StreamGraph Documentation
KBS Projekt

Florian Ziesche, Jakob Karge and Boris Graf
TU-Berlin
2016

***Abstract*—StreamIt is a textual stream processing language. StreamGraph is a project to put a graph-based programming environment layer on top of it. This is achieved by building on top of a pre-existing library for interactive graph representations and attaching a simple code generator as a backend to generate code that can be compiled and run with StreamIt. Coupled with a custom file format a simple coding environment was created that is able to generate code in a subset of the StreamIt language and finally act as a front end to compilation and execution of the resulting code.**

I. Introduction

THE StreamGraph project aims to present the StreamIt language simplified in a graphical environment. As such it is intended to be used by people who have little programming experience or do not wish to engage with the specifics of the StreamIt language and alleviates the amount of StreamIt specific semantic knowledge. StreamGraph does not contain every construct or possibility that StreamIt provides since it does not aim to fully rebuild the language in a graphical environment.

A. *StreamIt Language*

StreamIt is a language specifically build for applications which process a continuous stream of data. To achieve good performance on multi-core or multi-processor machines StreamIt enables highly parallel programming and programs, while trying to keep the programming as simple as possible.

The language features specific basic modules. This documentation focuses on the StreamGraph project and so only the subset of StreamIt modules which are realized in StreamGraph today are included and briefly explained. For further documentation on StreamIt see [1].

The most basic module of the StreamIt language and the only module that processes data is the filter. A filter, just as any other StreamIt construct, has exactly one input and exactly one output for data. It consists mainly of an init and a work function. The init function describes the filter's initial condition (e.g. initial values of variables). The work function describes the steps taken in one execution step of the filter and is called every time that the preconditions are fulfilled.

These preconditions are described by the pop, peek and push values. The pop value describes how many data packages are taken from the input of the filter in a single execution step of the work function. Similarly the pop value describes how many data packages are produced in the output of the filter in that execution step. The peek value describes how far back the filter needs to read in the input data, i.e. the minimum input queue length, since any value read has to be processed by previous filters.

1) StreamIt topology: Since filters alone do not specify in which order and if they are connected it is necessary to have language constructs which describe the connections and the structure of the processing. These constructs will be referred to as topological construct in the further documentation.

One of these constructs is the pipeline. A pipeline connects filters, pipelines and split-join constructs in a linear fashion. Each pipline has an init function in which the elements of the pipeline are added and which determines the order in which a datapacket is processed. Agraphical representation of a pipeline can be seen in fig. 1.

The second construct supported by StreamGraph is the split-join construct. This constucts splits the data packets of its input to multiple data paths on which other constructs are placed and joins these paths together at the end of every path. All in all a split-join construct has exactly one input and one output, as can be seen in fig. 2.

The StreamIt language generates a pipeline based structure. As such it is not possible to connect two filters or constructs in different pipelines (see Fig. 3.).
Since split-join constructs have exactly one input and one output and only appear in pairs it is not possible to have connections from one split-join pair to another pair(see Fig. 4) or to have an unequal numbers of splits and joins (see Fig. 5).

II. StreamGraph

The StreamGraph project is composed of multiple parts. In this section they are listed and explained and along with the reason behind the individual decisions that led to them.

