Electronic Communications of the EASST
Volume 6 (2007)

Proceedings of the
Sixth International Workshop on
Graph Transformation and Visual Modeling Techniques
(GT-VMT 2007)

The Jury is still out: A Comparison of AGG, Fujaba, and PROGRES

Christian Fuss, Christof Mosler, Ulrike Ranger, and Erhard Schultchen

14 pages

Guest Editors: Karsten Ehrig, Holger Giese
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

The Jury is still out: A Comparison of AGG, Fujaba, and PROGRES

Christian Fuss, Christof Mosler, Ulrike Ranger, and Erhard Schultchen

[fuss|mosler|ranger|schultchen]@i3.informatik.rwth-aachen.de
http://www-i3.informatik.rwth-aachen.de
Department of Computer Science 3 (Software Engineering)
RWTH Aachen University, Germany

Abstract: Graph transformation languages offer a declarative and visual programming method for software systems with complex data structures. Some of these languages have reached a level of maturity that allows not only conceptual but also practical use. This paper compares the three widespread graph transformation languages AGG, Fujaba, and PROGRES, considering their latest developments. The comparison is three-fold and regards conceptual aspects, language properties, and infrastructure features. Because of the different relevance of these aspects, we do not determine a clear winner but leave it to the reader.

Keywords: Graph Transformation Languages, AGG, Fujaba, PROGRES

1 Introduction

Graph transformation languages are one branch of visual programming languages, which provide advanced concepts for modeling software tools. Some languages reached a level of maturity that allows utilization in practice. In this paper, we compare the three widespread general-purpose languages AGG, Fujaba, and PROGRES. Our goal is to point out main differences in conceptual aspects, language properties, and infrastructure features. There exist some comparisons of graph transformation languages dating back several years, e.g. [BTMS99, FNT98, Roz97]. Comparisons that are more recent focus only on particular application areas, e.g. [Agr04] deals with model integration aspects. In [VSV05], the runtime efficiency of the generated applications is analyzed. While this paper concentrates on general-purpose languages, [MG05, TEG$^+$05] describe also languages dedicated to model transformation and to model checking, e.g. GReAT [KASS03] and GROOVE [Ren04].

We do not only want to update the older comparisons, considering recent developments, but also lay focus on practical aspects. We do not claim our comparison is complete. This paper shall support users when deciding, which language is most appropriate for his or her application. We hope especially a novice in the area of graph transformation languages will profit from this practical overview.

This paper is structured as follows: In Section 2, we describe the aspects of the graph transformation languages that we examine and introduce a running application example. In the following sections, we study how each of the languages AGG, Fujaba, and PROGRES meets the stated requirements and describe specific aspects. Finally, Section 6, summarizes our comparison and points out strengths and weaknesses of each language.

2 Compared Aspects

The graph transformation languages are compared by different aspects, which are introduced in this section. We compare the theoretical concepts building each language's background, the language features when specifying a graph transformation system, and the infrastructure offered to edit and run these systems.

2.1 Theoretical Concepts

Graphs are clear and intuitive data structures, whose fundamentals are mathematically founded. Since the late 1960s, different approaches to *graph grammars* have been developed, which differ e.g. in their graph model, the expressiveness of transformation rules, and the definition of semantics. Basically, two main approaches can be distinguished, which are briefly described in the following.

The *algebraic approach* considers a graph as a 2-sorted algebra, where nodes and edges are typed, attributed, and identified. The *derivation* of a graph by applying a graph transformation rule is defined by *pushouts* known from category theory. The approach allows formal and easy to understand proofs of properties on graphs and on graph transformation rules, e.g. the amalgamation of graph transformation rules. Two different branches concerning the derivation have been evolved within the algebraic approach, namely the *double pushout approach* (DPO) [CEH$^+$97] and the *single pushout approach* (SPO) [EHK$^+$97]. In DPO, a derivation is constructed by two pushouts using a gluing graph between the left-hand side and the right-hand side of a transformation rule, which enables to reverse transformations. A graph transformation rule can only be applied if all edges incident to its match in the working graph and to the context graph are specified within the transformation rule, which leads to complex specifications. For this reason, SPO has been developed, which overcomes this restriction and constructs only one pushout for a derivation. Thus, the graph transformation rules are easier, but the theoretical properties of SPO are limited.

The *set-theoretic approach* [Nag79] offers an intuitive understanding of graph transformation systems, but does not provide a theoretical foundation that is as powerful as in the algebraic approach. Graphs are described as sets of nodes and edges and the effect of applying a graph transformation rule is defined by set-theoretic operations. In contrast to the algebraic approach, edges are considered as relations between nodes and thus are neither identified nor attributed. The approach allows more expressiveness within graph transformation rules, e.g. embedding rules, which enable user-defined embedding of a rewritten sub-graph in its context graph. Furthermore, the application of graph transformation rules can be managed by control structures offering a *backtracking* mechanism for determining matches of transformation rules. The approach does not provide any means for describing static and derived graph properties. These aspects are integrated in the *logic-oriented approach* [Sch91], which is an enhancement of the set-theoretic approach. The approach allows to define an explicit graph schema and uses *predicate logic formulas* for defining graphs and graph transformation rules.

