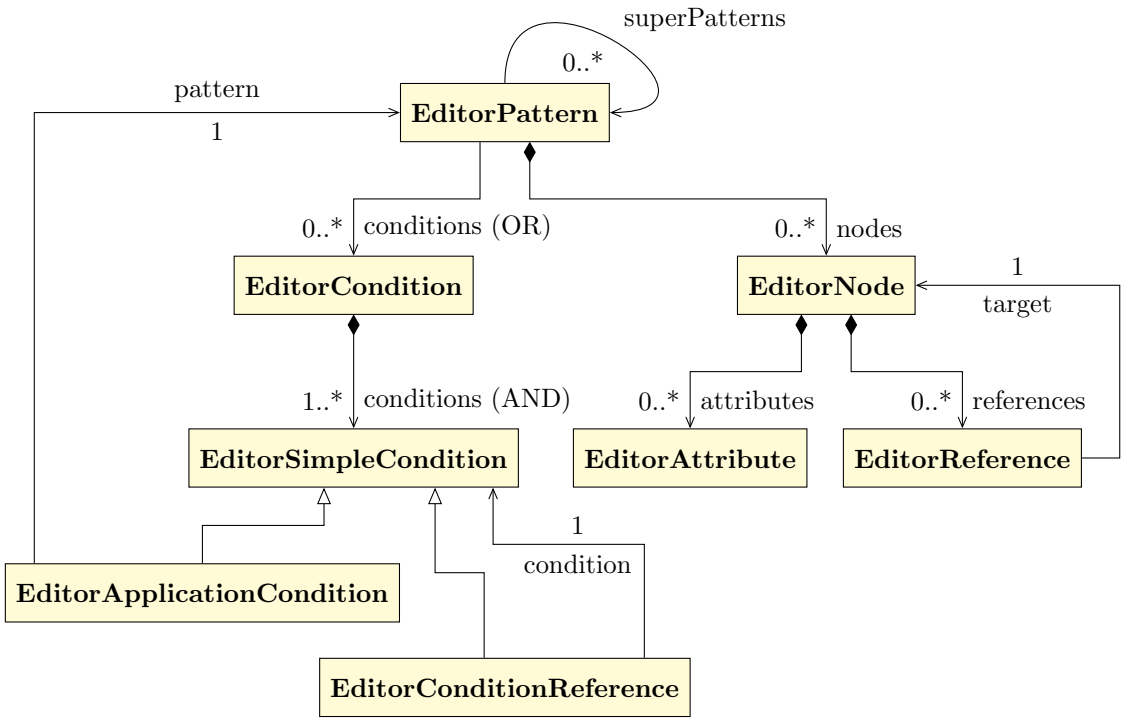


Figure 5.15: Simplified Meta-Model of the Editor Model

5.3.1 IBeX Pattern Networks

During the build of a graph transformation project in Eclipse with eMoflon::IBeX, the editor specification is transformed into the IBeX pattern model, which is saved in an `ibex-patterns.xmi`. Just this file is used by the graph transformation interpreter and must be read at runtime.

A simplified class diagram for the IBeX pattern model (just context patterns) is shown in Figure 5.16. An `IBeXContextPattern` consists of signature nodes, local nodes and edges, injectivity constraints, attribute constraints, and pattern invocations.

Table 5.3 summarizes how patterns and rules of the editor model are transformed into the IBeX pattern model. `EditorPattern`s with no or just one clause in the DNF are transformed into an `IBeXContextPattern`. If there are more than two disjunctions, the `EditorPattern` is transformed into an `IBeXContextAlternatives`, containing one `IBeXContextPattern` per clause in the DNF.

Each `EditorNode` is transformed to an `IBeXNode`. If the name of the name starts with an underscore, the node is local (i. e. not part of a match), otherwise the node is a signature node. For each pair of two nodes for which the type declaration does not ensure that the nodes are mapped to different objects, an injectivity constraint is added. Such a constraint defines a pair of nodes, which must not be equal.

Pattern refinement is not included in the table as the refinement is “flattened” and the flattened pattern is transformed afterwards.

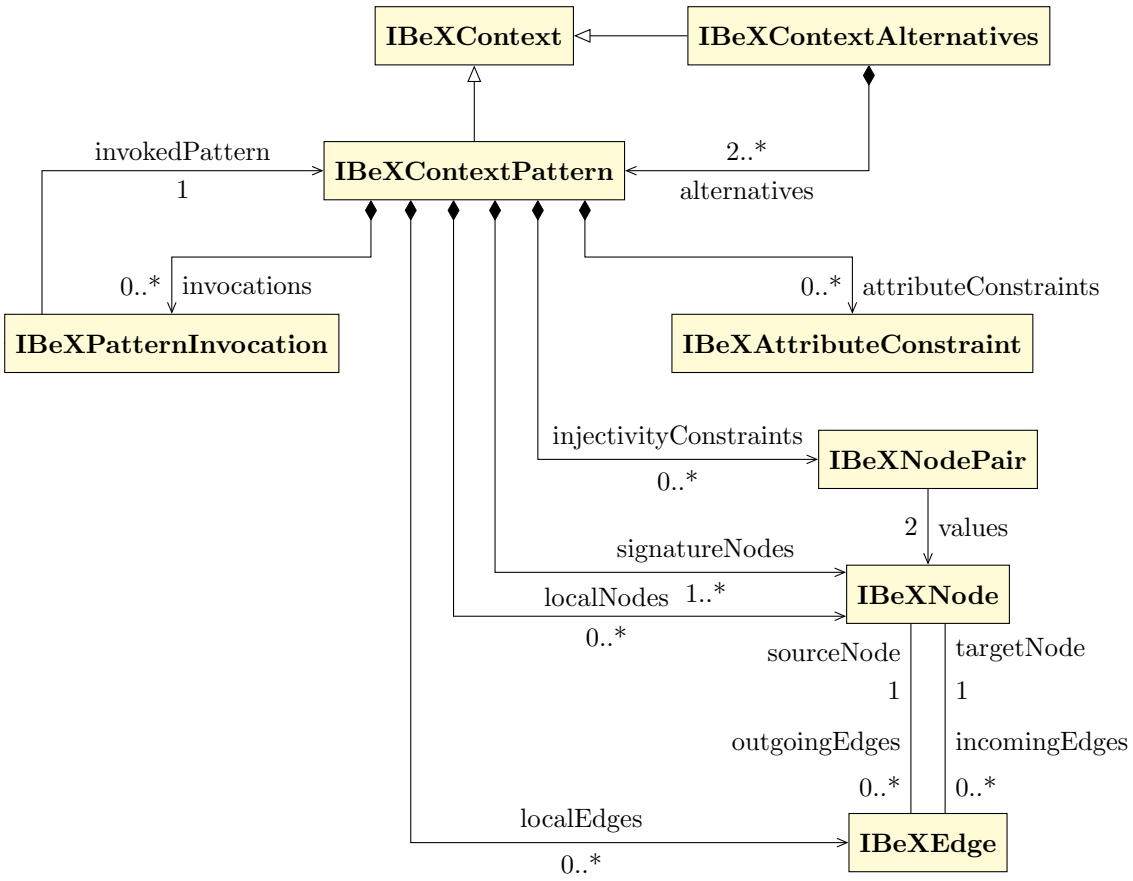


Figure 5.16: Simplified Meta-Model of the IBeX Pattern Model

Editor model	IBeX model
`EditorNode` (if the operator is `CONTEXT` or `DELETE`)	`IBeXNode` (as local node if the name starts with _, otherwise as a signature node) and injectivity constraints
`EditorReference` (if the operator is `CONTEXT` or `DELETE`)	Positive `IBeXPatternInvocation` of an edge pattern with two signature nodes and one `IBeXEdge`
`EditorAttribute` (if the relation is not an assignment)	`IBeXAttributeConstraint` (value given by a constant, a node and an attribute type or a parameter name)
`EditorApplicationCondition`	`IBeXPatternInvocation` (positive invocation for PACs, negative invocation for NACs)

Table 5.3: Transformation from the Editor Model into IBeX Patterns

References and application conditions are transformed into so-called pattern invocations. A pattern invocation is the equivalent to a method call in general programming languages. It defines that the graph structure of invoked pattern must be matched with the mapping of nodes from the original pattern to the signature nodes of the invoked pattern. The patterns connected via pattern invocations form a pattern network.

The context pattern generated for the rule `moveCharacterToNeighboringPlatform` (introduced in the previous section, Figure 5.14c) is shown Figure 5.17. The main pattern contains only the three nodes. The left pattern invocation ensures that there must be a `standsOn` edge between the objects mapped to the signature nodes `character` and `platform1` in the main pattern. `platform1` and `platform2` must be connected via a `neighbors` edge to fulfill the right pattern invocation.

Extracting the edges into edge patterns invoked by the main patterns leads to smaller patterns. Of course, edge patterns can be invoked by multiple patterns (or even multiple times by the same pattern, using different node mappings). For example, the edge pattern `edge-Character-standsOn-Platform` is invoked by all patterns which require a `standsOn` edge between a character and a platform.

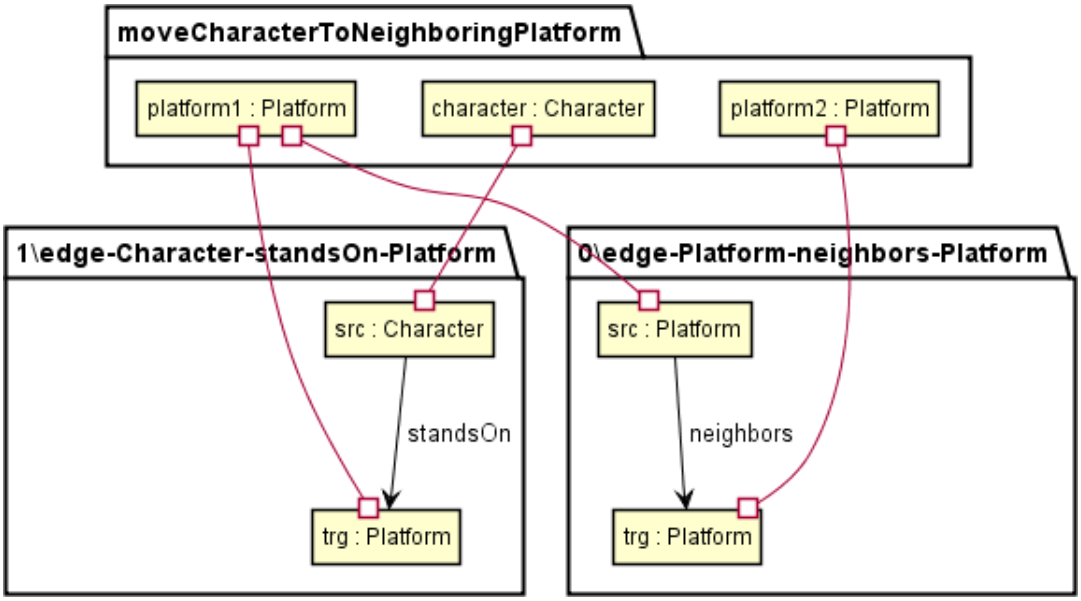


Figure 5.17: IBeX Context Pattern `moveCharacterToNeighboringPlatform`

5.3.2 Democles Pattern Networks

The meta-model of the pattern network used by Democles is shown in Figure 5.18 (simplified). A Democles `Pattern` consists of symbolic parameters and a `PatternBody`, which contains local variables and constraints.

