A Flexible and Easy-to-Use Library for the Rapid Development of Graph Tools in Java

H. J. Sander Bruggink[1], Barbara König[2], Marleen Matjeka[2], Dennis Nolte[2], and Lara Stoltenow[2](✉)

[1] GEBIT Solutions, Düsseldorf, Germany
[2] Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen, Duisburg, Germany
{barbara_koenig,lara.stoltenow}@uni-due.de

Abstract. We present a programming library for the rapid development of graph tools, with applications in graph transformation and related fields. Features include working with graphs, graph morphisms, basic categorical constructions such as computing pushouts and pushout complements or enumerating all morphisms with certain properties, but also applications such as executing graph transformation steps. Additionally, we offer graphical user interface widgets for visualization and manipulation of graphs, morphisms and categorical diagrams.

Our objective is to allow users to quickly develop graph tools for both simple and complex problems, to allow easy embedding into existing software, and to have comprehensible code especially for the main algorithms. Existing tools that demonstrate the versatility and ease of use of the library include: DPOdactic (a didactic tool for teaching double-pushout graph transformation), DrAGoM (a tool to handle multiply annotated type graphs for abstract graph rewriting), and Grez (termination analysis of graph transformation systems).

Keywords: Graph transformation · Rapid development · Graph tools

1 Introduction

The graph transformation community has always been strong in the development of tools, for support of generic graph rewriting, for supporting software development and for verification and analysis (see for instance [2,6,8,15,16] for a non-exhaustive enumeration).

However, to our knowledge there is no publicly available, easily accessible and flexible library that provides a backbone and toolbox for the rapid development of graph tools, including both the support of various constructions and visualization of graphs. We have developed such a library and are still in the process of extending it. Since we believe that there may be a wider interest, we will here present it as a community service and describe existing tools that are already based on such a library. It does not contain groundbreaking new functionality, but in our opinion we present a nice, comprehensive package.

Our design principles while developing this library are as follows:

- It is designed to have a low learning curve, which we have tested by its successful use in several student projects. It has been integrated both into tools that have been implemented from scratch, and the continued development of existing tools.
- Simple tasks should be easy to implement quickly. We will illustrate this with two suggestive examples:
  - If you need to compute the composition of two given graph cospans, it should – after becoming acquainted with the library – take you as much time to do it by hand on paper, as it does to just write a small prototype program.
  - Assume you develop a nice theory about commuting hexagons of graph morphisms and you want beautiful renderings of them. Using generic utility functions, you can convert the abstract representation into a displayable graph using no more than 50 well-formatted lines of code.
- We aim for readability and favor clear and understandable code over raw speed. It should e.g. be possible to learn how to compute pushouts of graph morphisms just by reading the code.
- We provide automatic visualization of entities such as graph morphisms, commuting squares, etc., which are typically not supported by general purpose graph display libraries.
- We aim for easy integration of the library into your own application.

The library can be downloaded from https://www.uni-due.de/theoinf/research/tools_javagraph.php.

Related Work. We have evaluated some of the tools that are commonly used in the context of graph transformation and offer similar functionality, in particular, Progres [16], GraJ [6], ENFORCe [2], AGG [15], and a tool for graph transformation by computational category theory [13].

Progres is a suite of tools that focuses on the specific application of graph grammars and graph rewriting systems. It is, however, not designed as a generic library. Together with the fact that no source code is available (but only non-portable binaries), it is probably hard to embed it into other applications.

GraJ is a tool for the execution of graph programs. It features a modular design that facilitates embedding into custom tools. ENFORCe builds on GraJ to prove correctness not only for graph programs, but also other weak adhesive HLR categories. Notably, it also supports graph conditions and constraints. Both tools are, however, currently unmaintained and not publicly available for download. Thus we could not evaluate its suitability for e.g. prototyping purposes.

AGG also focuses on graph grammars and graph transformation. It too has support for graph conditions, and has an extensible architecture. However, it appears to be designed as a standalone tool. While it is a very powerful tool and the components that do the actual computations feature useful algorithms, the programming interface does not appear to be specifically designed for use as a library.

In [13], a library for carrying out graph transformation in an abstract categorical setting is proposed. In this way, it is similar to the CatLib component of our work. The full code was not available for evaluation, but their focus is less on graphs and more on the categorical side, which makes it potentially more difficult to work with the library. In addition the library does not offer a ready-to-use visualization component.