Besides the fundamental approach, a graph language may be based on different programming paradigms. As all presented graph languages offer means for typing graph elements, we will analyze in how far they support the object-oriented paradigm. This includes providing type-specific attributes and methods, inheritance relations between types, and polymorphism.

2.2 Language Properties

In this subsection, we examine properties of graph transformation languages, concerning the *graph model* and *graph transformations*, in general. Some of the properties are similar to those compared in [BTMS99], some are owed to new developments in AGG, Fujaba, and PROGRES. Table 1 shows a feature matrix listing all properties. The sections on AGG, Fujaba, and PROGRES describe properties implemented for each language in detail.

Table 1: Feature matrix with language properties

Property			AGG	Fujaba	PROGRES
Graph Model	graphs	kind	directed, attributed, labeled	directed, attributed, labeled	directed, attributed, labeled
		graph schema	unchecked type graph	UML class diagram	graph schema with static rule-check
		integrity constraints	global event-condition rules with manual application	—	global and node-local ECA rules, schema constraints
	nodes	kind	typed, attributed, identified	typed, attributed, identified	typed, attributed, identified
		derived node types	multiple inheritance	[multiple] inheritance	multiple inheritance
	edges	kind	labeled, attributed, identified, directed, binary, between nodes	labeled, directed, binary, between nodes	labeled, directed, binary, between nodes
		derived edge types	—	paths (textual)	paths (materializable)
		constraints	—	ordered	—
	attributes	value types	Java objects/standard types	Java objects/standard types, node types	internal standard types, C types, node types, sets
		expressions	parsed Java expressions	unparsed Java expressions	parsed C or PROGRES expressions
		derived attributes	—	simulated using methods	directed equations
		meta attributes	—	const, static	const, static
Transformations	matching	homo-/isomorphic	global option	explicit folding per rule element	explicit folding per rule element
		multiple matches	[amalgamated subrules]	set nodes, for-each patterns	set nodes, star rules
	conditions	subgraphs	nodes, edges	nodes, optional nodes, set nodes, edges, paths, constraints	nodes, optional nodes, set nodes, edges, paths, restrictions, constraints
		NACs	neg. subgraphs	neg. nodes, neg. edges, neg. constraints	neg. nodes, neg. edges, neg. paths, neg. constraints
		attribute conditions	yes	yes	yes
	gluing/embedding		gluing	—	embedding
	signature		in parameters	in parameters, return value	in/out parameters
	control programming	mechanisms	iteration over layers	conditional, iteration, sequence, collaboration stmts, method calls	conditional, iteration, sequence, non-deterministic choice, transformation calls
		transactions	—	—	yes
		backtracking	—	—	yes

Graph Model

Graphs. The working graphs of all discussed languages are *directed, attributed, node- and edge-labeled*. The structure of the working graphs is constrained by *graph schemas* that define node and edge types and their relations. Transformation rules should be checked against the schema to avoid syntactical errors at specification time. *Integrity constraints* are used to prohibit certain patterns in the working graph. They are checked at runtime. Transformations of *hierarchical graphs* can be found in literature but are not implemented in any of the languages.

Nodes. In the three languages, nodes are generally *typed, attributed, and identified* elements. Node types can be derived from other types by *inheritance*.

Edges. Edges are typed, directed and connect two nodes in all three languages. Edges might be identifiable graph objects or represent an unidentified relation of graph objects. Further properties of edges are attribution and constraints (e.g. ordered or sorted edges). Derived edges in the form of paths can be used to simplify otherwise very complex rules. Edges between edges, inheritance of edges, and n-ary edges are supported in neither language.

Attributes. Besides the type label, graph elements might carry attributes, which are defined by the element type. Value types can be standard types, often borrowed from host languages like Java or C (evaluation of expressions might also be borrowed). *Derived attributes* are not set directly, but evaluated according to an equation that might reference other graph elements. Additionally, sets and graph elements are useful attribute values.

Graph Transformation Rules

Graph transformation rules describe possible transformations of the working graph. They can be divided into *compound rules*, combining other rules by control structures and *simple rules*. Simple rules have a left-hand side (LHS) and a right-hand side (RHS). If the LHS is found in the working graph (i.e. it can be matched), the match is replaced by the RHS.

Matching. A rule match is a morphism that maps a rule's LHS elements to elements from the working graph. If LHS elements are mapped to only one working graph element, the morphism is a homomorphism. Non-homomorphic constructs are e.g. set nodes, amalgamated rules (AGG), star-rules (PROGRES). If each LHS element is mapped to a different element from the working graph, the morphism is injective (default). Whether the matching is non-injective (i.e. one working graph element can play multiple transformation roles) might be determined per graph grammar, per rule, or per rule element.