All elements from the IBeX representation to their equivalent in the Democles model, as shown in Table 5.4. As Democles cannot handle parameterized attribute conditions and alternatives of multiple patterns, these two scenarios are handled by the interpreter via filtering and combining (see Section 6.3 for detailed information).

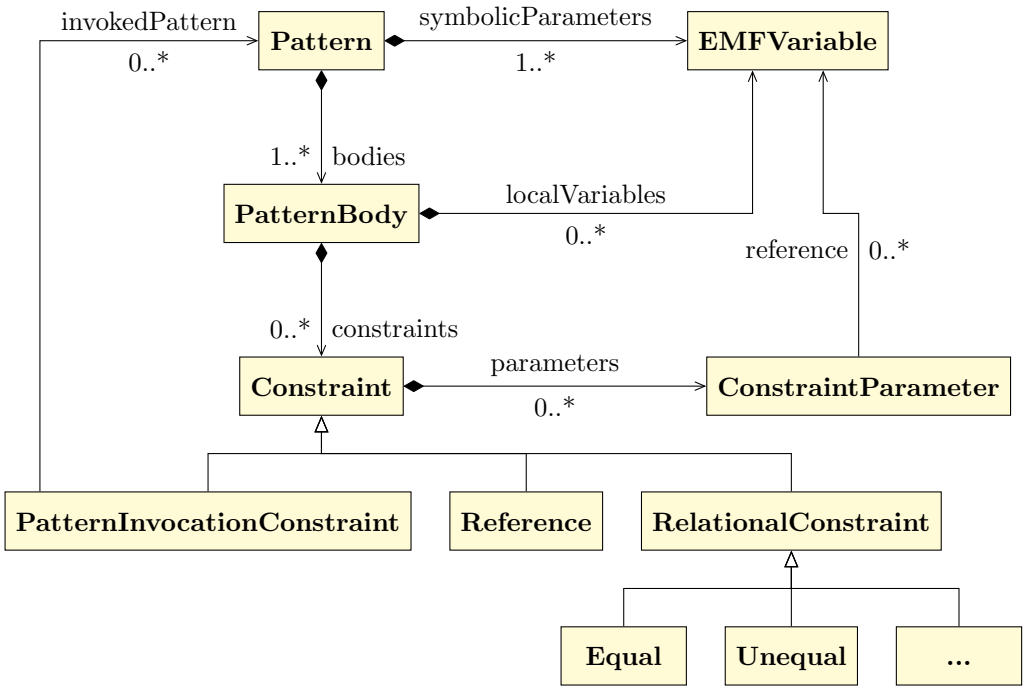


Figure 5.18: Simplified Meta-Model for Democles Patterns

IBeX model	Democles model
`IBeXNode` as local node	`EMFVariable` as local node
`IBeXNode` as a signature node	`EMFVariable` as symbolic parameter
`IBeXAttributeConstraint` (if the value is a constant or an attribute of another node)	Subtype of `RelationalConstraint` (for the correct relation) based on `EMFVariable`s and constants
Injectivity constraint (`IBeXNodePair`)	`Unequal` constraint
`IBeXEdge`	`Reference` constraint
`IBeXPatternInvocation`	`PatternInvocationConstraint`

Table 5.4: Transformation from IBeX Patterns into Democles Patterns

6 Graph Transformation Java API

This chapter describes how a typed Java API can be extracted from the textual specification such that the specified patterns and rules can be invoked from Java code without casting all results and losing type safety (Section 6.1). Section 6.2 describes how the API delegates method calls to the graph transformation interpreter, while Section 6.3 deals with the usage of the API. Finally, Section 6.4 presents the realization of features which exploit the incrementality of the underlying pattern matcher.

6.1 Code Generation for a Typed Java API

The graph transformation specification is placed in a graph transformation project in the Eclipse IDE.[1] All `gt` files must be placed in the `src` folder of the project. For each package containing at least one `gt` file an API will be generated. The generated code and models will be placed in corresponding packages in the `src-gen` folder.

Figure 6.1 gives an overview of the API classes assuming there is a package `example` containing a `gt` file with a pattern `ex1()` and a rule `ex2()`. For clarity, the class diagram shows only a small subset of the methods implemented by the abstract super classes. The API serves as a factory for patterns and rules as it provides methods for all non-abstract patterns and rules defined in the package. The app provides utility methods to create or load EMF resources and add them to the model and initialize the API for a concrete pattern matching engine. Within the packages `example.api.rules` and `example.api.matches` subclasses for the concrete patterns and rules are generated:

- The pattern/rule class contains methods for binding context and deleted nodes (except local ones) to a specific object (cp. Section 6.3.4 for more information on node binding). If a pattern/rule has parameters (cp. Section 6.3.5), they must be initialized in the constructor. In addition, setters for all parameters are generated.

- The match class contains getters for all nodes in the pattern/rule except the ones marked as local.

Pattern and rule classes inherit from a super class, `GraphTransformationPattern` or `GraphTransformationRule`. Match classes inherit from `GraphTransformationMatch`. These super classes are abstract and implement all methods which are independent from a concrete specification such that only a minimal subset of method implementation needs to be generated.

[1] Any Java and plugin project will be automatically converted to a graph transformation project when adding at least one `gt` file via the wizard. Note that Eclipse projects can have multiple natures: A graph transformation project must have at least the GT, Java and plugin nature.

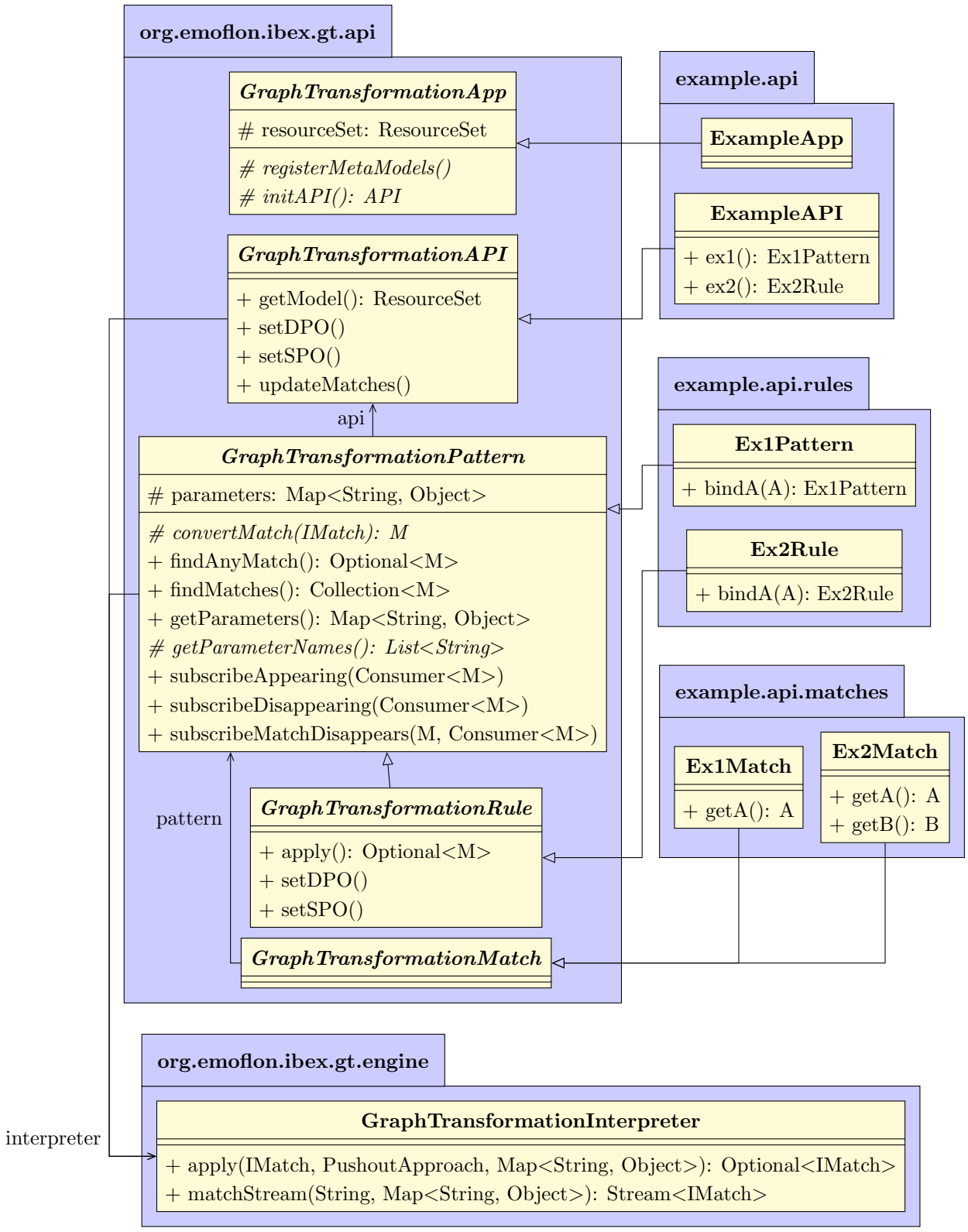


Figure 6.1: Overview of the API Java Classes

One difference to other graph transformation tools which provide an API for the specified rules (e. g. EMorF, Henshin)[2] is that the API generated by eMoflon::IBeX-GT is typed to avoid casting in the code using the API (cp. Section 6.3). If the usage of the API does not fit to the pattern or rule specification, the developer will get error messages at compile time instead of class cast exceptions at runtime. The typing in the API requires generated code for the meta-models used in the patterns and rules, as the interfaces of those are referenced in the generated API code. Per default, eMoflon::IBeX-GT assumes that the generated code for the meta-model is in a Java package named as the package in the Ecore file of the meta-model. This can be adjusted for custom use cases via a setting in a properties file.[3]

6.2 Graph Transformation Interpreter

The interaction of the `GraphTransformationInterpreter` and the API is explained in this section. The interpreter is independent of a concrete rule specification and ignores typing (i. e. everything is an `Object` in the method signatures, cp. Figure 6.1). The API provides a typed interface for the graph transformation interpreter such that all methods only accept objects of the correct type as defined in the editor specification. Figure 6.2 illustrates how the API classes (shown with a purple border) use the graph transformation interpreter (olive border). The interpreters for context, deletion and creation are shown in their typical colors black, red and green.

