DrAGoM Manual

Dennis Nolte

June 9, 2019

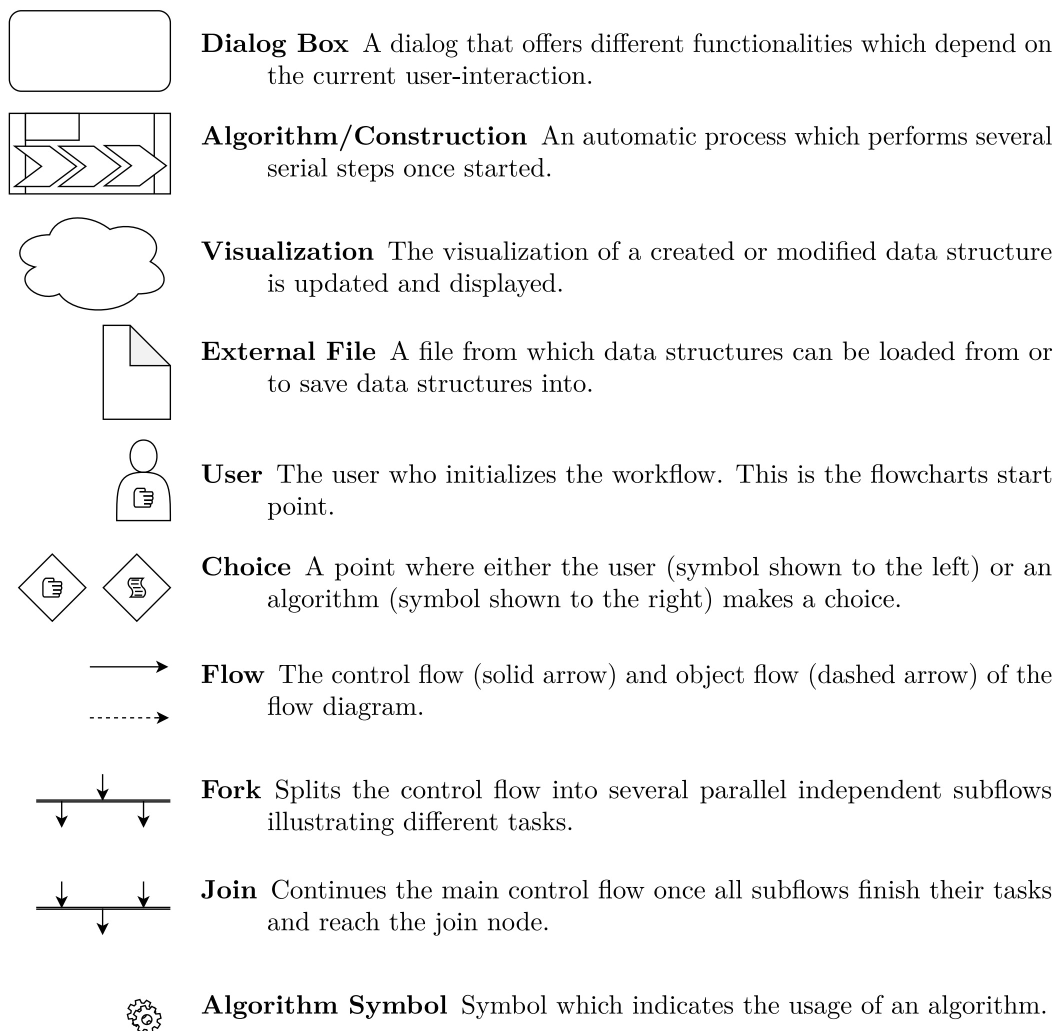
An Introduction to DrAGoM

The implementation of DrAGoM started in Spring 2016 as a prototype tool to visualize graphs contained in a graph language specified by an annotated type graph. In Fall 2018 the prototype tool was used as a base to build a new tool on top of it. Given a graph language specified by a multiply annotated type graph, we implemented techniques which are able to construct an abstract graph that specifies the strongest postcondition with respect to a graph transformation rule. To this end, we created DrAGoM.

Readers are invited to consult [CH+19] and [CKN19] for the theoretical background.

The DrAGoM-User Workflow

In this section we describe the typical workflow of a user interaction with DrAGoM. The workflow which we refer to is depicted in Figure 1. First, we explain the semantics of all elements in the shown flowchart:



In the following we will describe the flowchart (see Figure 1) in more detail.