Conditions. Conditions define constraints for rule applications. Structural conditions are found in the LHS of a rule and include nodes, optional nodes, set nodes, paths, and restriction expressions. *Restrictions* constrain the match of a rule node by attribute or structure conditions. Attribute conditions are defined by expressions referring to element attributes. Negative application conditions (NACs) [HHT96] define structures that must not be found in the working graph, if a rule is applied; these might be integrated into the LHS or separated and range from simple negative nodes and edges to negative paths and complete negative partial graphs.

Gluing/Embedding. Gluing means the merging of two nodes into one, which owns all incident edges and all non-conflicting attributes of both. Embedding is somehow similar: it allows the redirection of incident edges from one node to another.

Signature. Procedure-like signatures support the use of graph transformation rules in a way known from imperative programming. Input parameters allow the parameterization of rules, while output parameters let transformation results influence following rules.

Control Structures. With control structures, the definition of *compound rules* is possible by combination through conditional, iteration, and chaining statements. Statements with non-deterministic behavior and backtracking allow the convenient specification of many graph algorithms. The chaining of rules should be accompanied by transactions, in order to rollback a chain of rules if one fails.

2.3 Infrastructure

Besides concepts and language properties, the infrastructure, offered to edit, analyze, and run the graph transformation system is crucial to its applicability. A graph language environment should provide a *visual and textual editor* for specifications. It should allow free-hand as well as syntax-directed editing. At least some *analyzing functions*, e.g. a sophisticated type checker, should be integrated to detect and explain inconsistencies with respect to the language's static semantics. Basic *layout algorithms* for the rules should be available in the editor.

For testing a specification, the language environment should provide an *interpreter*. During an interpreter session, the environment performs a sequence of graph transformations and visualizes the working graph. Different application strategies for transformation rules should be possible, e.g. a debugging mode allowing step-by-step execution. Additionally, a code generator should produce compilable source code for a general programming language to support the development of stand-alone applications. The generator's backend should be sufficiently flexible to allow the extension to further programming languages. A graphical framework providing access to the specified graph transformation rules should be available to obtain an executable application.

To store large graphs and support efficient manipulation of graph structures, a database should be provided. It should also support undo/redo of transformation rules and provide persistence for the working graph. Another requirement concerns the extensibility of the language environments. Monolithic architectures are hard to extend, while plug-in structures are more flexible.

Sometimes the user is confronted with limited choices concerning the platform for installation of the language environment. Therefore, the language environments should be available for at least the most common operating systems, and offer an easy and fast installation. Ideally, the environment should be implemented in a platform independent language like Java and be freely available. As all presented languages are distributed under the terms of the GNU (Lesser) General Public License.

2.4 Example

To explain the different aspects of each graph transformation system in the next sections, we introduce a simple example of a Shipping Company. The Shipping Company resembles the example used in [ERT99]. Its graph schema is illustrated in Figure 1 as a class diagram. In the example, Pallets of different weights are kept in Stores. Every Pallet has to be brought to a certain City by a Truck, which is modeled by a toDestination-edge storing also the due date. A Truck has a maximum loading weight (maxLoad) and stores its current weight (load). The order of a Truck's target cities is determined by a route, which is modeled by *ordered* onRoute-edges. The Truck is drivenBy an Employee of a Store. The boolean attribute onDuty indicates whether the Employee is at work.

Figure 2 shows the sample graph transformation rule loadUrgentPallet, which is used for loading