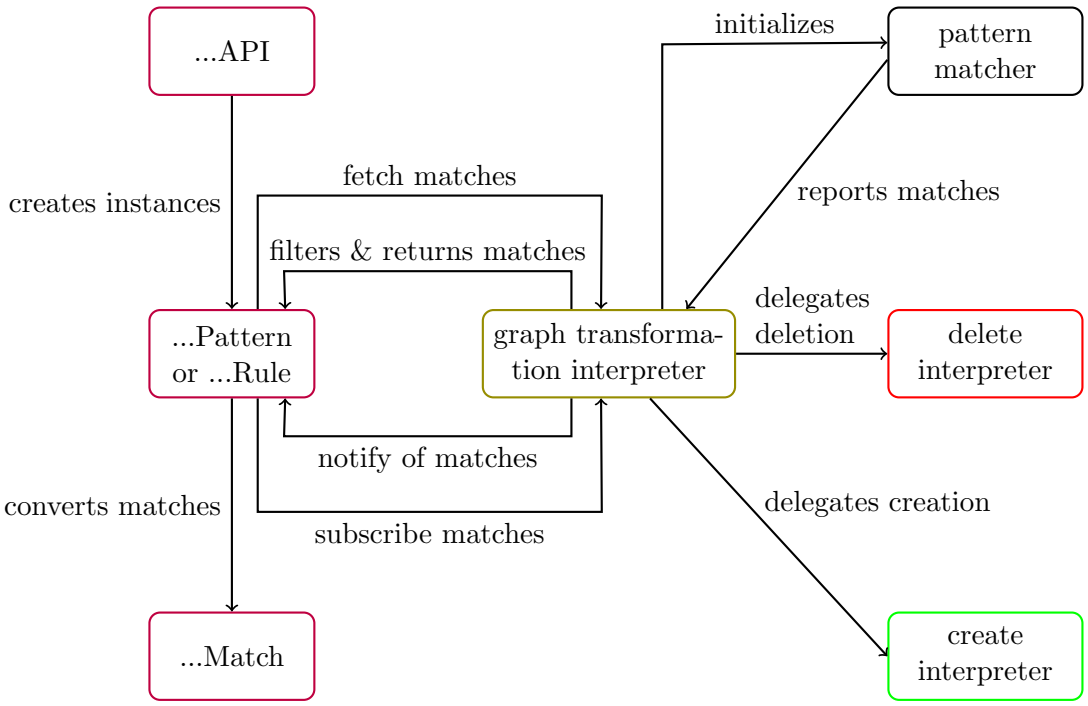


Figure 6.2: API and Graph Transformation Interpreter

[2]The tool Viatra [HVRU17] also provides a generated, typed API in a similar way as eMoflon::IBeX-GT, but cannot handle rule applications.

[3]see appendix of the handbook [AR18], Section "Frequently Asked Questions"

The graph transformation interpreter initializes the pattern matching engine (Democles) by registering the pattern set. Democles will report any appearing and disappearing matches such that the graph transformation interpreter can maintain a set of `IMatch`es for each pattern. `IMatch`es are an untyped representation of matches, which abstracts from the structure in the concrete pattern matching engine.

Remember that the API class is a factory for patterns and rules, i. e. the methods return an instance of a pattern or a rule. On a pattern/rule instance nodes and parameters can be bound to a fixed value (cp. the following section for details). When a method for querying matches is called on a pattern or a rule, the call is delegated to the graph transformation interpreter which will return the matches as a `Stream` of `IMatch`es. Using a `Stream`, unnecessary conversions between `Stream`s and collections can be avoided in the implementation of the API. If nodes are bound or parameters are set, the matches will be filtered according to the fixed nodes or parameter values. If a pattern is defined as being one of multiple alternative patterns, the graph transformation interpreter will combine the matches of all alternative patterns and remove duplicates (cp. Section 5.2.3.3). Before a pattern or a rule returns the matches, these matches are converted to a typed representation.

Before a rule can be applied, a match must be found where the rule can be applied to. If a rule contains only created elements, the rule is always applicable. An empty match will be generated in this special case. Rule applications are delegated to the delete and the create interpreter which handle the rule application according to the defined pushout approach using the create or delete patterns from the IBeX model.

6.3 Usage of the API

This section deals with the usage of the Java API. The API features are explained using small example code snippets.

6.3.1 Initialization and Conventions on EMF Resources

An EMF model is assumed to be represented by a `ResourceSet`, containing one or more resources (files, usually having the file extension `xmi`). If not explicitly specified, the first resource in the `ResourceSet` is chosen as default resource. All created nodes whose resource is not determined automatically due to their container object, will be placed in the default resource. Deleted nodes will be moved to a trash resource, which is created automatically.

The generated app class provides convenient methods for model initialization and setting the default resource. The meta-models used by the patterns and rules in the API will be registered automatically. The easiest way to initialize an API with a model is to implement a subclass of the Democles app generated in the API package, as shown in Listing 6.1. As the method names suggest, the method `createModel(URI)` creates a new empty resource with the given URI, while `loadModel(URI)` loads an existing resource.

In this case, the file `newModel.xmi` will be the default resource such that created elements will be added to this file – except if the container of the created elements is an object in the file `model.xmi`. To use `model.xmi` as the default resource, it can be set via `setDefaultResource(Resource)`. Alternatively, changing the order of the two method calls adding a resource to the model will lead to the same result.

```
public class MyGTApp extends
    SheRememberedCaterpillarsGraphTransformationDemoclesApp {
     SheRememberedCaterpillarsGraphTransformationAPI api;

     public MyGTApp() {
         String path = "./instances/";
         createModel(URI.createFileURI(path + "newModel.xmi"));
         loadModel(URI.createFileURI(path + "model.xmi"));

         api = initAPI();
     }

     public static void main(String[] args) {
         new MyGTApp();
     }
}
```

Listing 6.1: Loading Models

6.3.2 Model Queries

Listing 6.2 shows examples how the model can be queried via the API using the pattern `findCharacterOnExit` (Figure 5.4). All methods for querying matches fetch the untyped matches from the interpreter and convert the matches into the typed representation. To get the number of matches, one should always use the method `countMatches()` because the implementation avoids the conversion.

```
public Optional<Character> findAnyCharacter() {
    return api.findCharacterOnExit()
        .findAnyMatch()
        .map(m -> m.getCharacter());
}

public List<Character> findAllCharacters() {
    return api.findCharacterOnExit()
        .matchStream()
        .map(m -> m.getCharacter())
        .collect(Collectors.toList());
}

public long countCharacters() {
    return api.findCharacterOnExit().countMatches();
}
```

Listing 6.2: Model Queries

6.3.3 Rule Applications and Pushout Approaches (DPO vs. SPO)

Rules can be applied via the `apply()` method, which returns an `Optional` for the co-match. The `Optional` may be empty if the rule is not applicable.[4] Per default rules are applied according to the single pushout approach such that any dangling edges are deleted. This behavior can be changed on API level (holds for all future rule applications which do not overwrite this setting), for all applications of a certain rule or for a concrete rule application.

In Listing 6.3 the rule is applied via DPO. Note that the set pushout approach holds just for this rule. To enable DPO for the applications of all rules, the setting must be changed on API level via `api.setDPO()`.

```
1 public void moveDPO() {
2     api.moveCharacterToNeighboringPlatform()
3         .setDPO()
4         .apply()
5         .ifPresent(m -> {
6             String name = m.getCharacter().getName();
7             System.out.println("Character␣" + name + "␣moved␣to␣
                  neighboring␣platform");
8         });
9 }
```

Listing 6.3: Usage of DPO for a Single Rule Application

6.3.4 Node Bindings

By default all nodes of a pattern are unbound, i. e. they can be matched to any object of the correct type. A node binding fixes a node to a specific object. At runtime the graph transformation interpreter filters the matches reported by the pattern matching engine for those whose nodes are bound to the same objects as defined by the node binding.

A node binding can be defined for all context nodes and deleted nodes (except local nodes).[5] The API provides typed `bind` methods for those nodes. For convenience there are methods to bind all parameters of any match to the nodes of the same name. This allows to pass the bound objects in a match result of a model query or a rule application directly to another query or rule.

Listing 6.4 gives an example for the usage of node bindings: The query selects an arbitrary character which is not on an exit platform. After that, this character is bound to the application of the rule `moveCharacterToNeighboringPlatform` (Figure 5.6) such that only the selected character can be moved if a match for the rule can be found. The method `moveCharacter2` in Listing 6.5 does exactly the same as `moveCharacter`, since `character` is the only node name which occurs both in the pattern `findCharacterNotOnExit` and in the rule `moveCharacterAcrossBridge` (cp. Figure 6.3). So the `bind`-method will bind only the character and ignore the other nodes.

[4] A rule is not applicable if there is no match for the rule or the DPO approach forbids the rule application on the chosen match due to dangling edges.

[5] Since local nodes are not contained in matches, so local nodes cannot be used as a filter on matches.

```java
public void moveCharacter() {
    api.findCharacterNotOnExit()
        .findAnyMatch()
        .ifPresent(m -> {
            api.moveCharacterToNeighboringPlatform()
                .bindCharacter(m.getCharacter())
                .apply();
        });
}
```

Listing 6.4: Node Binding

```java
public void moveCharacter2() {
    api.findCharacterNotOnExit()
        .findAnyMatch()
        .ifPresent(m -> {
            api.moveCharacterToNeighboringPlatform()
                .bind(m)
                .apply();
        });
}
```

Listing 6.5: Node Binding based on Naming Convention

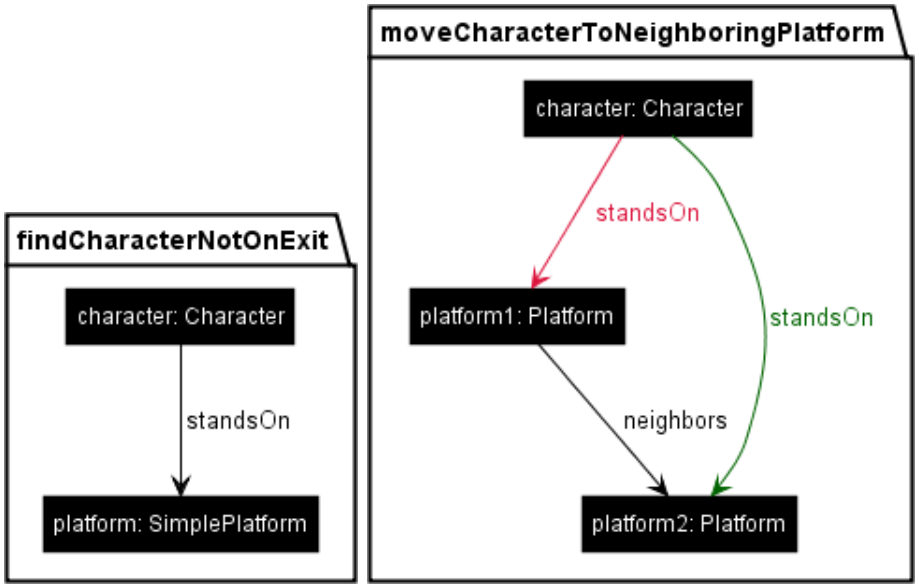


Figure 6.3: Common Nodes of the Pattern `findCharacterNotOnExit` and the Rule `moveCharacterToNeighboringPlatform`

6.3.5 Parameters

Patterns and rules may define parameters of primitive data types to be used in attribute assignments or conditions. In contrast to node bindings, parameters are required. When calling a pattern or rule, all parameters must be set in the constructor. The parameter values may be changed using the setters.