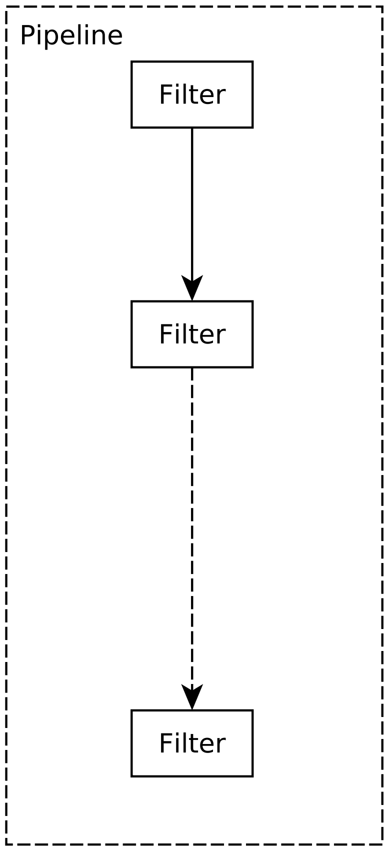

Fig. 1. Example of a pipeline

A. View

The GUI of StreamGraph includes three parts. At the top is the menu bar with the important functions for the project. The first submenu is, similar to other textual editors, the File menu with common functions to create new, open, save, save-as or to quit projects. In the Edit submenu it is possible to create new objects. In the Run submenu the project can be executed. The Debug submenu helps with debugging the project.

The view takes up the main part of the GUI window. It is based on the Gtk2::Ex::MindMapView Perl modul [2] and shows the objects and their connections. The view design is minimalistic: a plain surface to have a clear view and simple usability. There are no toolbars or other control windows. The graph structure can be completely edited with the mouse. A right click opens a pop-up menu with the same entries as in the Edit submenu. This allows the creation of an object with two mouse clicks. The surface can be scrolled with the scrollbars in all four directions, but can also be dragged with the left mouse button. It gives the user access to a larger editing area and to seperate objects into different groups.

At the bottom of the GUI is a dynamic status bar for program feedback for the user. Errors or important information are shown there.

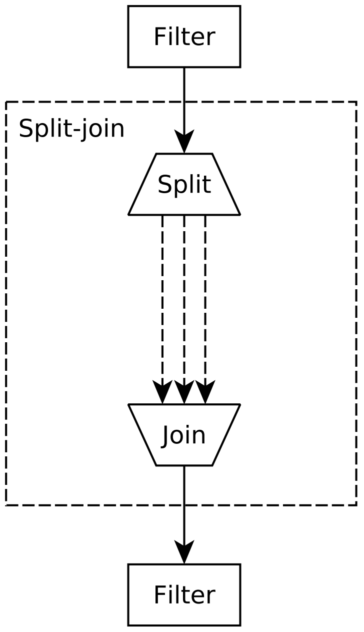

Fig. 2. Example of a split-join construct

1) Objects: Objects in the view area are nodes of the program graph. There are four different types: Filter, Parameter, Subgraph and Comment. Each type has a different geometric form. Connections between the nodes are visualized as lines with a direction arrow. Connections have circular pins at both the beginnig and the end. Pins on nodes with no connection line show that this node has an open input or output. A new connection is created by clicking on an out pin and dragging to another node. All possible connections are visualized with bigger pins. If the mouse hovers over a node to which the connection is possible, the arrow is glued to the available pin.

A connection can be marked through hovering over it with the mouse. This is visualized with a much thicker line. A node can be marked with a left mouse click. To mark more than one node a right mouse click on a node adds the node to the marked nodes. Another intuitive way is to draw a selection rectangle with a right mouse click on the view surface. Marked objects can be deleted with a DEL key press.

2) Property Window: A double left click on a node opens its property window. It contains all the parameters from this node. For example a filter node has three tabs: Join, Filter and Split. In the filter tab the user has the possibility to set the input and output types and also to edit the filter source code.

B. Model

The StreamGraph layer on top of StreamIt needs data structures to work with. Their details contain much of the design of StreamGraph as an abstraction over StreamIt. The various data objects of StreamGraph are now examined in further detail:

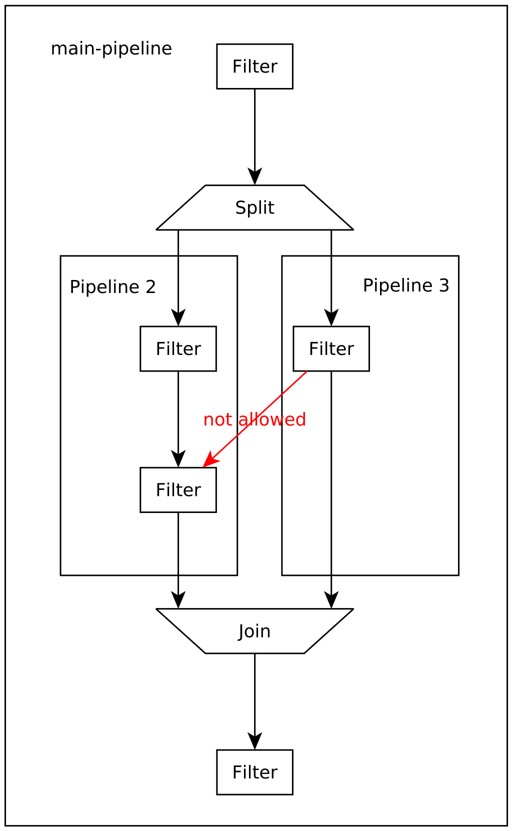

Fig. 3. Example for a non permissable connection between pipelines

1) Filter: A filter in StreamGraph is a combination of two StreamIt constructs, the filter and split-join constructs. The combination of the two into one object has one important advantage: It allows filters to have more than one input or output. The filter part is a very straight wrapper around a StreamIt filter, with all the fields exposed in the property window, and a direct assembly into a StreamIt filter on code generation.

The split-join part however is reversed and a StreamGraph filter includes a join before and a split after. This way the multiple inputs of a filter in StreamGraph are joined to form the required single input for a StreamIt filter. Similarly the data packages coming from the single output of the filter code are spread among the outputs of the StreamGraph filter node. A depiction of a filter in StreamGraph can be seen in 6. The join and split is configured through the filter's property window, along with the filter code.

2) Subgraph: Subgraphs are slightly special in that they open another .sigraph file for inclusion in the program. From the outside though they behave just like normal filters and have the same properties and can be used the same way. They encapsulate StreamIt code in a similar way that functions or methods in textual code do.

Currently there are limits to their implementation though:

- Changes aren't reflected in code generation until saved.
- No synchronization between multiple instances/windows of the same subgraph.

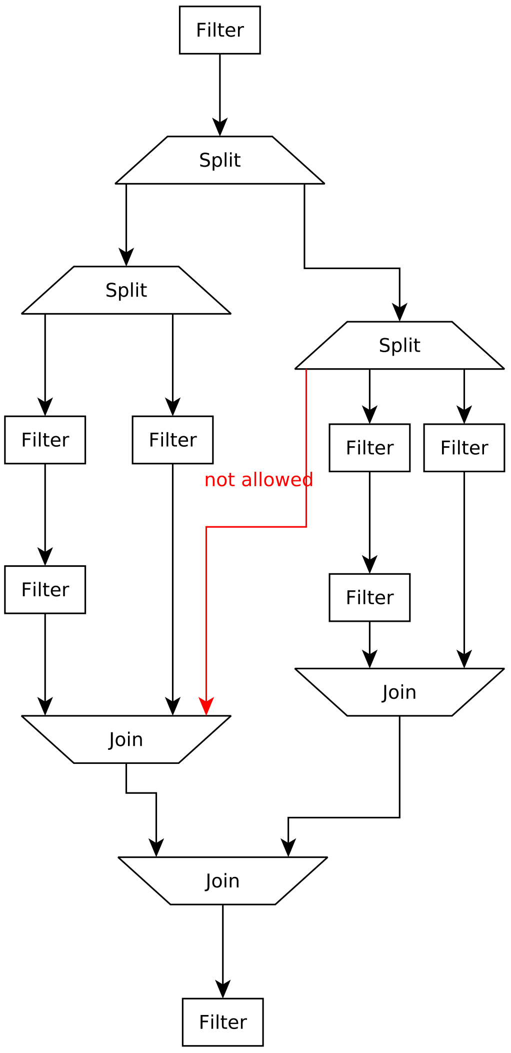

Fig. 4. Example for a non permissable connection between split-join pairs

- Parameters connected to subgraphs are forwarded to *all* nodes in the subgraphs, including sub-subgraphs, leading to a lot of potentially unnecessary parameter connections.