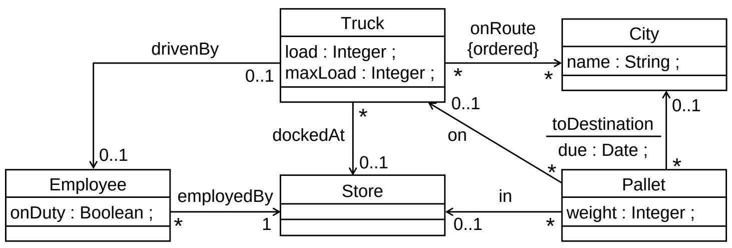


Figure 1: Graph schema of the Shipping Company

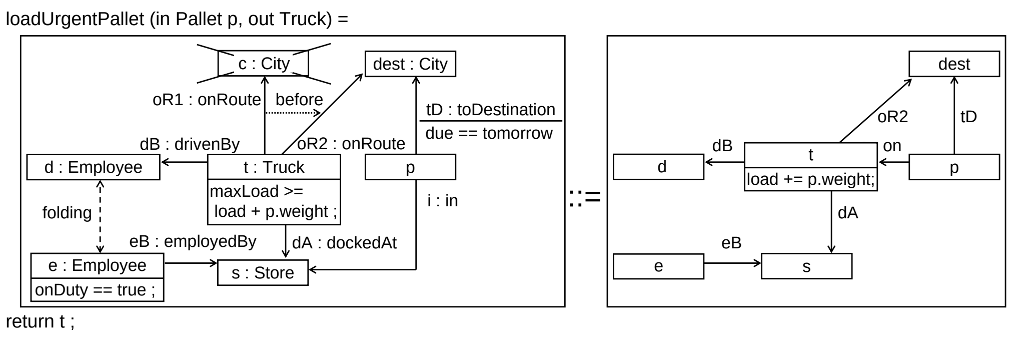


Figure 2: Graph transformation loadUrgentPallet

a given Pallet p in a suitable Truck t. The match for loadUrgentPallet is determined by the following constraints: The working graph is searched for the destination City dest and the current Store s of the given Pallet p, which is due tomorrow. Additionally, a Truck t dockedAt Store s has to be found, whose first target City is equal to the destination dest of Pallet p. This is modeled by the negative node City c, i.e. there exists no City c, which is before City dest on the route of Truck t. Furthermore, the maximum load of Truck t must not be exceeded by the weight of Pallet p. To load Pallet p on Truck t, an Employee is needed, which is onDuty. As even the driver d of Truck t may help to load Pallet p, if he is employed by Store s, Employees d and e are connected by a folding-construct. This enables the *non-injective* matching of the driver and the store employee in the working graph. A match found for the LHS is transformed according to the RHS: The in-edge incident to Pallet p is deleted and a new on-edge is created connecting Pallet p and Truck t. The load-attribute of Truck t is updated and t is returned.

3 AGG

Conceptually, AGG (Attributed Graph Grammar) [ERT99] follows the algebraic approach to graph transformation and implements single-pushout behavior. The implementation is based on the Colimit library [Wol98], which provides colimit construction for category theory of signatures and graph structures. Colimit could easily be used for the transformation of hierarchical graphs, but AGG does not support this.

An AGG graph grammar consists of a type graph, a start graph, and simple rules. Figure 3 shows a graph grammar for the Shipping Company example from Subsection 2.4.

The *type graph* contains an object-oriented description of node types, edge types, and their relations. Node types can be derived from other node types by multiple inheritance. Attributes can be defined for node and edge types. All constraints from the schema (attribute types, edges' source and target types and multiplicities) have to be checked manually within the rules. AGG does neither support derived edges (e.g. paths), derived attributes, nor meta attributes (e.g. *constant* or *static*). Although the Colimit library would allow complex edges, the language only supports binary edges between nodes. Edge constraints like ordering or sorting are not supported either, thus the ordered onRoute-edge has to be modeled as *edge-node-edge construct* with an ordering before-edge in the example's type graph (see Figure 3, top left). The AGG feature of graph constraints is not used in the example, with it one can define graph patterns and their conclusion

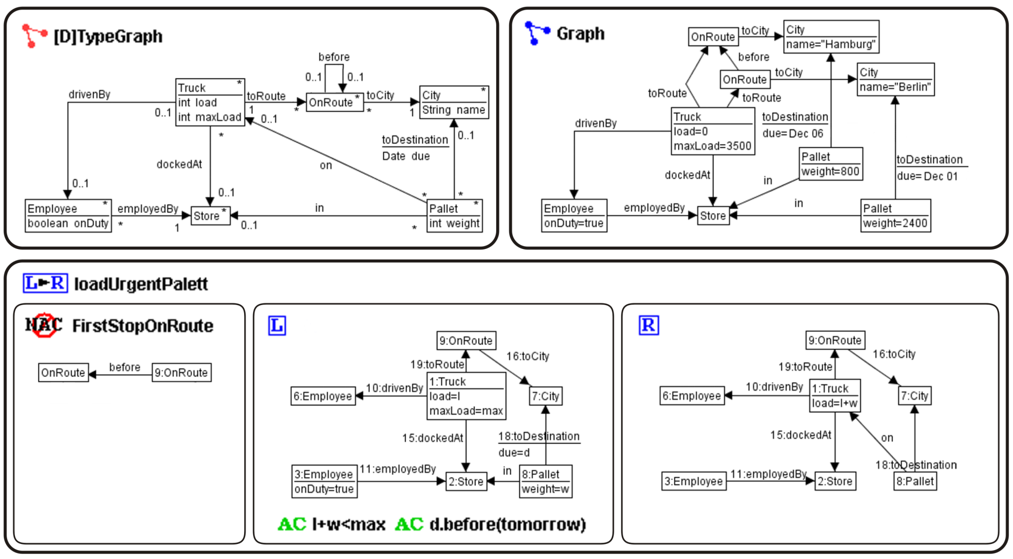


Figure 3: A simple AGG graph grammar for the Shipping Company example

to check structural properties of the working graph.

The *start graph* defines an initial working graph. All nodes and edges in a working graph are typed, identifiable, and might be attributed. Figure 3 (top right) shows a simple start graph for the example, with a small 3.5t Truck having two cities on his route (Berlin before Hamburg) and a driver, who is an Employee of the Store. Two Pallets with different weights are stored in the store, with destinations Berlin and Hamburg. They have to arrive on December 1st resp. 6th.