Attribute conditions with parameters are filtered by the graph transformation interpreter similar to node bindings while all other attribute conditions are checked by the pattern matching engine. Parameterized attribute assignments will set the value of the parameter as new attribute value. An example for this is shown in Listing 6.6: The parameter for the color is passed to the pattern `findCharacterOfColor` (Figure 5.7) in the constructor (so it cannot be omitted!). Without the parameter, the attribute condition could not be evaluated. The interpreter knows all matches for the pattern reported by Democles and filters those for the ones whose color attribute has the given value `BLUE`.

```
1 public void outputBlueCharacters() {
2     api.findCharacterOfColor(COLOR.BLUE)
3         .forEachMatch(m -> {
4             System.out.println(m.getCharacter().getName());
5         });
6 }
```

Listing 6.6: Parameters

6.4 Exploiting the Incrementality

As eMoflon::IBeX-GT is based on an incremental pattern matcher, the incrementality can be exploited to support certain scenarios which are more difficult to implement without support for incrementality. The following sections describe tasks which could not be easily supported without incrementality because they require permanent observation of all matches. The incremental features add support for reactive programming [BCvC+13], which is based on automatic propagation of changes.

6.4.1 Notification System

The pattern matcher permanently maintains a set of matches for all patterns and notifies the interpreter every time a new match appears or an existing match disappears. This can be used to provide a notification system in the API: Subscribers can register themselves for notifications of appearing and disappearing matches. If subscribers are registered, the interpreter forwards the notification of appearing or disappearing matches to them.

One application scenario is the permanent checking of constraints via reception of notifications whenever matches for the observed pattern appear or disappear. For positive constraints there must be a match, otherwise the constraint is violated. For negative constraints any reported match is a violation of the constraint.

In the context of the She Remembered Caterpillars game the notification system could be used to check whether the goal of the game (all characters stand on an exit platform) is reached by subscribing all matches for a character not on an exit platform: As soon as

the number of characters which are not on an exit platform reaches 0, the game is over. Note that one needs to subscribe to the disappearing matches as well such that characters leaving an exit platform are added to the set again. Listing 6.7 shows the necessary code for initializing and maintaining the set of characters which have not reached their final destination yet. The defined `Consumer`s will be called automatically whenever a change in the set of matches for the subscribed pattern is reported.

Without the notification system, one would need to check the subscribed pattern after every change in the model to be sure to notice that the game is over as soon as the last character reached an exit platform. The incremental pattern matcher notices even changes made by third party-applications and not via the API – and the changes in the set of matches triggers notifications if a subscription has been registered for those.

```
1  Set<Character> characters = new HashSet<Character>();
2
3  public void registerSubscriptions() {
4      FindCharacterNotOnExitPattern notOnExit = api.
           findCharacterNotOnExit();
5      notOnExit.matchStream()
6          .map(m -> m.getCharacter())
7          .forEach(c -> this.characters.add(c));
8      notOnExit.subscribeDisappearing(m -> {
9          this.characters.remove(m.getCharacter());
10         checkEndOfGame();
11     });
12     notOnExit.subscribeAppearing(m -> {
13         this.characters.add(m.getCharacter());
14         checkEndOfGame();
15     });
16 }
17
18 private void checkEndOfGame() {
19     if (this.characters.size() == 0) {
20         System.out.println("GAME␣OVER!");
21     }
22 }
```

Listing 6.7: Subscription of Notifications

6.4.2 Instant Automatic Rule Application

Due to the notification of appearing matches, instant and automatic rule application can be easily supported. If automatic rule application is enabled, the notification system will send a notification and the interpreter will apply the rule immediately after the new match appeared. Note that this only holds for matches reported after enabling automatic rule applications. Rule applications can be subscribed via the API's notification system.

Listing 6.8 shows an example of enabling automatic rule application. Assuming that the rule `transformBlueAndRedToPurpleCharacter` (Figure 6.4) shall be applied as soon as a match for this rule is found, i. e. a blue and a red character stand on the same

platform. Via a subscription of the rule applications, a `Consumer` outputs "Automatic transformation" on the console whenever the rule is applied.

```
1 public void enableAutoTransformations() {
2     TransformBlueAndRedToPurpleCharacterRule transformation =
          api.transformBlueAndRedToPurpleCharacter();
3     transformation.subscribeRuleApplications(m ->
4         System.out.println("Automatic transformation"));
5     transformation.enableAutoApply();
6 }
```

Listing 6.8: Instant Automatic Rule Application

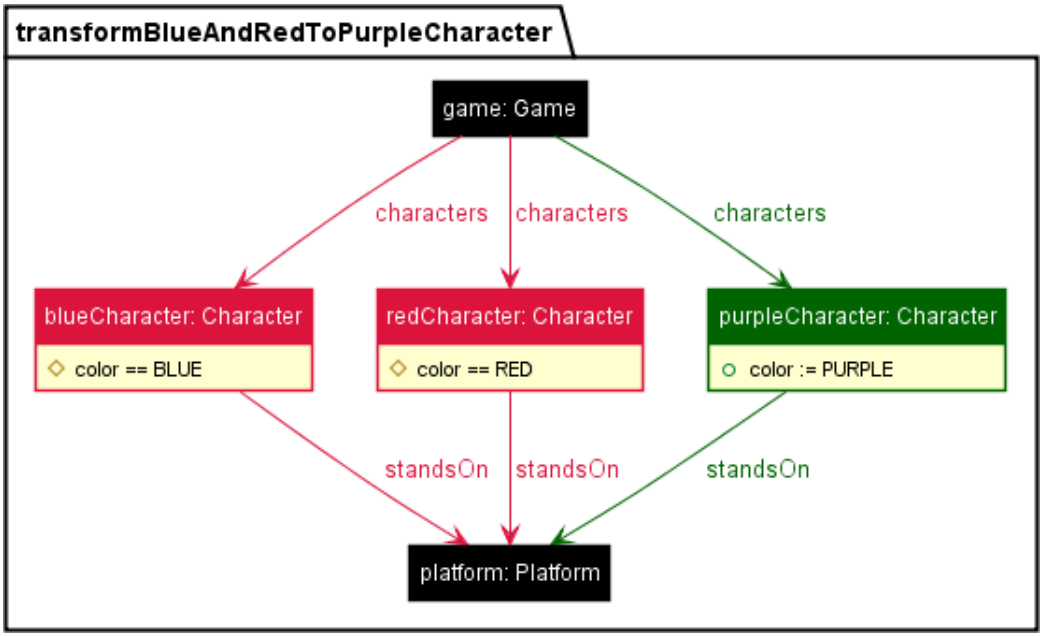


Figure 6.4: Rule `transformBlueAndRedToPurpleCharacter`

7 Evaluation

This chapter evaluates the implementation of eMoflon::IBeX-GT with respect to the following criteria:

- compliance with the requirements from Chapter 3, i. e. whether all requirements for the tool are met or not (see Section 7.1),
- correctness of the graph transformation API, i. e. whether the correct matches are found and the rules are applied as defined by the semantics or not (see Section 7.2),
- correctness of the validation of the rule specification in the textual editor, i. e. whether the user gets feedback for faulty specifications or not (see Section 7.3),
- performance/scalability depending on model size (see Section 7.4),
- and feedback on the usability of the eMoflon::IBeX-GT received from actual end-users (see Section 7.5).

7.1 Compliance with the Requirements

In Chapter 3 we presented a list of requirements which are checked for eMoflon::IBeX-GT in the following.

Incrementality
eMoflon::IBeX-GT uses an incremental pattern matching engine. Section 6.4 presents how the incrementality of the engine can be used to support incremental features in the API.

Interpreter
The eMoflon::IBeX-GT interpreter is used by the API to realize model queries and transformations. It can be used without the API if someone prefers to deal with type casting in his/her code directly.

Integration with a TGG tool
eMoflon::IBeX-GT is integrated into the eMoflon::IBeX tool suite which combines a GT and a TGG tool with some shared libraries. After refactoring the TGG part to use the IBeX pattern model as well (cp. Section 8.3) even larger parts of the code can be shared.

Mature integration with a general purpose language
The API is a typed interface for the graph transformation interpreter. This allows to a type-safe invocation of model queries and rule applications.

Dedicated support for model queries
Matches for patterns and rules can be queried via the API.

DPO or SPO semantics
The API uses with SPO semantics by default, which can be changed to DPO (see Section 6.3.3). The pushout approach can be set for the whole API, a rule or just a single rule application.

Modularity on rule level
Pattern refinement (see Section 5.2.4) has been implemented to support modularity on rule level.

Application conditions
The new graph transformation tool has mature support for positive and negative application conditions, which can be combined via OR and AND (with the only restriction that one must define the logical expressions in DNF).

Attribute manipulation
Possible values for attributes in eMoflon::IBeX are parameters, constants and other attributes, but not user-defined attribute values. We have decided to skip this advanced feature due to the time limit of this thesis. In addition, many examples do not require user-defined attribute values at all. However, the API integration allows to add arbitrary attribute assignments and constraints: Assignments can be realized via a subscription for rule applications (and setting the attribute value in the registered `Consumer`). Matches can be filtered for additional constraints including any arbitrary conditions on attribute values, e. g. using the `filter` method for `Streams`.

Textual concrete syntax and visualization
The patterns and rules are specified in a textual editor. A graphical visualization of the patterns and rules is provided, flattening the refinement hierarchy.

Modeling standard
eMoflon::IBeX-GT uses the modeling standard EMF.

End-user documentation
A handbook [AR18] introduces eMoflon::IBeX-GT based on an example application. The appendix contains a complete reference of all features of the pattern language and the API, intended for advanced users.

To summarize the evaluation with respect to our requirements list, all requirements are fulfilled except support for user-defined attribute values.

7.2 Correctness of Graph Transformation

To test for correctness, many JUnit tests on API level have been written to ensure that everything works as expected for many scenarios using different meta-models. Although tests can never guarantee correctness, a test suite is important to check that existing graph transformation specifications still work after the latest changes.

Table 7.1 gives an overview of the JUnit tests. Each package contains the tests for another API, defined in a graph transformation project of the same name. For each package the number of involved meta-models, the number of patterns and rules in the API and the number of test cases is given.