3) Parameter: A parameter is a single static named value that can be used by filters (and subgraphs) and is independent of the data stream. Its significance lies in the use for a DRY programming style.

4) Comment: A comment is an inactive, unconnectable node that allows to place text anywhere on the canvas and thus to document the code rather freely, or to otherwise help in understanding the program.

C. Graph hierarchy and Namespaces

StreamGraph introduces Subgraphs as a new feature for structuring code. Subgraphs act like normal filter nodes in the parent graph, but are actually a pipeline. This makes use of the fact that pipelines and filters can be used in the same context in StreamIt, namely in other pipelines and in split-join constructs.

The passing of parameters down through a subgraph hierarchy makes the use of namespaces necessary. These are fully

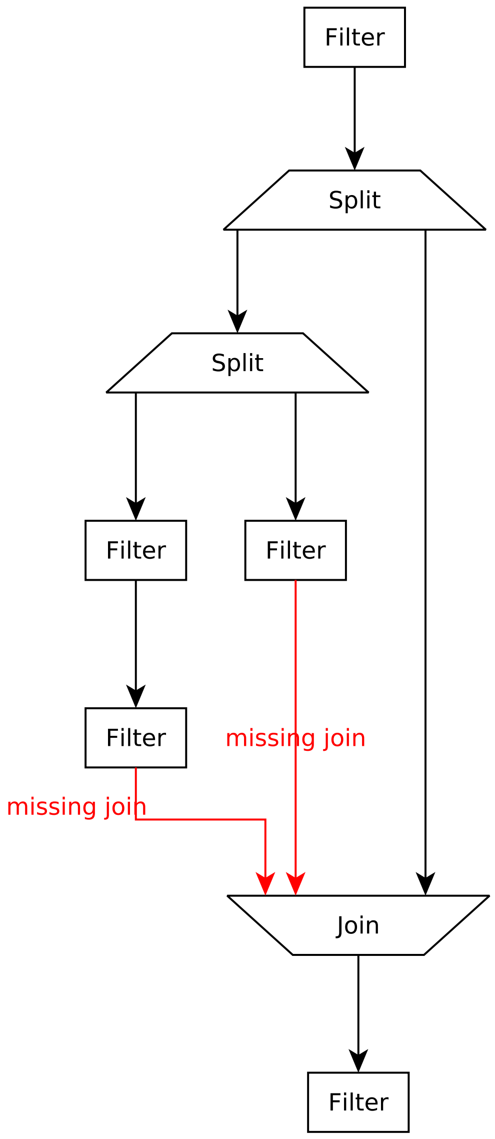

Fig. 5. Example for a unequal number of splits and joins (in this case a missing join)

automatic though and the user has no interaction with them. Any parameters in a subgraph are prepended with the name of the subgraph, to differentiate it from possibly identically named parameters being passed in from the parent graph.

D. GraphCompat

The StreamIt topology, as discussed in I-A1, is not only a directed Graph but has additional restrictions. Some of these are not strictly necessary and some effort was made in StreamGraph to allow for graphs that are not strictly StreamIt-compatible and to automatically convert them.

This task is taken on by the GraphCompat class. It transforms a given StreamGraph graph, coming from the user interface, into a new graph which is compatible with the StreamIt topology.

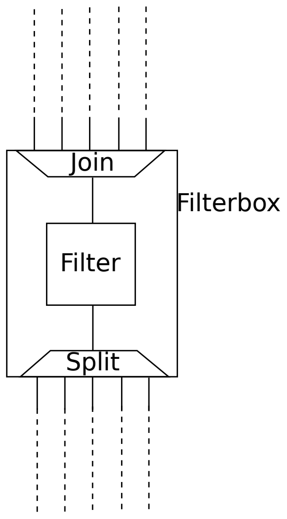

Fig. 6. A node in the StreamGraph

At the moment it takes care of the following 3 transformations:

- Identities
  StreamIt does not allow direct connections between a split and a join without a filter or a pipeline. In StreamGraph such a connection is allowed. In this case GraphCompat adds an identity filter instead of the direct connection between the split and the join, as can be seen in Fig. 7. This filter copies his input to his output without changing the content in any way.