AGG only supports simple *rules*. The LHS consists only of nodes and edges (no other elements are available). Injective matching can be switched on and off globally[1]. With non-injective matching, the employee node from the working graph can be matched for the depicted rule in Figure 3 as node 6 (driver) *and* node 3 (store worker). NACs are subgraphs defined outside the LHS that must not be fulfilled. Here there must not be another OnRoute node before node 9. Attribute conditions, e.g. d.before(tomorrow)[2], are defined in a special attribute editor (not depicted) and can contain arbitrary Java expressions. The match is determined by the LHS, NACs, and attribute conditions. One feature not shown in the example is gluing, i.e. two nodes are merged into one node. The resulting node owns all non-conflicting attributes and edges. Merge conflicts have to be solved interactively by the user. For the complete grammar, AGG allows to compute critical pairs of rules. i.e. rules which execution disable the application of other rules.

The execution of rules can be programmed slightly, by defining *layers* for the rules. Then the execution loops over the sequence of all rules on one layer, until none is executable anymore, then the loop is executed on the next layer until the last. Additionally, single rules can be selected for execution manually.

The editing of graph grammars is done by a graphical editor, which is completely built in Java and easily installed on different platforms. The editor has a GUI that is intuitive, but does not

[1] Non-homomorphic matching, i.e. multiple matches for one rule element, can be obtained in AGG with amalgamated subrules [TB94], which is an extension not yet publicly available.
[2] d is of type java.util.Date and the Java method before compares this date with another date (tomorrow).

offer much support for syntax-directed editing. Positive is the good integration of the interpreter into the AGG editor. The generation of executable code from the graph grammar is not possible but grammar specifications can be exported to XML files.

The TIGER framework [EEHT05] allows the generation of visual editors for an AGG graph grammar. For that, the graph grammar has to be decorated by a visual concrete syntax for all elements. The generated editors are GEF-based Eclipse-plugins, where the user can pick single rules for execution. The editors use AGG's Java API to interpret the graph grammar.

AGG is based on a very sound theory and the editor is simple to use and install. This allows easy testing of prototypical specifications. [MTR06] gives a good example of a small prototypical reengineering editor specified with AGG, relying on the notion of critical pair analysis. For an application in larger projects, code generation and control structures are missing.

4 Fujaba

Originally, the focus of Fujaba (From UML to Java And Back Again) was to provide a visual modeling tool based on UML diagrams and to generate Java code from these models. Meanwhile, Fujaba also supports other metamodels and output formats. Adaptation to special application domains is eased by the template-based code generation module and the plugin based architecture. For example, Fujaba has been applied in [BGS05] to model real-time systems, including extensions of the modeling language. Fujaba has also been applied in the MOFLON framework [AKRS06] for building model transformation systems based on MOF and QVT.

In Fujaba, graph schemas are modeled using simplified UML class diagrams, resembling the one shown in Figure 1. Classes can be attributed and any Java class or ordinal type is supported as attribute type. Derived attributes are not directly offered, but can be simulated by a method replacing the getter-method generated for the attribute. We therefore model the getLoad method to derive the truck's load. Thus, this attribute does not require manual update when pallets are loaded on the truck. Inheritance of classes is supported, although multiple inheritance is restricted to interfaces. Overloading of methods and polymorphism is handled by the Java environment at runtime. Attributed associations, inheritance on associations or n-ary relations are not supported. Fujaba provides ordered associations, which impose a total ordering on the link instances during runtime. This feature is well-suited to model the onRoute association.

The behavior of applications is modeled using so-called *Story Diagrams* [FNTZ98] which combine UML-collaboration diagrams with activity diagrams. From each Story Diagram, Fu-