Test suite Test package	Meta-models	Patterns and rules	Test cases
org.ibex.emoflon.ibex.gt[1]			
BPMN	1	6	2
BPMNIR	2	4	2
ClassMultipleInheritanceHierarchy	1	16	7
FerrymanProblem	1	15	7
SheRememberedCaterpillars	1	29	18
SimpleFamilies	1	51	32
SimpleFamiliesToSimplePersons	2	5	1
SimpleFamiliesToSimplePersons2[2]	2	–	1
SimplePersons	1	9	2
de.upb.mbse.taxcalculationexample[3]			
businessrules.operationalsemantics	1	25	7
businessrules.structuralsemantics	1	6	10
cheat.rulestoreportstrafo	2	6	1
hot	2	2	1
rulestoreportstrafo	2	1	1

Table 7.1: JUnit Tests for Graph Transformation on API Level

The execution of the TestsuiteGT covers 89.7 % of the interpreter, the abstract API classes, and utilities for manipulating EMF models (shared with the TGG part via the Common project). The uncovered parts are mainly error cases in the transformation to Democles patterns and in the interpreter. In addition, the check for dangling edges (necessary for DPO applicability) needs more tests to handle all cases (the combinations of incoming/outgoing and containment/other references). Executing the JUnit test suite for eMoflon::IBeX-TGG covers all cases for the deletion utility methods. A good coverage for deletion is quite important, as the order of deleting nodes and references often caused errors in Democles.

Building all projects of TestsuiteGT covers 94.1 % of the code for the build of GT projects, which generates the API and the IBeX pattern model (excluding utility methods

[1] `https://github.com/eMoflon/emoflon-ibex-tests`, project TestsuiteGT
[2] using two separate APIs for both involved meta-models (SimpleFamilies API and SimplePersons API)
[3] JUnit tests written by Anthony Anjorin as examples for the MBSE lecture, `https://github.com/mde-lab-sessions/running-example-for-lecture`

from the Common project and generated code for the meta-models of IBeX patterns and the GT API). The uncovered parts are error cases which must not occur when building a project not containing syntax errors (such as the test suites).

7.3 Validation in the Textual Editor

In addition to the API and build code, the textual editor is unit-tested as well to ensure that faulty rule specifications are detected correctly: 111 test cases for scoping, validation, formatting, and the flattening of pattern refinement with a code coverage of 96.2 % yield a high confidence that the editor still discovers errors correctly after refactoring and the implementation of new features.

The tests for the editor mainly focus on cases the editor needs to report an error in scoping or validation, as the case that the textual specification does not contain any errors is implicitly tested when generating the code for the rules whose APIs are tested in the test suites described above: If there were any errors in any graph transformation project used in the API test suite, they would be reported upon code generation.

The integration into Eclipse (e. g. outline, syntax highlighting, visualization) is not checked with JUnit tests, but can be easily checked manually by opening some files.

The editor tests also help to speed up the development process: Normally changes in the editor validation require a restart of the runtime Eclipse application in which the validation rules can be checked manually. Implementing a test case first, it is possible to check the implementation without a restart of the runtime Eclipse application, as the JUnit test suite can be executed directly in the development workspace.

7.4 Performance and Scalability

Although performance and scalability are not the focus of this thesis, a small evaluation of runtimes for different model sizes is presented in the following. The performance is evaluated for model generation, model queries, and adding/deleting elements. All measurements were executed on a notebook with an Intel Core i-4510U processor (two cores, with hyper-threading four logical processors, with 2.0 GHz) and 16 GB main memory. The tests were executed on a Windows 8.1 system with Java version 8 Update 172 (64 bit). All runtimes are given as arithmetic mean of 100 test executions.[4]

As shown in Figure 7.1, the runtime for the generation of a new model with one game and x platforms (simple platform or exit, randomly chosen for each new platform) grows linear with x. Note that the time axis is logarithmic such that even small differences can be seen in the diagram. The number of model elements is limited by the available main memory, as the model and the found matches are kept in memory all the time. This result is expected due to the nature of the incremental pattern matcher. Note that the maximal model size which can be handled by eMoflon::IBeX-GT and the runtimes depend on the number of the observed patterns because in general observing more patterns will result in more matches, such that more main memory is required by the pattern matcher. For the She Remembered Caterpillars API with 29 patterns and rules, the tests for 320,000 elements needs nearly the whole main memory of the test environment. Trying to run the performance tests for 340,000 elements leads to an `OutOfMemoryError`.

[4]The source code for the performance tests can be found in the project TestsuiteGT as well.

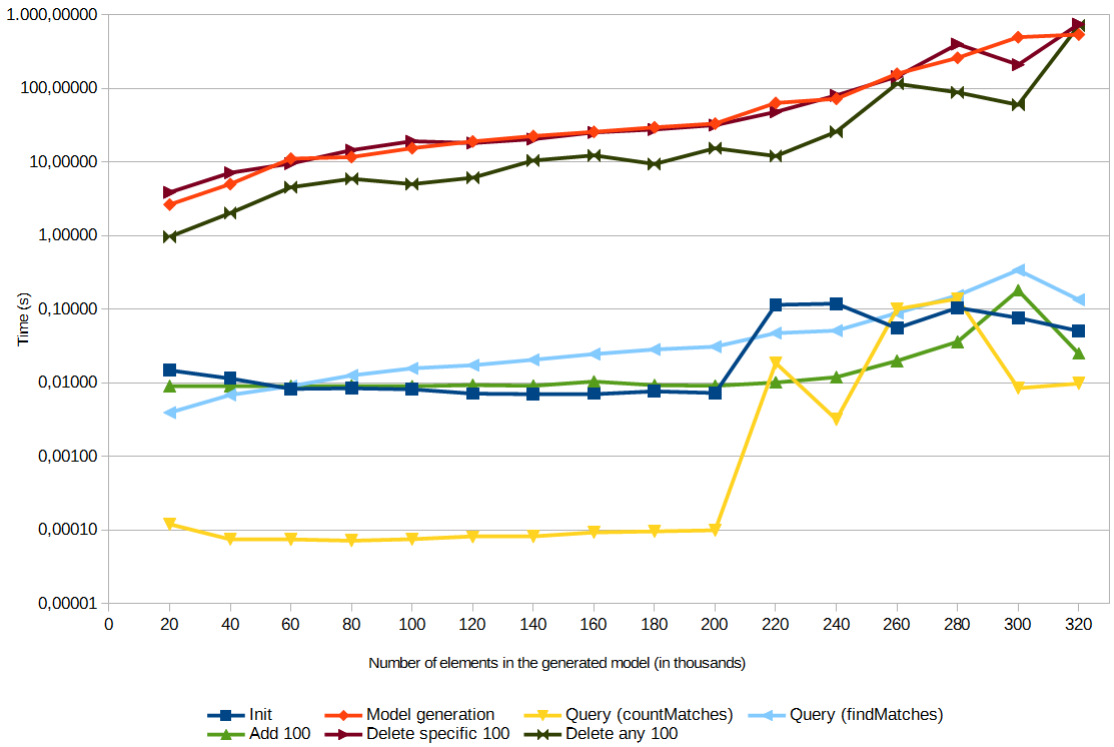


Figure 7.1: Runtime for Creation and Queries depending on Model Size

After creating the models as described above, the model is queried for all standalone (simple) platforms and empty exits with the patterns `findStandalonePlatform` and `findEmptyExit` known from Section 5.2.3. Queries for the number of matches require a constant time, while queries for all matches have a linear overhead due to the conversion to typed matches. Even for 320,000 matches the time is only 0.15 seconds.

As last insights, the time for adding 100 more platforms to the model of the given size and deleting them afterwards is shown. The time for adding the platforms is independent of the current model size. The deletion of the same 100 platforms is much more expensive, because this requires finding the matches for the pattern and filtering them for the ones with the respective platform (which will be exactly one in this case). As there are many platforms, it takes a while to find the match which bounds the platform to be deleted – the time for this step increases with the number of available matches, which is proportional to the model size in our example. Deleting any 100 platforms instead of 100 specific ones takes less time, as matches for the patterns must not be filtered for the ones with the correct node binding. The adding of elements also requires finding a match (in the pattern for creating a platform the game must be matched such that the containment edge between the game and the platform can be created), but as there is only one match this operation is faster than the deletion for which a lot of matches can be found.

For model sizes larger than 260,000 elements the measured runtimes vary a lot and the runtimes for model generation and deletion are not linear anymore.

For the performance tests presented in Figure 7.1, the API has been initialized with a resource set containing just one empty resource. In this scenario, Democles cannot find

any matches during the initialization (because the model contains no elements). Whenever the model is changed by creating new elements, new matches are found and added to the maintained set of matches.

Figure 7.2 illustrates the initialization of an API for loading existing models of different sizes (instead of starting with an empty model). In this case, Democles has to search for all matches of all patterns during the initialization – the larger the model, the more matches will be found. The time for updating the set of matches grows linear with the model size, while the time for the initialization seems to be worse than linear. After the initialization and the first update, the queries can be answered in constant time (plus a small overhead for the conversion to typed matches when all matches are returned) just as before.

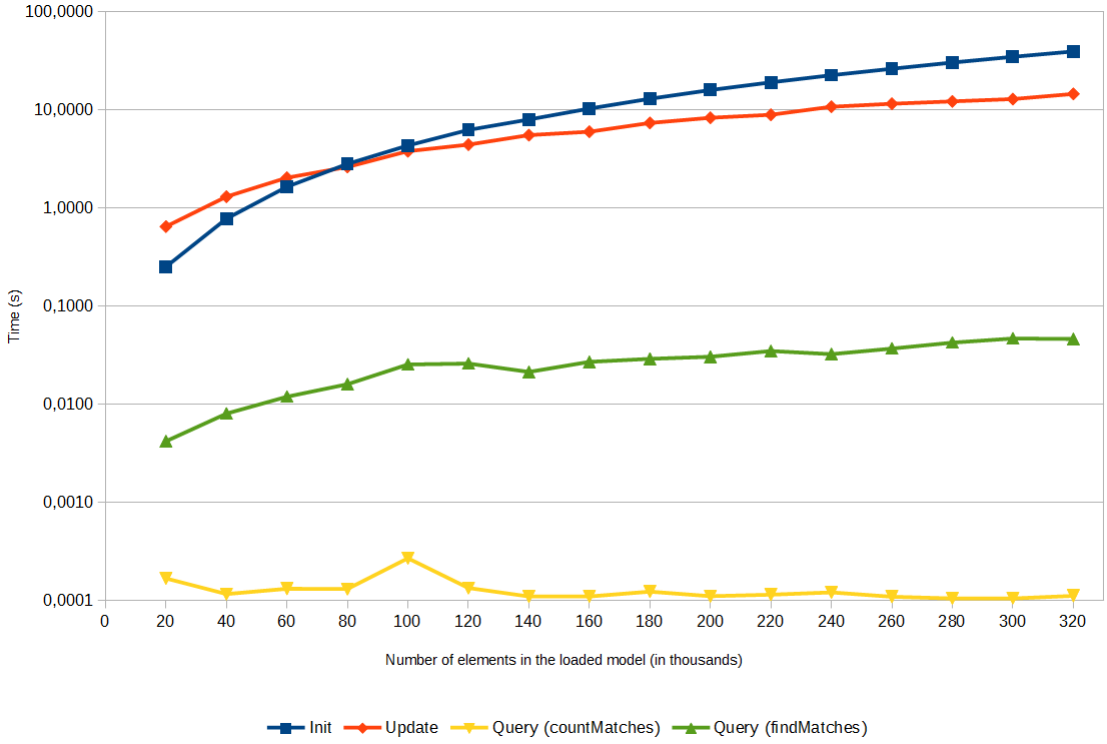


Figure 7.2: Runtime for Initialization and Queries depending on Model Size

As the measurements only use one meta-model and the model has a certain structure (one root element with a lot of children), further tests are necessary to check whether the results can be generalized for arbitrary models.