- Sources and Sinks
  A StreamIt graph must start and end with exactly one void-typed input and output respectively. StreamGraph also makes the void-typed requirement but does not restrict the graph to just one source and one sink. Multiple sources and sinks are combined by adding ”Void Sources” and ”Void Sinks” to the ends of the graph when necessary. See Fig. 8 for an illustration of this.

- Subgraphs
  Subgraphs as they are, are a feature of StreamGraph, and don’t make use of StreamIt’s joint-compilation features. The results are essentially the same though: Flattening the graph hierarchy into one single graph is possible because a subgraph is a pipeline and using a subgraph is identical to adding a pipline inline.
  The subgraph transformation also takes care of namespacing the parameters as discussed above.

If, for one reason or another, the a part of the graph cannot be transformed it is simply left out of the transformed graph and thus code generation. This feature is quite essential, because otherwise a quick draft on the side of a few nodes would quickly invalidate the whole program. But being able to test the program as far as possible during development is necessary for a more robust process.

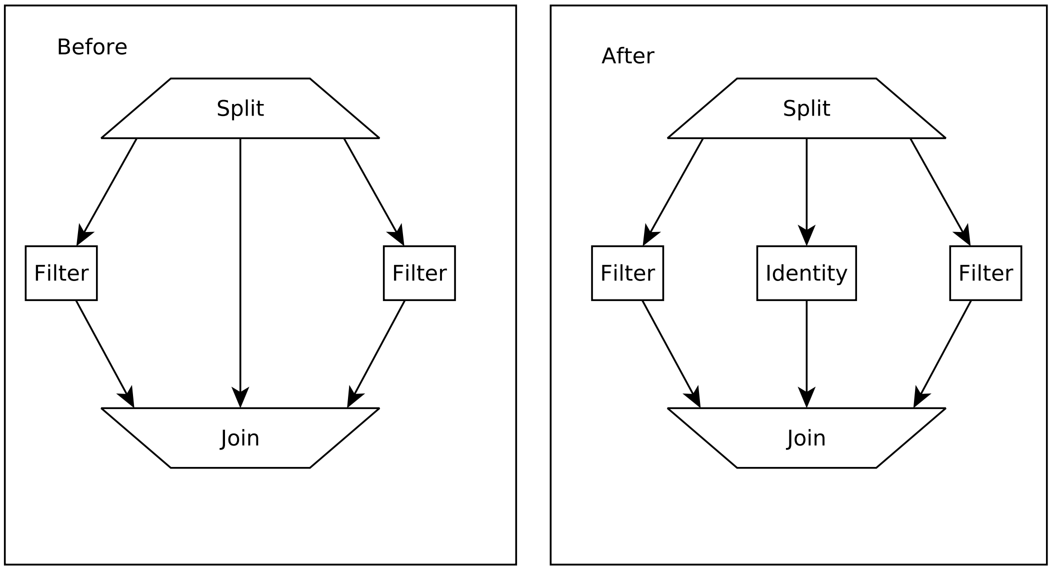

Fig. 7. Example of adding an identity filter

E. CodeObject

The CodeObject is an interface class which is extended by the Pipeline class, the SplitJoin class and the Parameter class. It is a general class for all implemented topological constructs of the StreamIt language. Filters are not CodeObjects since they do not describe the structure of the graph but rather are the nodes of the Graph.
While Parameters also do not describe the structure of the graph, they may be needed by multiple CodeObjects and are needed in different configurations while the code is generated. As such the parameters are an exception to the general rule that CodeObjects describe the structure of the graph. CodeObjects except for parameters reference to other CodeObjects or filters. As such a single CodeObject can have an entire graph ready for code generation stored in a manner that already has the necessary order and hierarchical structure of a StreamIt program.