Our justification for the development of a new library is not just to avoid these particular problems, but to also focus on additional aspects (ease of use, prioritize clear and understandable code over efficient implementations, make it easy to embed into your own tools) as detailed above. In this regard, it is similar in spirit to the SiTra library [1] which focuses not on practical applications, but to "aid a programmer in learning the concept of writing transformation rules".

Outline of the Paper. The article is structured as follows: In Sect. 2, we describe the architecture and the features of our library. In Sect. 3, we give a detailed overview of existing tools that are using the library. We conclude in Sect. 4 with an outline of future work.

2 Components and Features

2.1 Components Overview

In this section, we give a detailed overview of the components that together make up the library and the features that are available. The components can be used together or independently of each other as needed.

The *Java-Graph* component provides the computational foundations. It provides abstract representations of graphs, graph morphisms, graph conditions and related objects; categorical constructions such as pushout complements; enumeration of morphisms with certain properties; graph transformation; loading and saving of objects to files in a plaintext format that is easy to read and write. We give a more detailed description of this component in Sect. 2.2.

Java-Graph by itself provides no graphical user interface and can therefore be used for batch processing tasks, or as part of tools that already build on different frameworks. Graphical output is provided by a separate component.

The *VisiGraph* component is responsible for displaying graphs to the user, and provides a similar feature set as other graph display libraries. It automatically layouts graphs that can then be shown to the user. Currently, it provides display and editor widgets for Swing-based graphical user interfaces (however, it does not have a strong dependency on Swing and can be quickly ported to work with other GUI toolkits). It is also possible to export the graph to image files.

As a companion component, *VisiGraphJS* is a reimplementation in Javascript and can be used to provide the same type of visualizations in web applications.

It is also possible to do the layouting process in Java using VisiGraph and then only display the result in a web browser.

The *VxToolbox* component serves as a bridge between Java-Graph and VisiGraph. It is responsible for creating useful visualizations not just for ordinary graphs, but for the various objects that are supported by Java-Graph. As an example, the visualization of a pushout square should put the four graphs at the four corners of an appropriately-sized square, and to make the output more easily graspable, common elements (e.g. nodes that are in both the domain and codomain of some morphism) should be positioned in a consistent way. VxToolbox provides not only visualization routines for the objects supported by Java-Graph, but also basic building blocks to make it easy to generate visualizations of custom objects (as a rule of thumb, visualizing e.g. commuting hexagons should require no more than 50 lines of code).

Finally, *CatLib* is a generalization of Java-Graph to arbitrary categories. CatLib can be used independently of, or together with, Java-Graph. Prototype tools can thus be implemented in a generic way, doing computations on arbitrary categories, where Java-Graph is used to showcase the generically implemented tool for a specific example category. Currently, CatLib implements the categories **Set** and, using Java-Graph, the category of finite (hyper)graphs $\mathbf{Graph_{fin}}$.

2.2 Detailed Description of the Java-Graph Component

At the core, we have the `de.uni_due.inf.ti.graph` package (prefix abbreviated hereinafter as ...`ti.graph`), with classes for the basic entities. Graphs are represented with the `Graph` class, containing collections of `Node`s and (hyper-)`Edge`s with `Label`s. We provide the usual methods for construction and manipulation of graphs such as `graph.addEdge(new Label("A"),n0,n1,n2)` to add a ternary hyperedge or `graph.getNodes()` to obtain a (read-only) `List` of the nodes in a `Graph`. Although edges are generally hyperedges, we provide additional methods as simplifications for the common case of directed edges (e.g. `edge.getTarget()` as an alternative to `edge.getNodes().get(1)`).

Using the ...`ti.graph.io` package, all supported objects (graphs, conditions etc.) can be read and written in a custom text-based file format named SGF. The textual representation of SGF resembles the way a graph would be written on paper. The SGF code `graph { n0 --A-> n1 --A-> n2 --A-> n0; };` describes a graph with three nodes (`n0` to `n2`) that are connected by directed a-labeled edges in a circle. Objects can be loaded from files or from strings. In our example below, we use the latter, in conjunction with the Java 13 Text Blocks feature, to obtain very concise prototype code.

Graph morphisms map elements of one graph to compatible elements of another one, where the map can be either total or partial. A `Morphism` has a `Map`-like interface (`mor.get(node0)`, `mor.getPreimage(edgeA)` and the like) with additional functionality; for instance, `mor.put(domEdge, codomEdge)` maps not only the edge, but also creates mappings for all nodes that are incident to the given edge (unless this mapping would conflict with the node mapping of the graph, in which case an exception is thrown). Morphisms can be created easily by explicitly giving the node and edge mappings, either using the `put` method in Java, or using `=>` in SGF (see the example at the end of this section).