In addition to the model size, the total number of observed patterns has a significant impact on the runtimes, as more patterns lead to larger times for searching the pattern structures and more matches to maintain. The dependency on the size and complexity of the observed patterns (e. g. number of nodes and edges in a pattern, usage of application conditions) has not been evaluated yet.

7.5 Usability and End-User Feedback

As eMoflon::IBeX-GT is intended to be used for teaching purposes, students using the tool in the lecture "Model-Based Software Engineering" (MBSE) and students writing their master's theses in the area of graph transformation were asked to give feedback via an online questionaire[5] being a mix of multiple choice and open questions.

7.5.1 Experience of the participants

20 students with a different level of experience with graph transformation, all members of our intended target group, participated in the survey (cp. Figure 7.3). Most of the participants have no or only little experience with graph transformation and model-driven engineering in general. With one exception, all participants have programming experience and used the Eclipse IDE before. Most participants are not familiar with visual (programming) languages.

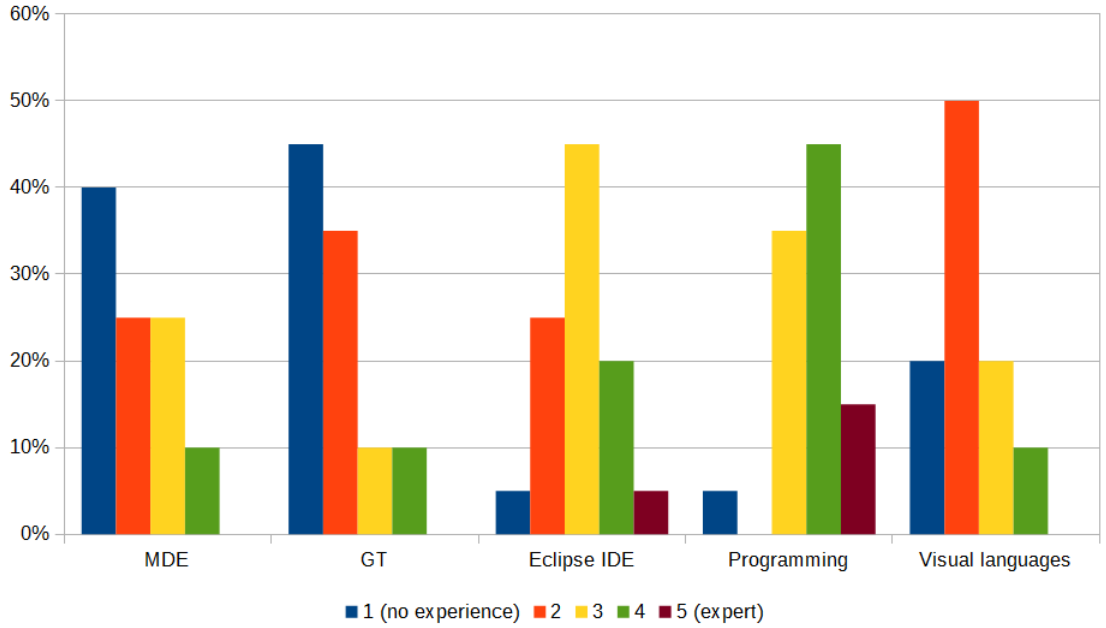


Figure 7.3: Experience of the Participants

7.5.2 Textual and Visual Syntax

Figure 7.4 shows how the end-users rated the textual and visual syntax. Most users think that the textual syntax is understandable (average 3.6 out of 5 stars, 90 % gave 3 or 4 stars). Comments on the syntax are mostly positive, and highlight the visualization and the intuitive syntax. For example, participants stated that the textual syntax "easy to understand and write down" and that "the ++ and −− is intuitive, and there are no unnecessary chars like ;".

[5] https://docs.google.com/forms/d/1r5pgkTv0CcvTQoqHlUuHcQqULl6A8CmbHgpt961BtHU

The participants especially like the mix of the textual and the visual syntax (average 4 stars, 70 % gave 4 or 5 stars). The visualization (average 4.0 stars) is considered to be more helpful than the error messages (average 3.65 stars). The usability of the editor as a whole is rated with 3.45 out of 5 stars on average, 90 % giving 3 or more stars.

Suggestions for improvements of the editor mainly focus on the visualization. 30 % would like to edit the visual syntax directly, which has not been implemented due to the complexity of visual editors (cp. the requirements). They also point out that the overlapping of elements is an issue for large patterns. As the layout of the visualization is handled by PlantUML, there seems to be only little potential for improvements with the current choice of visualization tool.

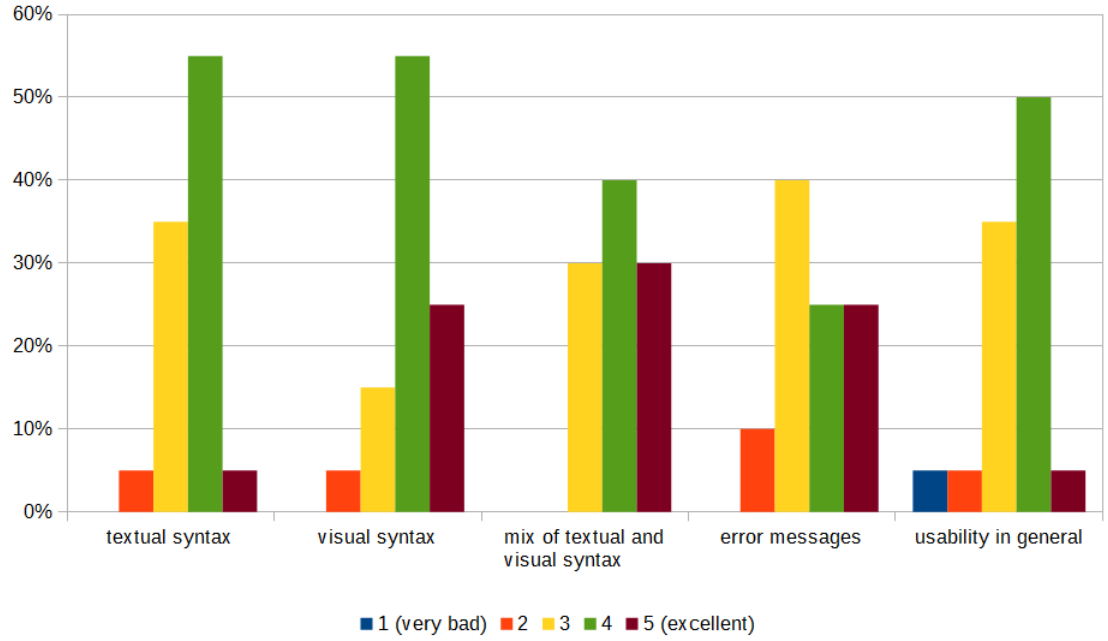


Figure 7.4: Evaluation of Textual and Visual Syntax

7.5.3 Language Features

The users' estimation regarding the importance of selected language features is shown in Figure 7.5. Complex graph conditions are the feature with the highest average rating (3.7), followed by application conditions (3.65) and attributes (3.55). Pattern refinement and incrementality[6] are considered to be less important (2.9), which is surprising for us.

In our requirements list (Chapter 3) we have considered incrementality and pattern refinement as the most important features, especially as there are no other graph transformation tools with support for those features. One reason for this surprising result might be that the students asked for feedback only have a little experience with the tool and have not used all features in their own specifications yet. Another important point is that

[6]The incremental features are called "support for reactive programming" in the survey because the participants of the survey do not know the details of the implementation based on incremental pattern matching.

both features make an impact for large applications maintained and evolved over a long time.

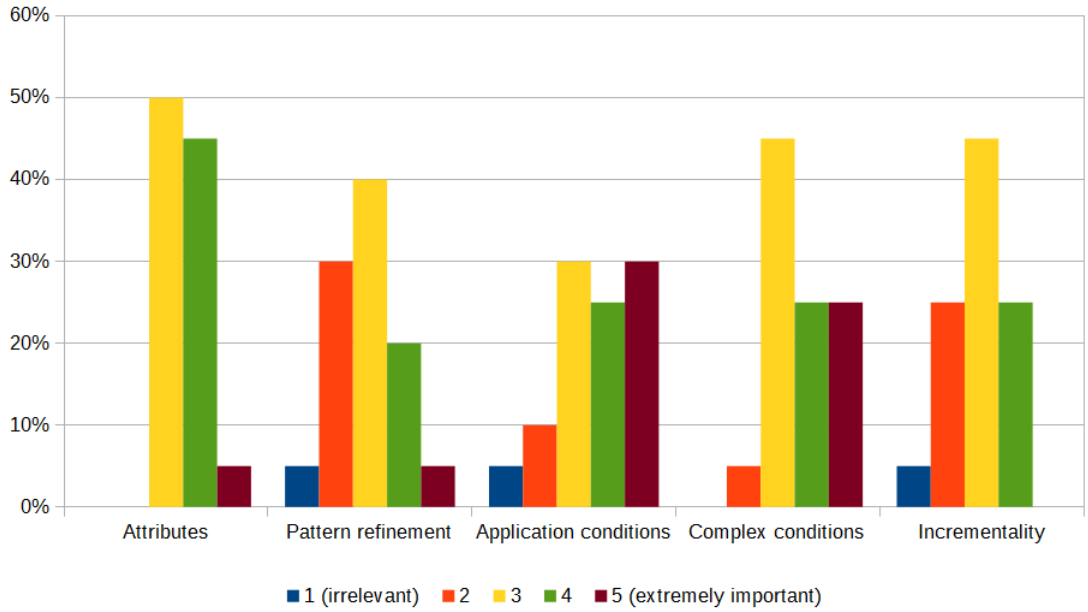


Figure 7.5: Importance of Language Features

7.5.4 Potential for the Usage in Java Applications

The users have been asked to comment on the integration with Java code. Suggestions for improvements focus on a possibility to jump from the generated Java classes to the specification via one click ("Make it possible to directly navigate to the patterns/rules from the code"), although switching between `gt` files and the Java classes is a quite seamless experience from the point view of 35 % of the participants, and average for another 55 % of them. As this gets more complex when the specification consists of many `gt` files, a linking between editor patterns/rules and Java classes is planned for the future.