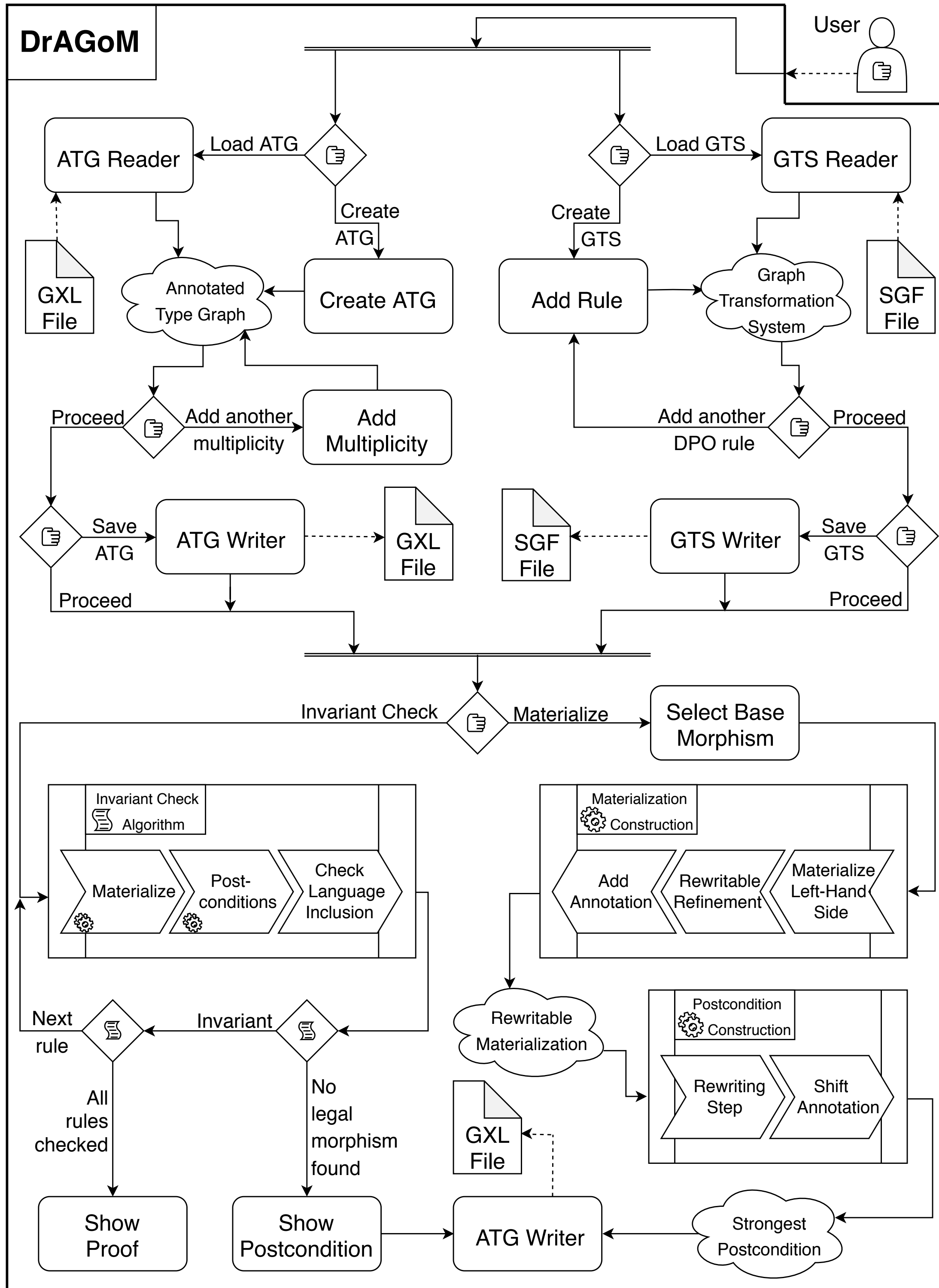


Figure 1: UML flowchart of a typical user interaction with DrAGoM

The interaction starts with the user who initializes DrAGoM. Before the tool can start with the analysis, the user needs to create (or load) an annotated type graph (short: ATG) and a graph transformation system (short: GTS).

Annotated type graphs can be directly created via the user interface. After the creation, the user can add additional multiplicities to extend the annotated type graph to a multiply annotated type graph. For loading and storing multiply annotated type graphs, DrAGoM uses the XML based standard *Graph eXchange Language*[1] (GXL) [Win02; KRW02; HS+06]. An example for the file format, of an encoded multiply annotated type graph, is given at the end of this manual.

Likewise, graph transformation systems can be directly created via the user interface. The user can create a graph transformation rule which is added to an empty graph transformation system. This process can be repeated until all rules have been added. For loading and storing graph transformation systems, DrAGoM uses the *Simple Graph Format* (SGF) [Bru15]. Unlike other text-based formats, such as XML-derived formats, which are mainly designed to be easily parsed by a computer program, the SGF format is designed to be easily written, read and maintained in source form by the user. An example for the file format of an encoded graph transformation system, can be found at the end of the manual.

Once both data structures, the multiply annotated type graph and the graph transformation system, are created, the user can choose to either compute a rewritable materialization or to let DrAGoM perform an invariant check.

For the case that the user wants to compute the rewritable materialization, the currently displayed graph transformation rule is used for the construction. The user can select a semi-legal base morphism (if there are any) and afterwards DrAGoM automatically computes the rewritable materialization. This materialization can be used as an input for the computation of the strongest postcondition. The user can choose to save the multiply annotated type graph which specifies the strongest postcondition in the GXL format. Furthermore, the user can choose to let DrAGoM perform a language inclusion check with respect to the computed postcondition and the initial annotated type graph.