Graph conditions can be used to specify additional properties of graphs such as the existence or absence of certain elements. They come in two flavours: the *nested conditions* (roughly, first-order formulas on graphs) as introduced in [9] for weak adhesive HLR categories; and *cospan conditions*, which use a slightly different tree-based structure, as introduced in [3] for adhesive categories. As an example for the former, the condition $\exists(m_1, \text{true}) \vee \exists(m_2, \text{true})$, where m_1, m_2 are morphisms describing the elements that should exist at some point, can be written in SGF as follows: `c = or [ exists(m1,true), exists(m2,true) ];`

We provide various fundamental categorical constructions (...`ti.graph.ext`). Given a span, a cospan, or a pair of composable morphisms, it is possible to compute the pushout, pullback, or pushout complement, respectively. It is possible to enumerate all morphisms between two graphs with certain properties (examples include enumeration of all total injective morphisms; all partial morphisms; all isomorphisms; all morphisms that extend a given base morphism). Furthermore, given a span, it is possible to enumerate all jointly epi squares. `Enumerator` implements `Iterable`, and hence can be used in loops (e.g. `for (Morphism i : Morphism.getIsomorphisms(g1, g2)) { ... }` to executesome code for all isomorphisms between two graphs g_1, g_2), or as `Streams` (`Morphism.getIsomorphisms(g1, g2).stream().map(i -> ...)`). All of these enumerators compute their results lazily and so also work when the total number of possible morphisms is very large.

The following example code creates objects for a pair of graph morphisms $g_L \xleftarrow{m_{TL}} g_T \xrightarrow{m_{TR}} g_R$, where $g_T = \overset{1}{\bullet}\ \overset{2}{\bullet}\ \overset{3}{\bullet}$ (three isolated nodes), $g_L = \overset{1}{\bullet}\xrightarrow{A}\overset{2\,3}{\bullet}$, $g_R = \overset{1\,2}{\bullet}\xrightarrow{B}\overset{3}{\bullet}$, and morphisms m_{TL}, m_{TR} merge nodes $2, 3$ and $1, 2$ respectively. Then their pushout is computed and the result is printed to standard output:

```
String sgfContent = """
  gT = graph { node n1; node n2; node n3; };
  gL = graph { n1 --ea:A-> n23; };
  gR = graph { n12 --eb:B-> n3; };
  mTL = morphism from gT to gL { n1 => n1; n2 => n23; n3 => n23; };
  mTR = morphism from gT to gR { n1 => n12; n2 => n12; n3 => n3; };
""";
Map<String, Object> sgfMap = SgfParser.parseSgfString(sgfContent);
Morphism mTL = (Morphism) sgfMap.get("mTL");
Morphism mTR = (Morphism) sgfMap.get("mTR");
Square po = Pushout.compute(mTL, mTR);
System.out.println(po);
```

As an application of the fundamental constructions, graph transformation systems using the single-pushout and double-pushout approaches can be directly described and processed by the library (...`ti.graph.transformation`). So far we restrict to injective match and rule morphisms. In SGF, if a rule morphism is not explicitly specified, then elements on the left and right hand sides are automatically related if they have the same name. For instance, in the rule `r = rule { { n1 --A-> n2 --B-> n3 } => { n1 --C-> n3 } }`, nodes n_1, n_3 are mapped to their counterparts on the right hand side, the c-edge is created, and node n_2 and the two edges are deleted at rule application. To enumerate all possible results of rewriting a `Graph g` using `Rule r`:

```
for (Morphism match : r.getMatches(g)) {
  Transition t = r.applyToMatch(match);
  Graph rewrittenGraph = t.getTarget();
  // process rewrittenGraph somehow
}
```

3 Existing Tools Using the Library

In this section, we describe some of the existing tools that are currently using the library. Notably, we present: DPOdactic (a didactic tool for graph transformation), DrAGoM (multiply annotated type graphs for abstract graph rewriting), and Grez (termination analysis). Additionally, we give a quick overview of tools that are currently under development. These tools demonstrate that the library can be used in a variety of different application areas.

3.1 DPOdactic

DPOdactic [12] is a tool that walks the user through the process of applying double-pushout (DPO) graph transformation rules. In this setting, a rule states that the occurence of some subgraph L is to be replaced by another graph R. The relationship between L and R is established via an interface graph I and two injective morphisms that map I to L, R respectively. A rule is applied by locating a match of L – where DPOdactic also allows non-injective matches – removing parts of L, but keeping I, and then adding the missing parts of R.