Most users think that the pattern language is expressive enough to specify patterns of realistic complexity in practice (average 3.55 out of 5 stars). Almost a two-third majority voted with 4 or 5 stars.

Asked for the main arguments for using eMoflon::IBeX-GT in a Java project many students named the visualization, as this makes the specification understandable for non-programmers. Furthermore, one user commented: "Using patterns is closer to the underlying model, easier to adapt to changes, less error-prone and much less work than hard-coding the operations on the EMF model". Drawbacks in the users' opinion are performance, a more complex project setup (considered to be costly especially for small projects) and a missing debugging facility. Actually, debugging is possible via analyzing the pattern specifications and the received matches.

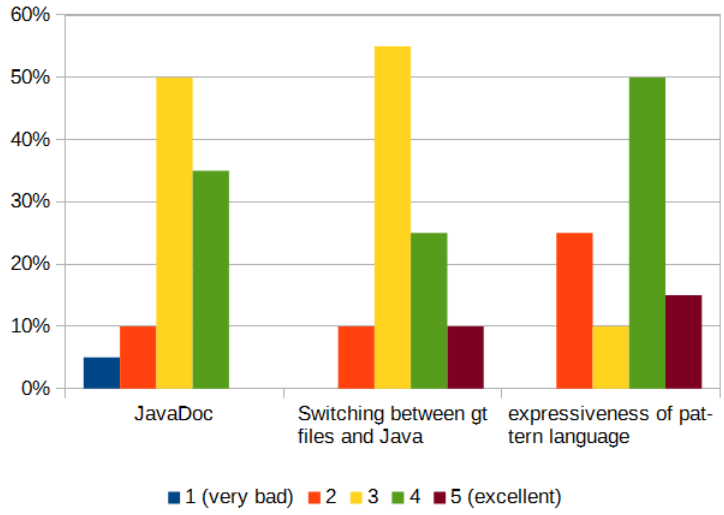


Figure 7.6: Evaluation of the Potential for the Usage in Java Applications

7.5.5 Handbook

The handbook [AR18] introduces eMoflon::IBeX-GT by developing an example application ("Sokoban") step by step. In an appendix all features of the pattern language and the API are explained, as not all of them are used in the example. The students enjoyed working through the handbook ("fun factor" average 4.0 out of 5 stars). The appendix is rated as average by most students. The students were asked for feedback after working through the tutorial, which does not require the appendix. The pattern and API references in the appendix are considered for advanced users who need to lookup some details.

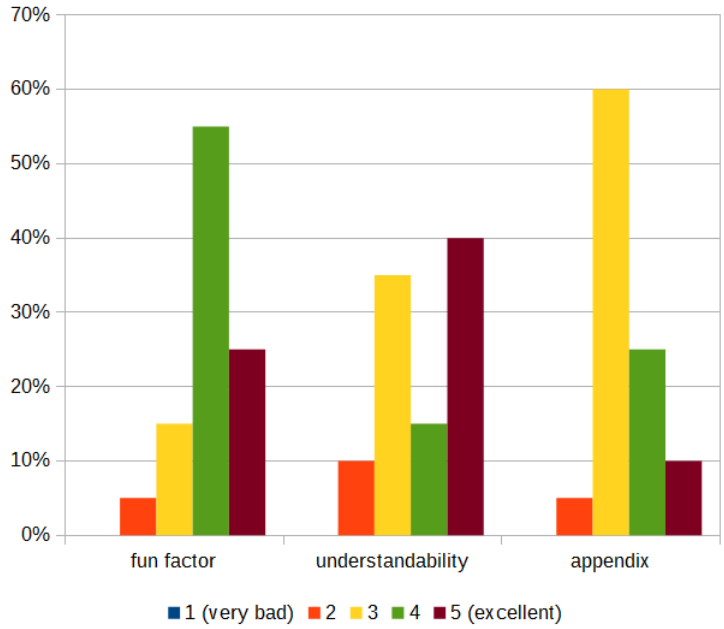


Figure 7.7: Evaluation of the Handbook

8 Conclusion and Future Work

This thesis presented eMoflon::IBeX-GT, a new graph transformation tool with a Java API exploiting the incrementality of its underlying pattern matching engine, Democles. It provides mature support for all requirements from Chapter 3 with some minor restrictions (cp. Sections 7.1 and 8.5). In Chapter 4 we have seen that no other GT tool supports all requirements.

8.1 Evaluation of Performance

As performance is not the focus of this work, performance has only been evaluated shortly with respect to scalability for models of different sizes. The evaluation in Section 7.4 clearly indicates that the maximal model size which can be handled by eMoflon::IBeX is limited by the available main memory. It also shows that the time for creation is linear to the number of created elements, but the time for answering queries on the model is nearly constant. The incrementality helps to reach a constant response time for model queries, as the matches are not searched on demand, but maintained permanently. However, queries with bound parameters seem to have a worse performance as filtering the matches reported by Democles is necessary.

8.2 Optimization of the Pattern Network

Due to a missing detailed evaluation of the performance, optimizing the pattern network generated for the graph transformation specification with respect to performance remains as future work. The insights for optimizing the patterns of eMoflon::IBeX-TGG by Weidmann (see [Wei18], Section 5.2) could be a useful starting point for the optimization. eMoflon::IBeX-TGG has configuration flags for the different possible patterns (e. g. whether all edges are extracted into an invoked pattern) such that the user can evaluate which configuration is best for the used TGG rules. A similar approach could be implemented for GT, hopefully improving the performance for many pattern specifications.

Currently the transformation from the editor model into the IBeX model does not exploit the refinement hierarchy. Instead the patterns are flattened into a structure without refinement before the transformation. It could be checked whether the refinement hierarchy can be exploited for the pattern network to improve the performance. However, a similar approach for the TGG part [Sto17] has not shown significant performance improvements.

8.3 Shared Patterns with eMoflon::IBeX-TGG

In future, IBeX patterns should be shared with the TGG part of eMoflon::IBeX. The IBeX pattern model is designed such that this should be easily possible. Just for attributes the IBeX model cannot handle the complexity of attribute constraints necessary

for eMoflon::IBeX-TGG such that a little extension of the IBeX model is required. Technically, the following steps are necessary to use the IBeX pattern model in the TGG part as well:

- the extension of the IBeX pattern meta-model such that attribute constraints can be specified in the complexity necessary for TGG specifications (especially user-defined attribute constraints),
- the transformation of the internal TGG model to the IBeX pattern model,[1]
- the adaption of the transformation from the IBeX pattern model to Democles patterns (and the initialization of the Democles patterns in the `DemoclesGTEngine`) to handle the additional attribute constraints (after that the transformation from `IBlackPatterns` to Democles patterns in the TGG part can be removed),
- the removal of the pattern initialization in the `DemoclesTGGEngine` (because this can be inherited from the `DemoclesGTEngine` as soon as both engines are initialized with IBeX patterns),
- and the adaptation of the green and red interpreter for TGGs (handling creation and deletion during rule applications), their interfaces, and the operational strategies using them.

Maybe the current `IbexGreenInterpreter` for TGGs can be even replaced with the `GraphTransformationCreateInterpreter`, if create patterns are generated in the IBeX pattern model. For the red interpreter an own implementation for the TGG part will remain necessary, as deletion for TGGs is undoing a previous rule application instead of applying the deletions specified by a delete pattern.

8.4 Applications using Graph Transformation and TGG

Besides the integration of eMoflon::IBeX-GT and eMoflon::IBeX-TGG from the perspective of the eMoflon::IBeX development team, both parts can also be used together by end-users. An example is the usage of graph transformation for preprocessing a model synchronized with another model via a TGG.[2] Currently this is possible, but requires some knowledge how the pattern matcher works.

After eMoflon::IBeX has been migrated to the IBeX pattern model, it may be possible to use only one Democles engine instance in an application using a GT API and a TGG in order to save memory and sharing common patterns (e. g. edge patterns). Currently the GT API and the operationalization for the TGG use two different engines.

8.5 Expressiveness of the Graph Transformation Rules

The specification of attribute assignments and conditions is currently restricted to a subset of the theoretical possibilities. Regarding attributes, the reason is a lack of time during the

[1] Alternatively a direct transformation from the editor TGG model to the IBeX pattern model could be implemented, but this would be a huge step, as the IBeX pattern model completely abstracts from TGGs.

[2] The eMoflon::IBeX-TGG handbook [Anj18] provides an example for GT as a preprocessor.

implementation of this thesis. In the future it could be evaluated whether the implemented subset is satisfying for all use cases. However, the API allows to set arbitrary attribute values after rule applications (e. g. via a subscription of rule applications with a `Consumer` calculating and setting the attribute value in self-written Java code). After eMoflon::IBeX-TGG has been refactored to use IBeX patterns as well, the IBeX model needs to support more complex attribute conditions. So just the editor and the transformation from editor patterns to IBeX patterns has to be extended.

The support for complex graph conditions is restricted to expressions in DNF, which does not limit the expressiveness. In general, this can lead to longer logical expressions the user has to define. Technical limitations of the currently used pattern matching engine are the reason for this decision, as Democles cannot handle alternatives. As described in Section 5.2.3.3, the matches for the alternatives are collected separately and merged by the interpreter. This approach ensures that as much as possible of the work is done by the pattern matcher, which ensures that invalid matches are filtered out in the first step already. In the case that support for alternatives is implemented in Democles, the implementation could be adjusted to check the alternatives directly in Democles which avoids combining and removing duplicates in the graph transformation interpreter.

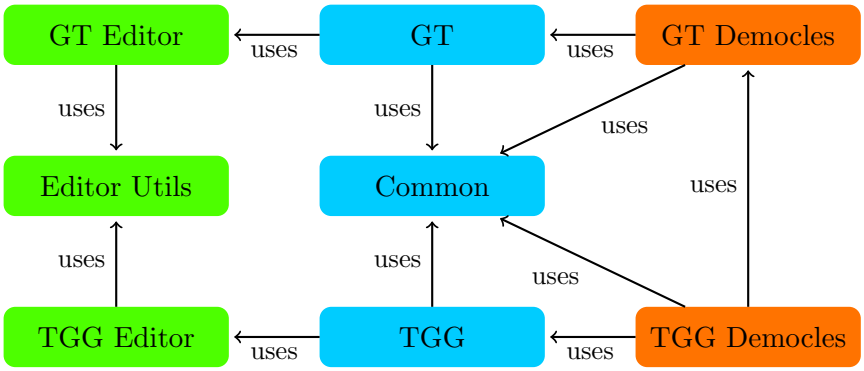
If it turns out that the restriction to expressions in DNF is disturbing of many users, one could allow arbitrary logical expressions in the editor and transform the expression into the DNF before the patterns are transformed.

8.6 Improvements to the Editor

The feedback of end-users pointed out that a link between editor patterns/rules and Java classes would be helpful (cp. Section 7.5.4). A visualization is intentionally not editable according to our requirements list, as this requires a more complex implementation and the handling of the layout. Although some end-users have suggested an editable visual syntax, our initial consideration has not changed.