If the user chooses to let DrAGoM perform an invariant check instead, both algorithms, the materialization construction (for all semi-legal base morphisms) and the postcondition construction, are performed in sequence. The gearwheel symbols in the algorithm steps indicate that these two tasks resemble the two constructions depicted on the right side of the diagram. Afterwards, a language inclusion check is used to possibly find a legal morphism from the graph specifying the postcondition to the graph that specifies the initial graph language. If such a legal morphism can not be found, the corresponding multiply annotated type graphs are displayed and DrAGoM is unable to prove that the initial specified graph language is an invariant for the given graph transformation system. Otherwise, the algorithm is restarted until every rule of the graph transformation system is checked. If for all rules and every semi-legal base morphism there exists a corresponding legal morphism for the postcondition graph, then all legal morphisms are displayed as a proof to the user.

[1] See also `http://www.gupro.de/GXL/`

Tutorial: How to Use DrAGoM

The following tutorial provides an overview of the basic functionalities in DrAGoM. The API documentation, a pre-compiled version of the tool alongside the source code and additional informations can be found on the DrAGoM homepage[2].

System Requirements, Third-Party Libraries and Installation

DrAGoM has been implemented in Java. For the usage of DrAGoM a *Java Runtime Environment* (JRE) of version 1.8 or higher is required. The tool offers a *graphical user interface* (GUI) which allows the user to create and manipulate graph transformation systems and multiply annotated type graphs. Since DrAGoM is written in Java, it can be used on Linux, MacOS and Windows.
DrAGoM depends on the following two third-party libraries:

Z3-Java A Java library of the Z3 theorem prover from *Microsoft Research*.

JGoodies A Java library which offers reliable building blocks for Java applications.

DrAGoM is distributed under the terms of the 3-clause BSD open source license. The third-party library *JGoodies* is distributed under relaxed terms (2-clause) of the BSD license and *Z3-Java* is licensed under the MIT license.

Figure 2: DrAGoM GUI

The standard DrAGoM distribution is a pre-compiled `.jar` file that contains all required libraries except for the native library of the Z3 release, since these libraries depend on the user's operating system. DrAGoM calls the external SMT solver Z3 to compute annotations

[2] DrAGoM homepage: `https://www.uni-due.de/theoinf/research/tools_dragom.php`

for the rewritable materialization. Therefore the user has to download a Z3 release which fits the used operating system. The operating system's search path has to include the folder which contains the *libz3java* dynamic link library.

To compile DrAGoM from source instead, the user needs to configure the project's library dependencies. All required external libraries are provided in the resource folder of the source code archive on the DrAGoM homepage. A detailed description of the dependencies can be found in the DrAGoM API documentation, also available on the homepage.

In most operating systems, double-clicking on the `dragom.jar` file will open DrAGoM in GUI-mode (see Figure 2). Otherwise, the GUI can be directly started from the command-line via

```
java -jar dragom.jar
```

In some Linux operating systems, DrAGoM might not be able to immediately locate the folder containing the *libz3java* library. To help the tool find the required library the user should set the global variable named LD_LIBRARY_PATH to point to the corresponding folder path, i.e., DrAGoM is launched by typing

```
LD_LIBRARY_PATH=<PATH_TO_LIBRARY_FOLDER> java -jar dragom.jar
```

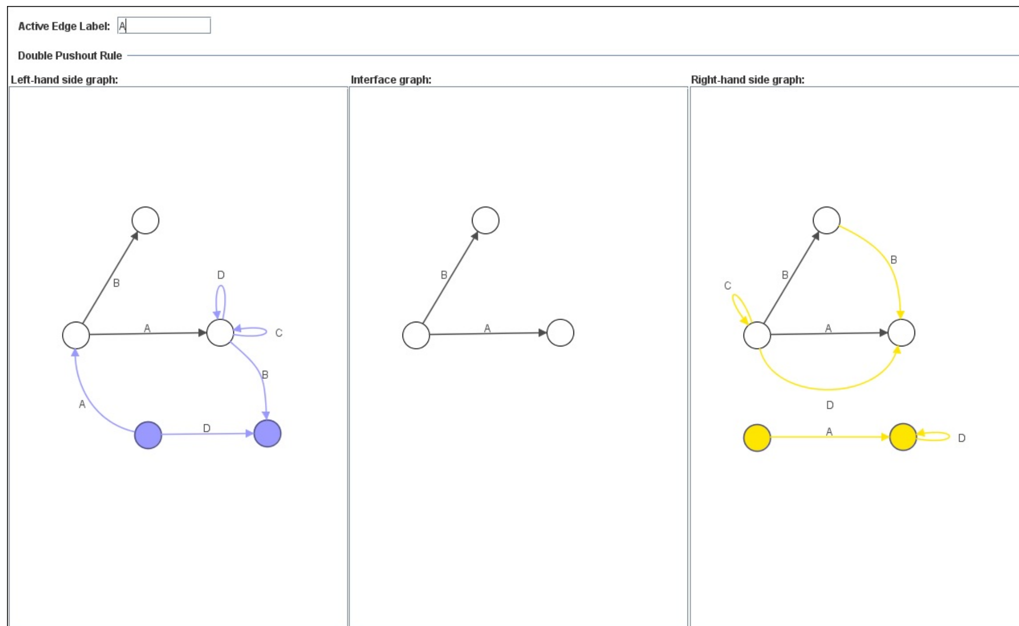


Figure 3: DrAGoM Add-Rule dialog

The Main Window and Main Menu

The main window of DrAGoM , which is shown in Figure 2, has a simple and clean design. The idea behind this puristic presentation is to not overtax the user's first impression after launching DrAGoM . In the beginning, the main window only allows interaction with the main menu, i.e., the system bar displayed at the top of the screen. DrAGoM 's main menu contains the following menus and options:

System Includes commands to generate or load graph transformation systems and multiply annotated type graphs.