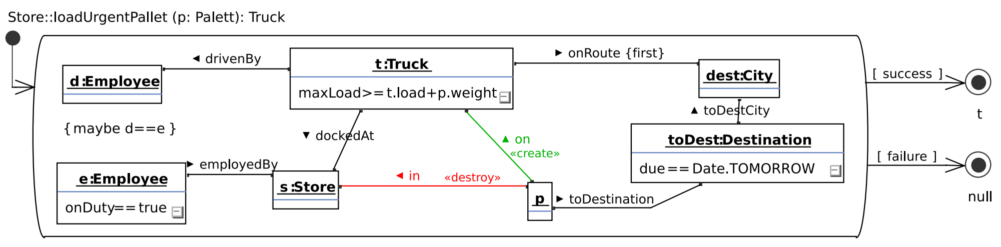


Figure 4: Fujaba Story Diagram loadUrgentPallet

jaba generates a Java method operating according to the modeled transformation rule. Story Diagrams consist of one start and at least one stop activity, and an arbitrary number of *Story Patterns* operating on the runtime graph. These elements are connected through transitions. Story Patterns correspond to rules in AGG, but incorporate LHS and RHS into one diagram using the stereotypes «create» and «destroy». For pattern matching, Fujaba offers obligatory, optional, set and negative node *variables*. By default, Fujaba creates injective morphisms from variables to objects, so that two variables are never bound to the same object. This behavior can be disabled per pair of variables. Attribute assertions may constrain the matched objects by an unparsed (thus arbitrary) Java expression. Furthermore, Fujaba supports obligatory, optional and negative edges between variables and textual path expressions. For ordered associations, additional constraints can be specified for the matching, e.g. first or last. Every pattern requires at least one *bound variable* which can be provided by a parameter of the Story Diagram, the this object the method is invoked on, variables bound in preceding patterns or by an arbitrary Java expression. From these bound variables, the other variables of the pattern are bound to objects from the runtime graph by traversing links of given type. Transformation rules are conducted after the complete pattern has been matched, and may create and delete elements, set attributes and call methods on matched objects.

Figure 4 shows a Story Diagram implementing the loadUrgentPallet transformation rule. The required bound variable is provided by the method parameter p, from which the other variables are bound. Attribute assertions are used to check if the Truck t is not overloaded and the given Pallet p needs urgent delivery (due attribute denotes tomorrow). Injective matching is disabled for variables d and e by adding the {maybe d==e} constraint. For the ordered onRoute association, {first} retrieves the first link from t to a City. If pattern matching succeeds, the runtime graph is transformed by removing the Pallet's in edge to the Store and creating an on edge to the Truck.

To model the control flow, Story Patterns may hold transitions to multiple successors. In the depicted example, two stop activities exist. By the transition guard [success], the left one is called when the transformation rule succeeds and returns the matched truck as return value. Otherwise, the right stop activity returns null. Transitions may form loops, causing repeated execution of Story Patterns. Also, *for-each*-patterns allow to process every match of a Story Pattern instead of only one match.

The formal background of Story Patterns is obtained from the logic-oriented approach described in Subsection 2.1. However, some of their semantic aspects are only incompletely defined (cf. [TMG06]). The Fujaba environment also performs very limited checks on the modeled diagrams, so the specifier is often not warned about erroneous specifications.

The generated source code can easily be integrated into existing projects or used in rapid-prototyping frameworks. eDOBS is a plugin for the Eclipse IDE which visualizes the runtime graph of a Fujaba-generated application. With the help of the CoObRA framework, generated applications are able to store their runtime states persistently. Recently, the graph-oriented database DRAGOS and the related UPGRADE [BJSW02] framework were adapted to support Fujaba. Being entirely written in Java, Fujaba works on multiple platforms and is easy to set up. Besides the regular stand-alone application, an Eclipse-plugin embedding Fujaba into the IDE is under development.

Fujaba's advantage is its extensible architecture and the use of the well-known UML. Major disadvantages are the lack of a complete semantic definition and the rare validity checks.

5 PROGRES

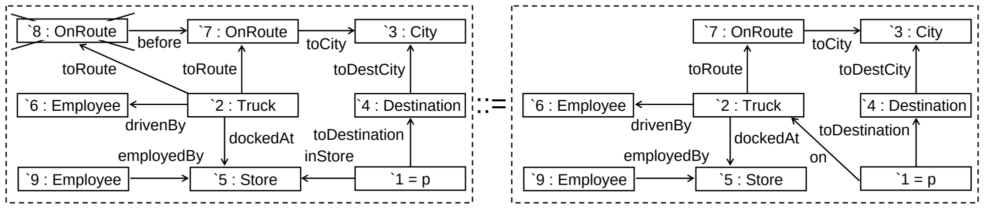
PROGRES (PROgrammed GRaph REwriting System) [SWZ99] is the eldest of the presented graph languages and environments. The logic-oriented approach [Sch91] forms the basis of PROGRES, which offers a proprietary language allowing the specification of a graph schema and consistent graph transformation rules.

PROGRES provides various constructs for defining a *graph schema* of a specification. For node types, three different types of attributes can be defined: Intrinsic attributes, whose values are assigned directly, meta attributes, which constitute class attributes and thus have the same value for every instance, and derived attributes. Values of derived attributes are computed dependent on attribute values of other nodes and are automatically updated when their values are invalid. For example, the node type Truck shown in Figure 5 owns a derived load-attribute, whose value is the sum of all loaded Pallet weights. The Pallet weights are obtained by traversing the incoming on-edges of the Truck. PROGRES supports the object-oriented paradigm regarding node types, which includes inheritance relations between node types, polymorphism, type-specific attributes and methods. *Edge types* define the type name, the source and target node types, and their cardinalities. *Paths* may be modeled allowing complex navigations through the working graph, traversing arbitrary edges of different types. PROGRES also enables the specification of *graph constraints*, e.g. there are at most *n* instances of a certain node type within the working graph. If such a constraint is violated, an appropriate repair action can be executed. Based on the schema, *incremental analyzes* check the specification for inconsistencies and show appropriate error messages.