In the tool (Fig. 1), the user is presented with a rule and a graph G that the rule should be applied to. First, they select one of (possibly) multiple occurences of L in G. Then, they input the context graph, followed by the morphisms that relate it to the other graphs. Finally, they input the result of the transformation step and the related morphisms. The tool checks all intermediate results for inputs and provides direct feedback to the user, including hints on where to look for mistakes. Optionally, the tool can also simply compute the result of each step.

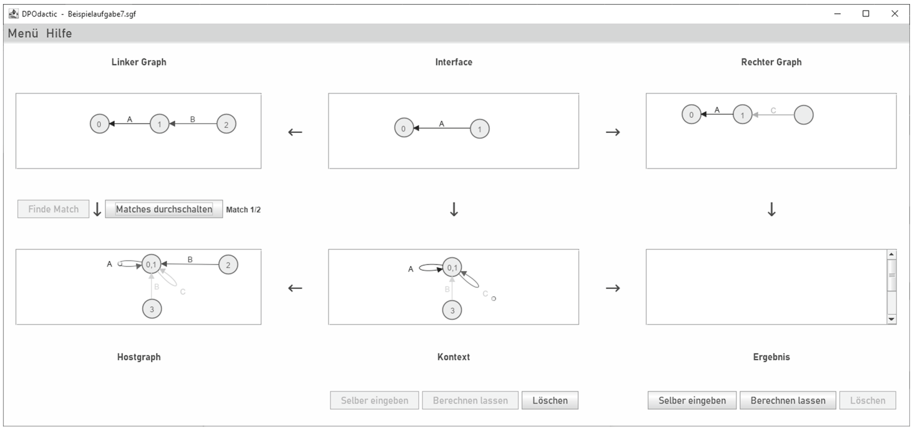

Fig. 1. Main window of DPOdactic after the user has provided the correct context graph, with the result graph yet to be computed (by the user or by the tool).

3.2 DrAGoM

DrAGoM [14] is a prototype tool to handle and manipulate so-called multiply annotated type graphs. The main application of DrAGoM is to automatically compute strongest postconditions in order to check invariants of graph transformation systems, in the framework of abstract graph rewriting.

DrAGoM uses a materialization construction to extract concrete instances of a left-hand side graph out of an abstract graph. Then, it can be used to automatically compute the strongest postcondition of the materialization, i.e. an annotated type graph, specifying exactly the language of all graphs which are reachable in one rewriting step.

3.3 Grez

Grez [5] is a tool to automatically produce proofs of uniform (non-)termination of graph transformation systems, i.e. whether it is possible to obtain an infinite sequence of rule applications from some start graph or not. Grez uses various approaches for analysis: some are simple (e.g. if all rules reduce the number of nodes, then rewriting must terminate at some point), while others are more complicated (e.g. termination arguments based on weighted type graphs [4]).

Typically, algorithms classify rules as decreasing, non-increasing, or possibly-increasing with respect to some order. Grez can then combine the results of multiple algorithms using a relative termination argument: if one algorithm can only prove a subset of the rules as decreasing (thus terminating) and the remaining rules as non-increasing, then termination of the remaining rules (for which a different algorithm can be used) implies termination of the original system.

3.4 Further Tools

Numerous other, smaller tools that are currently in alpha stage are being developed using the library, with areas of application being the analysis of (conditional) graph transformation systems, satisfiability checking of graph conditions, and tools that automate various basic tasks.

As an example for the automation of basic tasks, we have implemented a tool (Podmineny) that, given a pair of cospans (typically corresponding to the left-hand side of a rule and a graph with interfaces), computes all borrowed context diagrams [7]; a task that is tedious and error-prone when done by hand.

As a case study, we have partially re-implemented the tableau resolution algorithm for graph properties as described in [11]. While this tool only implements part of the functionality, it encouraged us to start work on another prototype tool, RSsat, for both model finding and unsatisfiability proofs in the more generic setting of reactive systems.

Table 1. Overview of tools that are currently using the library. The columns indicate which components (Java-Graph, VisiGraph, VisiGraphJS, VxToolbox, CatLib) are currently used (●), will (○) or could (○) be used in future versions.