- The option *"New. . . "* offers commands to open the *Add-Rule* dialog (see Figure 3) or the *Create-Annotated-Type-Graph* dialog (see Figure 4).
- The options *"Load. . . "* and *"Save. . . "* offer commands to load/save graph transformation systems in the SGFformat or to load/save multiply annotated type graphs in the GXLformat. All commands in the option *"Save. . . "* become available, once a corresponding data structure has been loaded/created.
- The option *"Exit"* closes DrAGoM .

Algorithms Includes commands to compute rewritable materializations, strongest postconditions or to let DrAGoM perform an invariant check. DrAGoM employs the SMT solver Z3 to compute annotations for the rewritable materialization. If the required native libraries were not found during the booting process, all commands in the *Algorithms* menu are disabled.

- The option *"Compute. . . "* offers commands to construct a rewritable materialization and the strongest postcondition. To enable the construction for the rewritable materialization, the user has to create/load a graph transformation system and an annotated type graph first. A strongest postcondition can be constructed once the rewritable materialization has been created.
- The option *"Invariant check"* is available if and only if the user created or loaded both, a graph transformation system and an annotated type graph. This option causes DrAGoM to perform an invariant check.

Help Includes commands to provide further information.

- The option *"Visit website. . . "* redirects the user to the DrAGoM homepage, where the user can find additional information.
- The option *"About. . . "* opens a dialog with additional information with respect to the current version and the DrAGoM license.

Graph Transformation System Creation

The user can select *System* -> *New. . .* -> *Graph transformation rule* from the main menu, to create a new graph transformation system. The *Add-Rule* dialog appears, which is displayed in Figure 3. The dialog contains three graph panels which can be used to create the left-hand side, the right-hand side and the interface graph of a double-pushout graph transformation rule.

By double clicking on the interface graph panel, a node is added to the interface graph and two corresponding nodes are added to the left-hand side graph and right-hand side graph respectively. The same holds for loops, which are added via a right click on an existing node, and directed binary edges, which are added via a drag and drop movement from the source

node to the target node while pressing the right mouse button. This intuitive interaction works similarly for the left-hand side and right-hand side graph panels. Elements added to either side only belong to the corresponding graph. All interface elements are drawn black and white, elements which only belong to the left-hand side are colored blue and elements which belong only to the right-hand side are colored yellow.

Please note that edge labels are determined during their creation. The user can enter the edge label that one wants to use for the next edge, by entering the label into the text field which can be found at the top of the dialog.

To complete the creation process, the user can hit the *Apply* button located at the bottom of the dialog. The created rule now is displayed in the top part of the main window and it generates a new graph transformation system which consists of the single rule. A right click on the rule panel shows a pop up menu where the user has the option to either add a new rule, delete the currently displayed rule or switch between the displayed rules of our graph transformation system. Furthermore, the option to save the graph transformation system is now enabled in the main menu.