Besides the graph schema, PROGRES offers modeling of graph queries and graph transformation rules, which may have several input and output parameters. A *graph query* defines a test for the existence of a graph pattern in the working graph. A *graph transformation rule* modifies the working graph. For their execution, the underlying graph database DRAGOS [Böh04] provides transactions for graph operations (ensuring ACID-properties). For every transformation rule, *pre- and postconditions* may be specified, which imply constraints on the working graph before

```
node_type Truck : ITEM
    intrinsic
        maxLoad : integer ;
    derived
        load : integer = 0 + all self.<-on-.weight ;
end ;

transformation loadUrgentPalett ( p : Palett , out t : Truck) =
```



```
    folding { `6 , `9 } ;
    condition `2.maxLoad > `2.load + `1.weight ; `4.due = tomorrow ; `9.onDuty = true ;
    return t := 2´ ;
end ;
```

Figure 5: PROGRES transformation rule loadUrgentPallet

resp. after the execution of the rule. Furthermore, a *qualifier* determines if a transformation rule should be applied to one match or to all possible matches in parallel. Graph transformation rules are classified as *production* (simple rule) or *transaction* (compound rule).

Productions are similar to AGG rules and Story Patterns in Fujaba. They are visually specified and allow to create and delete nodes and edges. They are described by a LHS and a RHS, which may contain obligatory nodes and edges, paths, optional nodes, set nodes, and restrictions on nodes. NACs are modeled by negative nodes, edges, paths, and restrictions. Additionally, a production may have a *condition*- and a *transfer*-part to imply conditions on attribute values resp. to change the value of node attributes. PROGRES allows the specification of *embedding rules* for redirecting edges incident to deleted nodes and embedding new nodes into the working graph. The *folding*-statement enables the non-injective mapping of two nodes in the production to the same node in the working graph.

Figure 5 shows the PROGRES production loadUrgentPallet introduced in Subsection 2.4. The production uses two edge-node-edge constructs for the ordered onRoute-edge and the attributed toDestination-edge, as these sorts of edges are not supported by PROGRES. The folding-construct, the attribute conditions and the return-statement are represented as textual statements. As the load-attribute of Truck t is defined as derived attribute, its value is not assigned explicitly.

In contrast to productions, transactions contain control structures for combining transformation rules and queries. This includes to sequence transformation rules and to execute one of a set of rules non-deterministically. Furthermore, loop- and condition-statements may be used.

PROGRES is the most expressive graph language of the three presented languages and offers extensive support for modeling big software systems. But the proprietary language is fairly complex and difficult to learn. From a specification, C and Java source code can be generated. This code can be used for rapid prototyping by applying the UPGRADE-framework. AHEAD [JSW00] is a good example of an industrial-sized project specified with PROGRES.

The syntax-directed PROGRES editor, that also features an interpreter, guides the user well, but is not really intuitive. In addition, it is only available for Linux. A further disadvantage of PROGRES is its monolithic architecture, which makes the development and implementation of new language concepts difficult.

6 Summary

Table 2 summarizes aspects of the three languages that are most important to their practical applicability, e.g. their support at specification time through a rich editor.

Table 2: Language aspects most important to practical applicability

	AGG	Fujaba	PROGRES
language expressiveness	-	o	++
most wanted features for practical applicability	control structures, paths, check against schema	paths, check against schema, search without bound object	namespaces, view concept
language learnability	graphical part intuitive (but many things hidden in dialogs)	UML-like, easy	hard due to rich expressiveness
specification editor	mixed graphical/dialog-based	mixed graphical/unparsed Java	mixed graphical/textual, syntax-directed
interpreter	manually controllable	—	step-through
code generation	—	adaptable, template-based, Java	not adaptable, C and Java
GUI support	TIGER [EEHT05]	eDOBS, UPGRADE	UPGRADE [BJSW02]
language extensibility	—	plug-in architecture	—

With the algebraic approach, AGG offers a graph transformation language with a sound theoretical basis. This offers convenient implementation possibilities for projects relying on theoretical notions. It provides a well-developed environment which can easily be installed and applied. However, the language does not seem rich enough for general purpose applications, the main disadvantage being the lack of control structures. Therefore, AGG still has to prove that it can be applied in large-scale projects.

The biggest advantage of Fujaba is its use of UML, which requires only little learning effort from the user. In addition, the vivid community is working intensively on improvements and further extensions. However, the language lacks a formal definition, forcing the user to inspect the code when in doubt about language semantics. Additionally, due to the lack of analyzes the user is not sufficiently guided during the specification process, often leading to malfunctioning or unexpected behavior of the generated code.

PROGRES offers the most sophisticated language, although there are still some features missing. The infrastructure, including a syntax-directed editor, an interpreter, and a code generation mechanism, provides the highest level of maturity. The experience with industrial-sized projects proves the practical usability of PROGRES. However, the environment does not conform to today's standards, requiring a painstaking installation process and providing a relatively inconvenient interface, particularly to new users. Additionally, it is very hard to extend this extensive environment and language for new features.

The jury is still out: Because of the different relevance of the compared aspects, we cannot give final advice, but leave it to the reader to decide which language to use.