Since StreamGraph does not use the concepts of pipelines and split-joins in the way StreamIt expects these constructs the codeObjects rebuild the graph with pipeplines and split-joins in a indirect recursive fashion, that resembles StreamIt's own way of nesting split-joins and pipelines into each other. At the same time this is the reason for creating the codeObjects in this way, since it automaticly generates the needed and desired hierarchical structure of a StreamIt program. This simplifies the generation of the code for a specific codeObject, which demands the names of all directly nested codeObjects. Figure 9 shows an example of a conversion from a simple StreamGraph graph to a CodeObject structure.

1) Parameter: The Parameter class is an approximation of the way parameters are handled in the StreamIt language. Parameters are an exception in the sense that they are not topological constructs. In the visible graph they are represented by nodes. Parameters do not reference other CodeObjects or filters. They only are referenced from other objects (filters or CodeObjects). Every parameter as a CodeObject exists exactly once.
For better readability in the StreamIt source file the parameters are written as global variables in the main pipeline and are passed through to the other code objects.

2) Pipeline: Every StreamIt program contains at least one pipeline, the main pipeline, in which all other constructs and filters are placed. A pipeline CodeObject stores the objects in the order in which a data packet in StreamIt will be processed. These objects may be other CodeObjects like split-join constructs or filters. If the need for a split-join construct is detected while creating the pipeline a split-join construct is created. When the creation is complete the pipeline continues its creation and eventually returns.
Pipelines encompass the maximum possible amount of nodes to minimize the amount of needed pipelines and with that minimizes the amount of code that has to be generated. An example of a pipeline can be seen in fig. 1.

3) SplitJoin: Split-join constructs are the most obvious parallel construct in the StreamIt language. A split-join construct has at least two parallel paths. Every path in a split-join is modeled as a pipeline. As such when a split-join is created, a pipeline is created for every path. In StreamIt it is not necessary to create a pipeline if there is exactly one filter on a path between a split-join pair. StreamGraph creates a pipeline on every path of a split-join pair for the sake of convenience. The approach to generate a pipeline in every case should not increase the execution time significantly and does not change the order of execution. Because of these reasons it was decided to not include the feature to only create pipelines in split-join constructs when strictly necessary.
A simple example of a split join is shown in fig. 2.