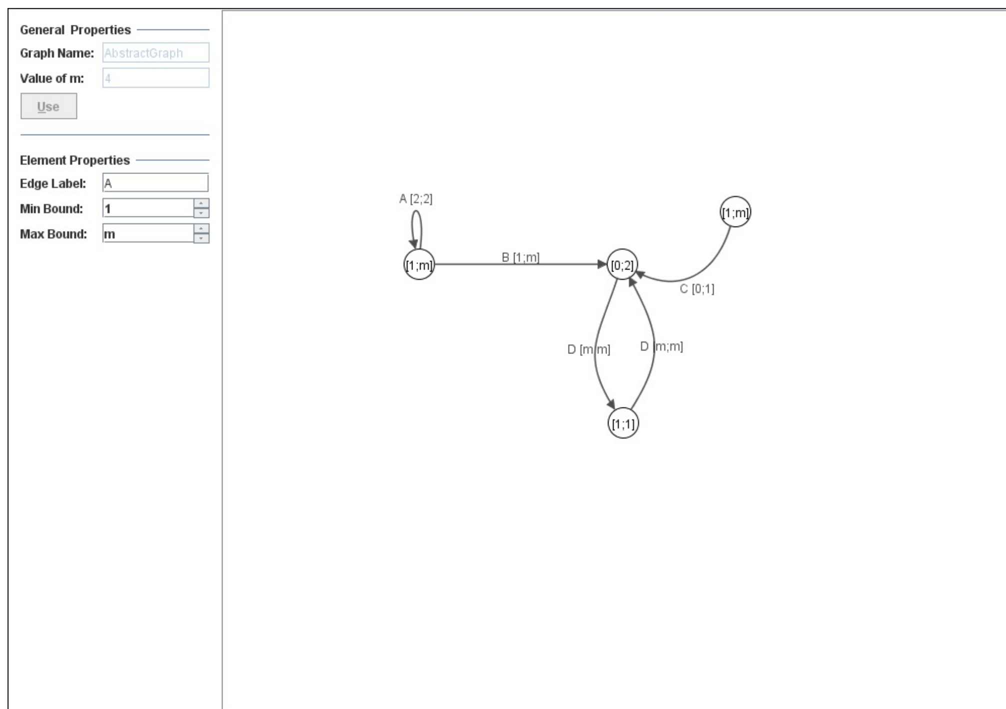


Figure 4: DrAGoM Create-Annotated-Type-Graph dialog

Multiply Annotated Type Graph Creation

Next, we explain how one can create an annotated type graph. For this purpose the user selects *System -> New... -> Annotated type graph* from the main menu. The *Create-Annotated-Type-Graph* dialog appears, which is displayed in Figure 4.

Before the user can create the graph structure, one needs to define a name for the new annotated type graph and choose a value m which represents the value $*$ ($m$any) of the ordered monoid $\mathcal{M}_n$ (see also [CH+19]). Subsequently, the user fills out the corresponding text fields which are located in the *General Properties* section on the top left of the dialog. Once the user filled out both text fields, a click on the *Use* button confirms the choice. Now the graph panel on the right side of the dialog gets enabled such that the user can create the annotated type graph.

The graph creation works similar to the creation of graphs for the graph transformation rule. Please note that multiplicities for graph elements are determined at the moment they are added to the graph. The values are taken from the respective combo boxes which are located in the *Element Properties* section on the left side of the dialog.

As soon as the user hits the *Apply* button, the annotated type graph is displayed in the bottom left corner of the main window. The option to save multiply annotated type graphs is now enabled in the system bar. The user can double click on the graph panel to maximize its view or right click on it to open a pop up menu.

The first option in the pop up menu is to add a multiplicity to the annotated type graph, i.e. to turn the annotated type graph into a multiply annotated type graph. If the user chooses this option the *Add-Multiplicity* dialog appears, which is shown in Figure 5.

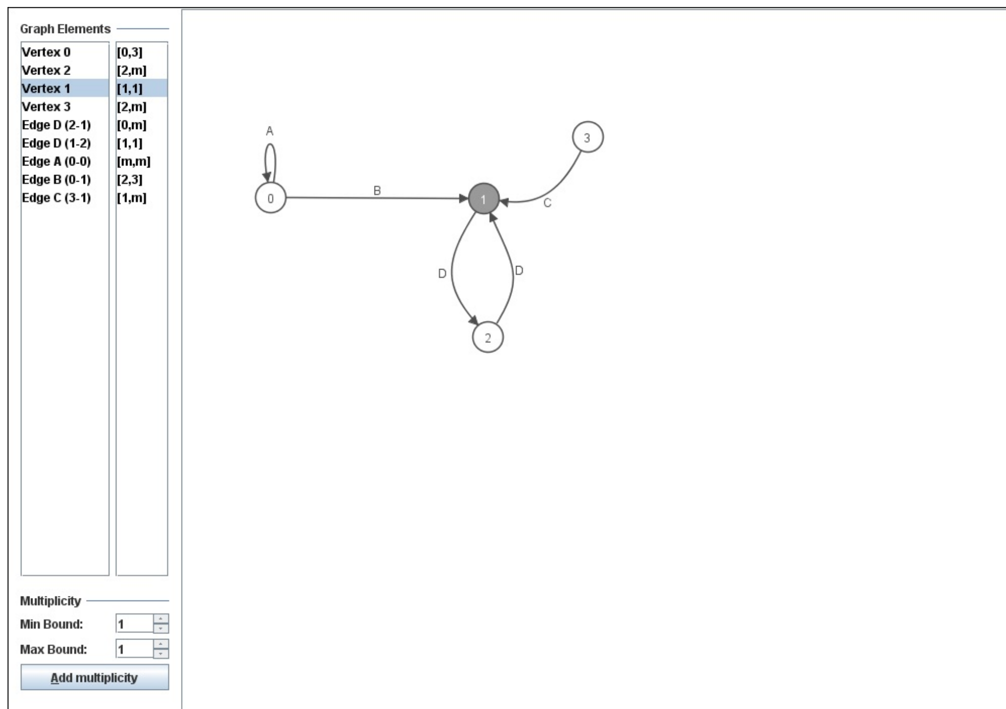


Figure 5: DrAGoM Add-Multiplicity dialog

To edit multiplicities for graph elements, the user can select the elements in the graph panel of this dialog. Every time that the user selects an element, two entries in the lists, which are located in the *Graph Elements* section on the left side of the dialog, are highlighted. The first list shows the element's corresponding identifier while the second list shows the current multiplicity to be added to the element. The default value for all elements is $[0, m]$ which corresponds to the annotation bounds $[0, *]$.