Bibliography

[Agr04] A. Agrawal. Model Based Software Engineering, Graph Grammars and Graph Transformations. Area paper, EECS at Vanderbilt University, 2004.

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In Rensink and Warmer (eds.), *Model Driven Architecture - Foundations and Applications (ECMDA-FA'06)*. LNCS 4066, pp. 361–375. Springer, 2006.

[BGS05] S. Burmester, H. Giese, W. Schäfer. Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In *Proc. of the European Conf. on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany*. LNCS 3748, pp. 25–40. Springer, 2005.

[BJSW02] B. Böhlen, D. Jäger, A. Schleicher, B. Westfechtel. UPGRADE: A Framework for Building Graph-Based Interactive Tools. In Mens et al. (eds.). ENTCS 72, pp. 149–159. Elsevier Science Publishers, 2002.

[Böh04] B. Böhlen. Specific Graph Models and Their Mappings to a Common Model. Pp. 45–60 in [PNB04].

[BTMS99] R. Bardohl, G. Taentzer, M. Minas, A. Schürr. *Application of Graph Transformation to Visual Languages*. In [EEKR99], pp. 105–180, 1999.

[CEH$^+$97] A. Corradini, H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner. *Algebraic Approaches to Graph Transformation – Part I: Basic Concepts and Double Pushout Approach*. In [Roz97], pp. 163–245, 1997.

[EEHT05] K. Ehrig, C. Ermel, S. Hänsgen, G. Taentzer. Generation of visual Editors as Eclipse Plug-ins. In *20th IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'05*. Pp. 134–143. ACM Press, New York, 2005.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*. Volume 2. World Scientific, 1999.

[EHK$^+$97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini. *Algebraic Approaches to Graph Transformation – Part II: Single Pushout Approach and Comparison with Double Pushout Approach*. In [Roz97], pp. 247–312, 1997.

[ERT99] C. Ermel, M. Rudolf, G. Taentzer. *The AGG Approach: Language and Environment*. In [EEKR99], pp. 551–603, 1999.

[FNT98] T. Fischer, J. Niere, L. Torunski. Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling. Master Thesis, University of Paderborn, 1998.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Ehrig et al. (eds.), 6th *Int. Workshop on Theory and Application of Graph Transformation (TAGT)*. LNCS 1764, pp. 296–309. Springer, 1998.

[HHT96] A. Habel, R. Heckel, G. Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae* 26(3/4):pp. 287–313, 1996.

[JSW00] D. Jäger, A. Schleicher, B. Westfechtel. AHEAD: A Graph-Based System for Modeling and Managing Development Processes. In Nagl et al. (eds.). LNCS 1779, pp. 325–339. Springer, 2000.

[KASS03] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science* 9(11):1296–1321, Nov. 2003.

[MG05] T. Mens, P. V. Gorp. A Taxonomy of Model Transformation and its Application to Graph Transformation. 2005. , presented at the 1st International Workshop on Graph and Model Transformation, GraMoT'05, Tallinn, Estonia.

[MTR06] T. Mens, G. Taentzer, O. Runge. Analysis Refactoring Dependencies using Graph Transformation. *Software Systems Modeling (SoSyM)*, 2006.

[Nag79] M. Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg Verlag, 1979.

[PNB04] J. L. Pfaltz, M. Nagl, B. Böhlen (eds.). 2nd *Int. Workshop on Applications of Graph Transformations with Industrial Relevance, AGTIVE'03*. LNCS 3062. Springer, 2004.

[Ren04] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. Pp. 479–485 in [PNB04].

[Roz97] G. Rozenberg (ed.). *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*. Volume 1. World Scientific, 1997.

[Sch91] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. PhD-Thesis, RWTH Aachen University, 1991.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. *The PROGRES Approach: Language and Environment*. In [EEKR99], pp. 487–550, 1999.

[TB94] G. Taentzer, M. Beyer. Amalgamated Graph Transformations and Their Use for Specifying AGG - an Algebraic Graph Grammar System. In *Int. Workshop on Graph Transformations in Computer Science*. Pp. 380–394. Springer, 1994.

[TEG+05] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varró, , S. Varró-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *Proceedings of the International Workshop on Model Transformations in Practice, MTiP'05 (Satellite Event of MoDELS 2005)*. Montego Bay, Jamaica, 2005.

[TMG06] M. Tichy, M. Meyer, H. Giese. On Semantic Issues in Story Diagrams. In Giese and Westfechtel (eds.), *Fujaba Days 2006*. Technical Report tr-ri-06-275, pp. 10–14. University of Paderborn, Germany, 2006.

[VSV05] G. Varró, A. Schürr, D. Varró. Benchmarking for Graph Transformation. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Pp. 79–88. IEEE Computer Society, 2005.

[Wol98] D. Wolz. *A Colimit Library for Graph Transformations and Algebraic Development Techniques*. PhD-Thesis, TU Berlin, 1998.