Currently the syntax for the textual specification in the GT and TGG editor and the formatting algorithms are not completely consistent. This should be adjusted for full consistency between GT and TGG.

A List of Projects



(cp. Section 5.1)

Repository	Component	Plugin Project
emoflon-ibex-ui[1]	Editor Utils	`org.emoflon.ibex.editor.utils`

	GT Editor	`org.emoflon.ibex.gt.editor`
		`org.emoflon.ibex.gt.editor.ide`
		`org.emoflon.ibex.gt.editor.ui`

	TGG Editor	`org.emoflon.ibex.tgg.editor`
		`org.emoflon.ibex.tgg.editor.ui`
		`org.emoflon.ibex.tgg.ide`

emoflon-ibex[2]	Common	`org.emoflon.ibex.common`
	GT	`org.emoflon.ibex.gt`
	TGG	`org.emoflon.ibex.core.language` `org.emoflon.ibex.core.runtime`
emoflon-ibex-democles[3]	GT Democles	`org.emoflon.ibex.gt.democles`
	TGG Democles	`org.emoflon.ibex.tgg.ide.democles` `org.emoflon.ibex.tgg.runtime.democles`

Table A.1: List of Projects in the eMoflon::IBeX Tool Suite

[1] `https://github.com/eMoflon/emoflon-ibex-ui`
[2] `https://github.com/eMoflon/emoflon-ibex`
[3] `https://github.com/eMoflon/emoflon-ibex-democles`

B List of Figures

C List of Tables

D Listings

E Bibliography

[AKK+17a] Anthony Anjorin, Roland Kluge, Géza Kulcsar, Erhan Leblebici, Lars Fritsche, and Gergely Varró. An Introduction to Metamodelling and Graph Transformations with eMoflon, 2017. Available at `https://github.com/eMoflon/emoflon-docu/releases/download/emoflon_2.32.0/eMoflonHandbook.pdf`. Retrieved 2018-03-07.

[AKK+17b] Anthony Anjorin, Roland Kluge, Géza Kulcsar, Erhan Leblebici, Lars Fritsche, and Gergely Varró. eMoflon - a tool for building tools, 2017. Available at `http://emoflon.org/`. Retrieved 2018-03-07.

[Anj18] Anthony Anjorin. Bidirectional Model Transformation with eMoflon::IBeX, 2018. Available at `https://paper.dropbox.com/doc/GxyQmS2198CgxBhOCj8Hv`.

[AR18] Anthony Anjorin and Patrick Robrecht. Unidirectional Model Transformation with eMoflon::IBeX, 2018. Available at `https://paper.dropbox.com/doc/siVjGl9SaMSuBnBYEv6cG`.

[AVS12] Anthony Anjorin, Gergely Varró, and Andy Schürr. Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques. In Frank Hermann and Janis Voigtländer, editors, *Proceedings of the 1st International Workshop on Bidirectional Transformations (BX 2012), Tallinn, Estonia, March 25, 2012*, 2012. Available at `https://pdfs.semanticscholar.org/b500/bf90a2d00040894da69c876eafe64ed20602.pdf`. Retrieved 2018-07-02.

[BCvC+13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A Survey on Reactive Programming. In *ACM Computing Surveys (CSUR), Volume 45 Issue 4, August 2013*, New York, 2013. ACM. Available at `http://soft.vub.ac.be/Publications/2012/vub-soft-tr-12-13.pdf`. Retrieved 2018-07-06.

[BHK+16a] Kristopher Born, Frank Hermann, Timo Kehrer, Christian Krause, Daniel Strüber, and Matthias Tichy. Henshin, 2016. Available at `https://www.eclipse.org/henshin/`. Retrieved 2018-03-07.

[BHK+16b] Kristopher Born, Frank Hermann, Timo Kehrer, Christian Krause, Daniel Strüber, and Matthias Tichy. Henshin, 2016. Available at `https://wiki.eclipse.org/Henshin`. Retrieved 2018-03-07.

[BHRV08] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer,

editors, *Proceedings of the 4th International Conference on Graph Transformations (ICGT 2008), Leicester, United Kingdom, September 7-13, 2008*, pages 396–410, Berlin, Heidelberg, 2008. Springer. Available at `https://link.springer.com/chapter/10.1007/978-3-540-87405-8_27`. Retrieved 2018-03-07.

[CFH+08] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective, 2008. Available at `http://www.cs.cornell.edu/~jnfoster/papers/grace-report.pdf`. Retrieved 2018-03-07.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches, 2006. Available at `https://pdfs.semanticscholar.org/7eca/ca8db190608dc4482999e19b1593cc6ad4e5.pdf`. Retrieved 2018-05-26.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, Berlin, Heidelberg, 2006.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation – Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation – Volume 1*, pages 247–312, Singapore, 1997. Worlds Scientific Publishing Co. Pte. Ltd.

[HMS05] Jan Heering, Marjan Mernik, and Anthony M. Sloane. When and how to develop domain-specific languages. In *ACM Computing Surveys (CSUR) Surveys Homepage archive Volume 37 Issue 4*, pages 316–344, New York, 2005. ACM. Available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.4178&rep=rep1&type=pdf`. Retrieved 2018-03-07.

[HP16] Ivaylo Hristakiev and Detlef Plump. Attributed Graph Transformation via Rule Schemata: Church-Rosser Theorem, 2016. Available at `https://link.springer.com/book/10.1007/978-3-319-50230-4`. Retrieved 2018-07-02.

[HVRU17] Abel Hegedus, Daniel Varro, Istvan Rath, and Zoltan Ujhelyi. VIATRA 1.6 is out with Eclipse Oxygen, 2017. Available at `https://www.eclipse.org/viatra/`. Retrieved 2018-03-07.

[JBG17a] Edgar Jakumeit, Jakob Blomer, and Rubino Geiß. GrGen.NET, 2017. Available at `http://www.info.uni-karlsruhe.de/software/grgen/`. Retrieved 2018-03-07.

[JBG17b] Edgar Jakumeit, Jakob Blomer, and Rubino Geiß. The GrGen.NET User Manual, 2017. Available at `http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf`. Retrieved 2018-03-07.

[KC15a] Nafiseh Kahani and James R. Cordy. Comparison and Evaluation of Model Transformation Tools, 2015. Available at http://research.cs.queensu.ca/TechReports/Reports/2015-627.pdf. Retrieved 2018-03-07.

[KC15b] Nafiseh Kahani and James R. Cordy. Comparison of Model Transformation Tools, 2015. Available at http://www.mdetools.com/. Retrieved 2018-03-07.

[Koz16] Sergejs Kozlovics. Models and Model Transformations Within Web Applications. In *Proceedings of the 12th International Baltic Conference (DB & IS 2016), Riga, Latvia, July 4-6, 2016*, pages 53–67, Cham, 2016. Springer International Publishing. Available at https://link.springer.com/chapter/10.1007/978-3-319-40180-5_4. Retrieved 2018-03-07.

[Kur05] Ivan Kurtev. Building adaptable and reusable XML applications with model transformations, 2005. Available at https://dl.acm.org/ft_gateway.cfm?id=1060772&ftid=314066&dwn=1&CFID=1012139749&CFTOKEN=52595033.

[Obj15] Object Management Group. About the Unified Modeling Language Specification Version 2.5, 2015. Available at http://www.omg.org/spec/UML/. Retrieved 2018-03-07.

[PGH+16] David Priemer, Tobias George, Marcel Hahn, Lennert Raesch, and Albert Zündorf. Using Graph Transformation for Puzzle Game Level Generation and Validation. In *Proceedings of the 9th International Conference (ICGT 2016), Vienna, Austria, July 5-6, 2016*, pages 223–235, Cham, 2016. Springer International Publishing. Available at https://link.springer.com/content/pdf/10.1007%2F978-3-319-40530-8_14.pdf. Retrieved 2018-05-26.

[RdMZ17] Arend Rensink, Maarten de Mol, and Eduardo Zambon. GROOVE - GRaphs for Object-Oriented VErification, 2017. Available at http://groove.cs.utwente.nl/. Retrieved 2018-03-07.

[Run06] Olga Runge. The AGG 1.5.0 Development Environment. The User Manual, 2006. Availbale at http://www.user.tu-berlin.de/o.runge/agg/AGG-ShortManual/AGG-ShortManual.html. Retrieved 2018-03-07.

[Run17] Olga Runge. The Attributed Graph Grammar System: A Development Environment for Attributed Graph Transformation Systems, 2017. Available at http://www.user.tu-berlin.de/o.runge/agg/. Retrieved 2018-03-07.

[Sch95] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching, Germany, June 16-18, 1994*, pages 151–163, Berlin, Heidelberg, 1995. Springer-Verlag. Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.457.5219&rep=rep1&type=pdf. Retrieved 2018-05-26.

[Sol12] Solunar GmbH. EMorF documentation, 2012. Available at http://www.emorf.org/doc.html. Retrieved 2018-03-07.

[Sto17] Florian Stolte. Exploiting the modularity of triple graph grammars via incremental pattern matching techniques, 2017.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. John Wiley & Sons Ltd, Chichester, 2006. Available at `http://www.voelter.de/data/books/mdsd-en.pdf`.

[SVM+16] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, Huseyin Ergin, and Jonathan Corley. AToMPM Documentation, 2016. Available at `https://msdl.uantwerpen.be/documentation/AToMPM/`. Retrieved 2018-03-07.

[VAS12] Gergely Varró, Anthony Anjorin, and Andy Schürr. Unification of Compiled and Interpreter-Based Pattern Matching Techniques, 2012. Available at `http://tuprints.ulb.tu-darmstadt.de/2922/1/29220.pdf`.

[VVS06] Gergely Varró, Daniel Varró, and Andy Schürr. Incremental Graph Pattern Matching: Data Structures and Initial Experiments. In *Proceedings of the Second International Workshop on Graph and Model Transformation (GraMoT 2006), Brighton, United Kingdom, September 8, 2006*, Berlin, 2006. Electronic Communications of the EASST. Available at `https://journal.ub.tu-berlin.de/eceasst/article/view/12/4`, detailed version with appendix: `http://www.cs.bme.hu/~gervarro/publication/IncrementalEngine.pdf`. Retrieved 2018-03-07.

[Web16] Jens H. Weber. Grape - Graph Rewriting And Persistence Engine, 2016. Available at `https://jenshweber.github.io/grape/` Retrieved 2018-03-07.

[Web17] Jens H. Weber. GRAPE – A Graph Rewriting and Persistence Engine. In Juan de Lara and Detlef Plump, editors, *Proceedings of the 10th International Conference (ICGT 2017), Marburg, Germany, July 18-19, 2007*, pages 209–220, Cham, 2017. Springer International Publishing. Available at `https://link.springer.com/chapter/10.1007/978-3-319-61470-0_13`.

[Wei18] Nils Weidmann. Consistency Management via a Combination of Triple Graph Grammars and Integer Linear Programming, 2018.