To change the multiplicity of an element one can select the desired bounds in the combo

boxes in the *Multiplicity* section located at the bottom left of the dialog. Afterwards, the user can hit the *Add multiplicity* button to overwrite the selected elements annotation bounds with the ones selected in the combo boxes.

Once the user is done, the dialog can be closed by hitting the *Apply* button. DrAGoM will check if the new multiplicity was already covered by an existing one in the set of annotations for this multiply annotated type graph. If it was not covered yet, the created multiplicity is added to the set, otherwise it is discarded and we get a notification.

The two remaining options in the pop up menu of the annotated type graph panel, allows the user to delete existing multiplicities from the set of available annotations or to switch the currently displayed multiplicity.

Rewritable Materialization Construction

Now, that the user has created both, a graph transformation system and a multiply annotated type graph, the option to compute rewritable materializations gets enabled in the menu bar (*Algorithms -> Compute... -> Materialization*). The graph transformation rule and the multiply annotated type graph displayed in the main window are being used for the computation. To proceed the user needs to choose a semi-legal base morphism φ from the left-hand side graph of the rule to the annotated type graph. The *Select-Base-Morphism* dialog appears (see Figure 6) which displays the set of available morphisms. DrAGoM filters the set of morphisms to only contain semi-legal ones. If there does not exist a semi-legal morphism, no graph contained in the graph language can be rewritten since none of them contain a concrete instance of the left-hand side.

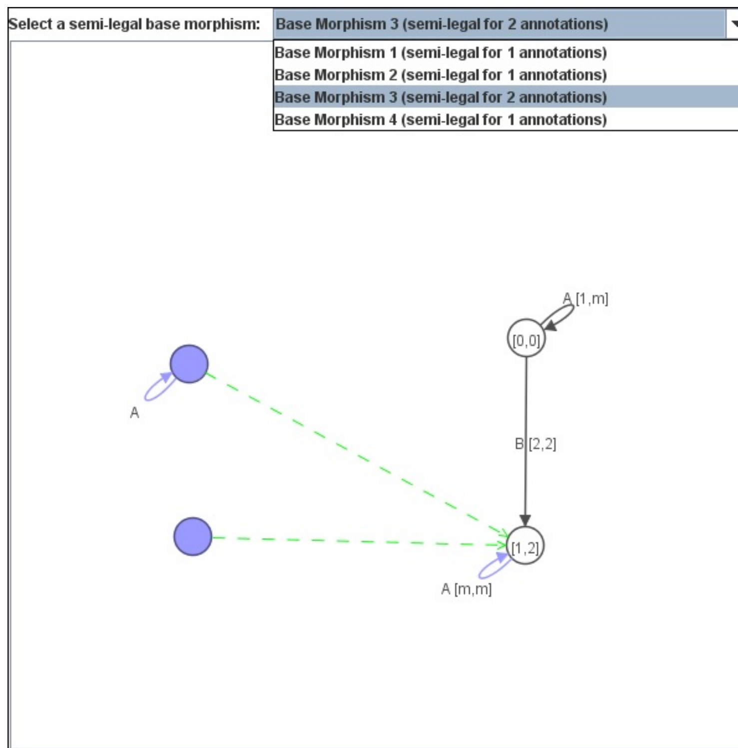


Figure 6: DrAGoM Select-Base-Morphism dialog

The user can select the semi-legal base morphism by choosing a corresponding entry in the combo box, located at the top of the dialog. Afterwards, the user can press the *Apply* button, which initializes the materialization algorithm. The rewritable materialization is displayed in the main window and the option to compute the strongest postcondition is enabled.

A right click on the materialization graph panel shows a pop up menu with options to either switch the currently displayed multiplicity for the rewritable materialization or to simplify its visualization. The simplification hides all elements annotated $[0,0]$ and all edges with a hidden source or target node.

Invariant Checking

To perform an invariant check the user chooses (*Algorithms -> Invariant check*). DrAGoM checks for all rules in the graph transformation system and for all semi-legal base morphisms if the language of the constructed postcondition is included in the initial graph language specified by the multiply annotated type graph. For the inclusion check, DrAGoM uses the sufficient condition introduced in [CKN19]. Whenever DrAGoM finds legal morphisms between the corresponding multiply annotated type graphs, we can infer the inclusion. However, if DrAGoM fails to find a legal morphism the inclusion could still hold. In this case, the rule, the rewritable materialization and the strongest postcondition graph with the multiplicity for which the check fails are displayed in the main window. Otherwise, in case of a successful check, the *Invariant-Proof* dialog appears (see Figure 7) in which the user can see every found legal morphism by choosing a rule and base morphism via the combo boxes located at the top of the dialog.