Tool	Description	Jg	Vx	Js	Tb	Cl
Grez	Termination analysis for graph transformation systems [5]	●	●	○	○	
DrAGoM	Manipulation of multiply annotated typegraphs [14]	●	●		○	
DPOdactic	A didactic tool for double-pushout graph transformation systems [12]	●	●			
Podmineny	Enumeration of all borrowed context diagrams, given two graph cospans	●	●	○	○	○
RSsat	Prototype tool for model finding and unsatisfiability proofs for conditions in reactive systems	●	●	●	●	●
TGC	A partial implementation of tableau resolution [11] for graph properties	●	●		○	
Your tool here	:-)	?	?	?	?	?

Table 1 gives a quick overview of current and future tools.

4 Future Work

In addition to the existing documentation for classes and methods, we plan to provide an introductory user guide for getting started with the library. As supporting material, we will implement several smaller tools that can serve as examples or templates for the development of other tools.

Naturally, we also plan to lift the restrictions on the injectivity of match and rule morphisms and to extend the functionality in general.

Our library currently supports SGF as a custom text-based data interchange format. We feel that the simple syntax of SGF goes well with the design goal of facilitating the development of prototype tools. Future versions of the library will additionally support the Graph eXchange Language (GXL) [10], an XML-based interchange format that is used by other tools. Note that GXL is not primarily designed to be hand-written by users (as SGF is), but to be generated by tools.

While the VisiGraph library has no strong dependency on Swing and support for other toolkits can be easily added if needed, we plan to provide interfaces to additional common GUI toolkits directly in our library. For the generation of mechanical proofs (e.g. (non-)termination proofs for graph transformation systems in Grez) we will also add direct generation of LaTeX code.

Furthermore, we plan to use the library to develop further tools to demonstrate applicability of our own future research, such as the analysis of reactive systems conditions.

References

1. Akehurst, D.H., Bordbar, B., Evans, M.J., Howells, W.G.J., McDonald-Maier, K.D.: SiTra: Simple transformations in Java. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Model Driven Engineering Languages and Systems, pp. 351–364. Springer, Heidelberg (2006). https://doi.org/10.1007/11880240_25
2. Azab, K., Habel, A., Pennemann, K.H., Zuckschwerdt, C.: ENFORCe: A System for ensuring formal correctness of high-level programs. In: Proceedings 3rd International Workshop on Graph Based Tools (GraBaTs 2006). vol. 1, pp. 82–93. Electronic Communications of the EASST (2007)
3. Bruggink, H.J.S., Cauderlier, R., Hülsbusch, M., König, B.: Conditional reactive systems. In: Proceedings of FSTTCS 2011. LIPIcs, vol. 13. Schloss Dagstuhl - Leibniz Center for Informatics (2011)
4. Bruggink, H.J.S., König, B., Nolte, D., Zantema, H.: Proving termination of graph transformation systems using weighted type graphs over semirings. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 52–68. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_4
5. Bruggink, H.J.S., König, B., Zantema, H.: Termination analysis for graph transformation systems. In: Diaz, J., Lanese, I., Sangiorgi, D. (eds.) TCS 2014. LNCS, vol. 8705, pp. 179–194. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44602-7_15
6. Busatto, G.: GraJ: A System for executing graph programs in Java. Berichte aus dem Fachbereich Informatik 3/04, Universität Oldenburg (2004)
7. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. Math. Struct. Comput. Sci. **16**(6), 1133–1163 (2006)
8. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. Int. J. Softw. Tools Technol. Transfer **14**(1), 15–40 (2012)
9. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**(2), 245–296 (2009)
10. Holt, R.C., Schürr, A., Sim, S.E., Winter, A.: GXL: A Graph-based standard exchange format for reengineering. Sci. Comput. Program. **60**(2), 149–170 (2006)
11. Lambers, L., Orejas, F.: Tableau-based reasoning for graph properties. In: Giese, H., König, B. (eds.) Graph Transformation, pp. 17–32. Springer, Cham (2014)
12. Matjeka, M.: Ein didaktisches Tool zur Anwendung von Graphtransformation. Bachelor's thesis, Universität Duisburg-Essen, June 2019

13. Minas, M., Schneider, H.J.: Graph transformation by computational category theory. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday, pp. 33–58. Springer, Heidelberg (2010)
14. Nolte, D.: Analysis and abstraction of graph transformation systems via type graphs. Ph.D. thesis, Universität Duisburg-Essen, August 2019
15. Runge, O., Ermel, C., Taentzer, G.: AGG 2.0 - new features for specifying and analyzing algebraic graph transformations. In: Schurr, A., Varro, D., Varro, G. (eds) Proceedings of AGTIVE 2011, vol. 7233, pp. 81–88. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34176-2_8
16. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: Language and environment. In: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools, pp. 487–550. World Scientific (1999)