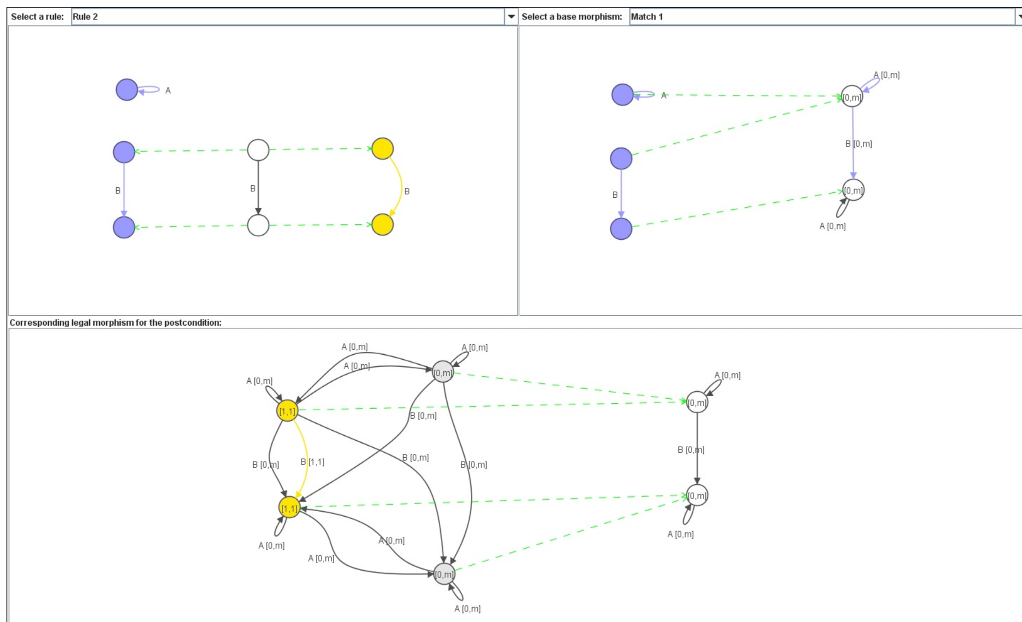
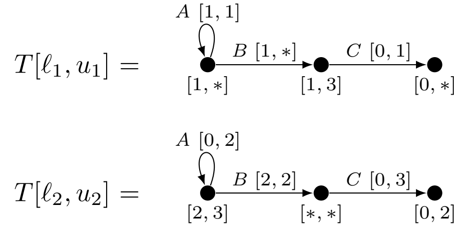


Figure 7: DrAGoM Invariant-Proof dialog

The GXL Format for Multiply Annotated Type Graphs

In this section we give an example for the GXL encoding of a multiply annotated type graph. The annotated graphs are stored into `.gxl`-files. A detailed description of the GXL format can be found on the website at `http://www.gupro.de/GXL/`.

Let the following multiply annotated type graph $T[M]$ be given, with the set of double multiplicities $M = \{(\ell_1, u_1), (\ell_2, u_2)\}$ over the annotation functor $\mathcal{B}^3$, i.e. the multiplicity of each graph element is indicated by an element of $\mathcal{M}_3 = \{0, 1, 2, 3, *\}$:



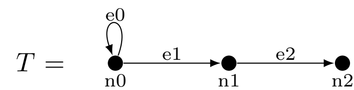
The multiply annotated type graph $T[M]$ is encoded in the following GXL format:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">
<gxl>
  <graph id="AbstractGraph">

    <node id="limit">
      <attr name="limit">
        <int>4</int>
      </attr>
    </node>

    <node id="n0">
      <attr name="min">
        <seq>
          <int>1</int>
          <int>2</int>
        </seq>
      </attr>
      <attr name="max">
        <seq>
          <int>4</int>
          <int>3</int>
        </seq>
      </attr>
    </node>
```

```
<node id="n1">
  <attr name="min">
    <seq>
      <int>1</int>
      <int>4</int>
    </seq>
  </attr>
  <attr name="max">
    <seq>
      <int>3</int>
      <int>4</int>
    </seq>
  </attr>
</node>

<node id="n2">
  <attr name="min">
    <seq>
      <int>0</int>
      <int>0</int>
    </seq>
  </attr>
  <attr name="max">
    <seq>
      <int>4</int>
      <int>2</int>
    </seq>
  </attr>
</node>

<edge id="e0" from="n0" to="n0">
  <attr name="label">
    <string>A</string>
  </attr>
  <attr name="min">
    <seq>
      <int>1</int>
      <int>0</int>
    </seq>
  </attr>
  <attr name="max">
    <seq>
      <int>1</int>
      <int>2</int>
    </seq>
  </attr>
</edge>
```

```
    <edge id="e1" from="n0" to="n1">
      <attr name="label">
        <string>B</string>
      </attr>
      <attr name="min">
        <seq>
          <int>1</int>
          <int>2</int>
        </seq>
      </attr>
      <attr name="max">
        <seq>
          <int>4</int>
          <int>2</int>
        </seq>
      </attr>
    </edge>

    <edge id="e2" from="n1" to="n2">
      <attr name="label">
        <string>C</string>
      </attr>
      <attr name="min">
        <seq>
          <int>0</int>
          <int>0</int>
        </seq>
      </attr>
      <attr name="max">
        <seq>
          <int>1</int>
          <int>3</int>
        </seq>
      </attr>
    </edge>

  </graph>
</gxl>
```

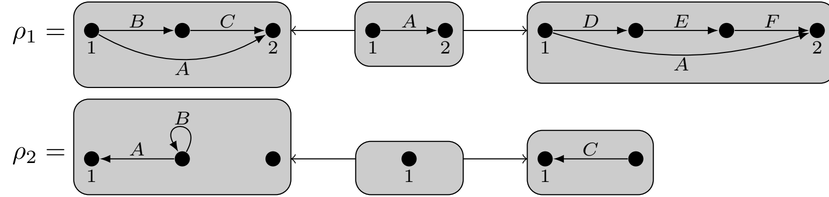
As evident from the encoding shown above, the GXLformat uses identifiers to encode the structure of the type graph. For completeness sake, we depict the identifier reference for each element of the encoded type graph $T[M]$:



The SGF Format for Graph Transformation Systems

In this section we give an example for the SGF encoding of a graph transformation system. The graph transformation systems are stored into `.sgf`-files, where one file in the SGF format can contain several graph transformation systems at the same time. A formal description of the SGF format is given in [Bru15].

Let the following graph transformation system $\mathcal{R} = \{\rho_1, \rho_2\}$ be given, where ρ_1 and ρ_2 are depicted below:



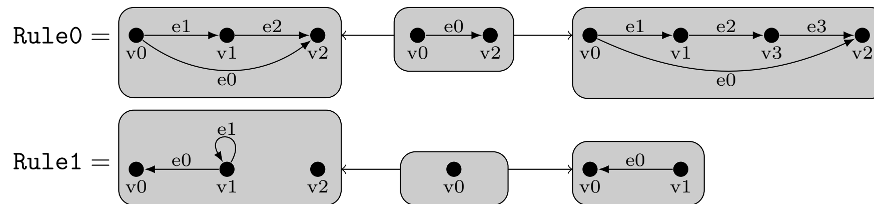
The SGF format represents a double-pushout rule using a corresponding morphism from the left-hand side graph to the right-hand side graph instead of saving two rule morphisms and the interface graph. Please note that the SGF format only supports injective rule morphisms. In the encoding, nodes are introduced implicitly via edge definitions, in contrast to distinct nodes which need to be added explicitly in the respective graph encoding.

The graph transformation system $\mathcal{R}$ is encoded in the following SGF format:

```
leftGraph0 = graph {
  e0:A(v0,v2);
  e1:B(v0,v1);
  e2:C(v1,v2);
};

rightGraph0 = graph {
  e0:A(v0,v2);
  e1:D(v0,v1);
  e2:E(v1,v3);
  e3:F(v3,v2);
};

Morphism0 = morphism from leftGraph0 to rightGraph0 {
  v0 => v0;
  v2 => v2;
  e0 => e0;
};

leftGraph1 = graph {
  e0:A(v1,v0);
  e1:B(v1,v1);
  node v2;
};

rightGraph1 = graph {
  e0:C(v1,v0);
};
```

```
Morphism1 = morphism from leftGraph1 to rightGraph1 {
  v0 => v0;
};

Rule0 = rule {
  left = leftGraph0;
  right = rightGraph0;
  morphism = Morphism0;
};

Rule1 = rule {
  left = leftGraph1;
  right = rightGraph1;
  morphism = Morphism1;
};

result = gts {
  rules = [
    Rule1,
    Rule0
  ];
};
```

For the sake of completeness, we depict the identifier reference for each element of the encoded graph transformation system $\texttt{result} = \{\texttt{Rule0}, \texttt{Rule1}\}$:



Bibliography

[Bru15] H. J. S. Bruggink. *Grez User Manual*. `www.ti.inf.uni-due.de/research/tools/grez`. 2015 (cit. on pp. 3, 14).

[CH+19] A. Corradini, T. Heindel, B. König, D. Nolte, and A. Rensink. "Rewriting Abstract Structures: Materialization Explained Categorically". In: *Foundations of Software Science and Computation Structures*. Ed. by M. Bojańczyk and A. Simpson. Cham: Springer International Publishing, 2019, pp. 169–188. DOI: `10.1007/978-3-030-17127-8_10`. arXiv: `1902.04809 [cs.LO]` (cit. on pp. 1, 8).

[CKN19] A. Corradini, B. König, and D. Nolte. "Specifying Graph Languages with Type Graphs". In: *Journal of Logical and Algebraic Methods in Programming* Vol. 104 (2019), pp. 176–200. DOI: `10.1016/j.jlamp.2019.01.005` (cit. on pp. 1, 10).

[HS+06] R. C. Holt, A. Schürr, S. E. Sim, and A. Winter. "GXL: A graph-based standard exchange format for reengineering". In: *Science of Computer Programming* 60.2 (2006), pp. 149–170 (cit. on p. 3).

[KRW02] B. Kullbach, V. Riediger, and A. Winter. "An Overview of the GXL Graph Exchange Language". In: *Software Visualization*. Ed. by Stephan Diehl. Vol. 2269. Lecture Notes in Computer Science. Springer, 2002, pp. 324–336 (cit. on p. 3).

[Win02] A. Winter. "Exchanging Graphs with GXL". In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. Lecture Notes in Computer Science. Springer, 2002, pp. 485–500 (cit. on p. 3).