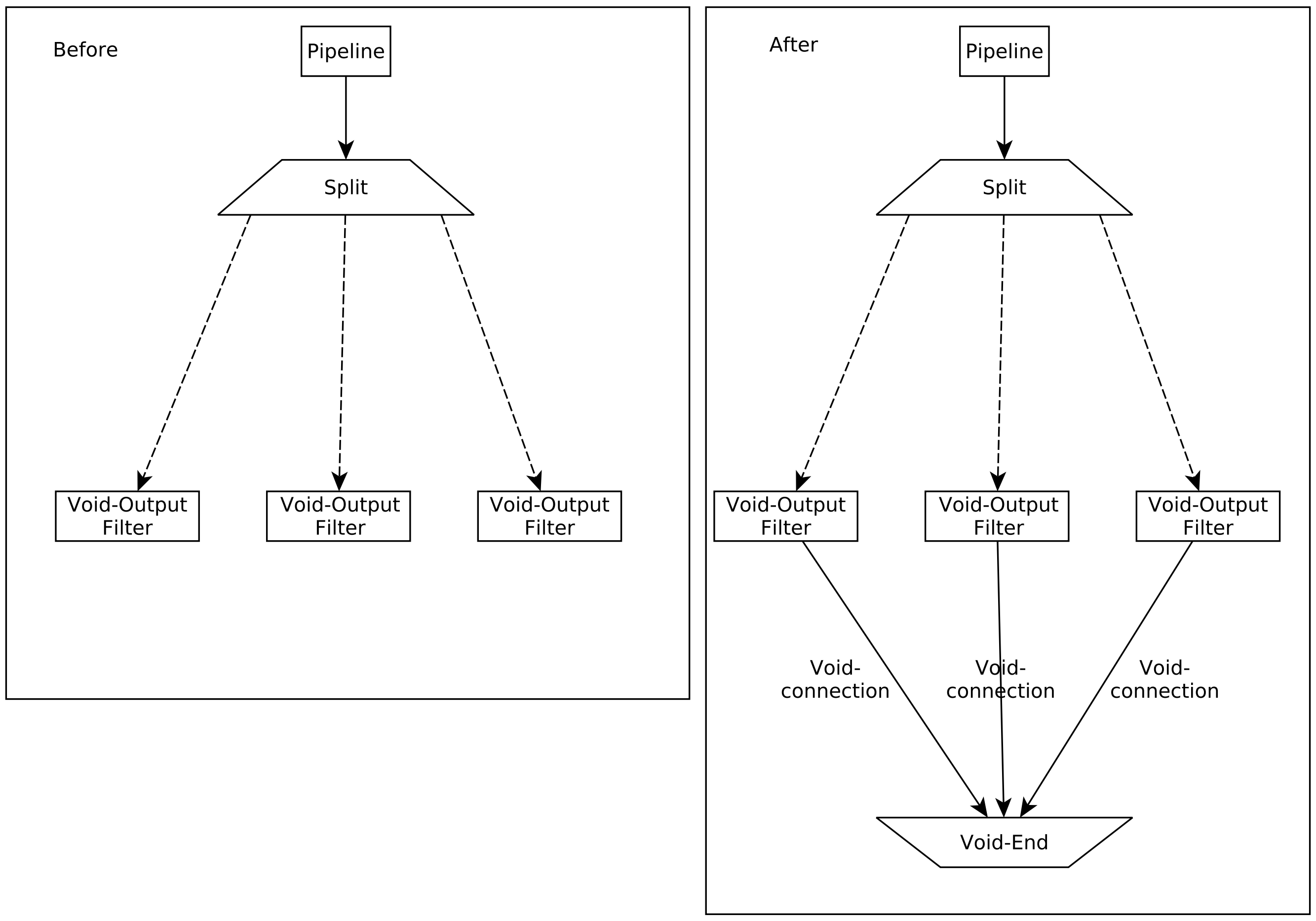

Fig. 8. Example of adding a void-end

The generation of split-joins is relatively easy as long as the graph is compatible (for StreamIt compatibility see I-A1). As soon as the graph is not StreamIt-compatible it becomes increasingly difficult to detect and mend. Simple mending methods like copying the node of a cross-connection between paths of a split-join is only possible if the nodes or filters are stateless. Since statelessness and requirements of other mending methods can only be ascertained through parsing and checking of the code from the respective constructs, which would be beyond the scope of this project, mending is not included in the current version of StreamGraph.

StreamGraph does not support specific orders of execution for paths in a split-join construct. That means, that at the time of the first processing of the paths a specific but random order is established. This order is then the order in which these paths will be processed for the whole code generation but it may be different between different executions of the code generation.

F. Code generation

At first the code generation generates the code for every filter in the graph. This is possible because filters are the basic modules of StreamIt and due to the fact that every meaningful StreamIt program has at least one Pipeline. As such StreamIt as a language does not allow programs consisting only of filters. Without a pipeline no connection between these filters is possible (see I-A1 for further explanation).
The reasons behind generating filters at first are that in this way it is not possible to 'forget' a filter and if a filter is generated that is not needed in the execution of the program it does no harm. Secondly it simplifies re-use of topological constructs, since these only need the name of the component and not the complete code.

After generating all filters the code generation continues with the main CodeObject which is a pipeline. While generating the code of a topological construct (a pipeline or a split-join-pair) it may be necessary to generate the code of another such topological construct since it is a part of the first construct. In that case the code generation of the current construct is paused and the code generation of the contained construct is started. By the time the code generation of the

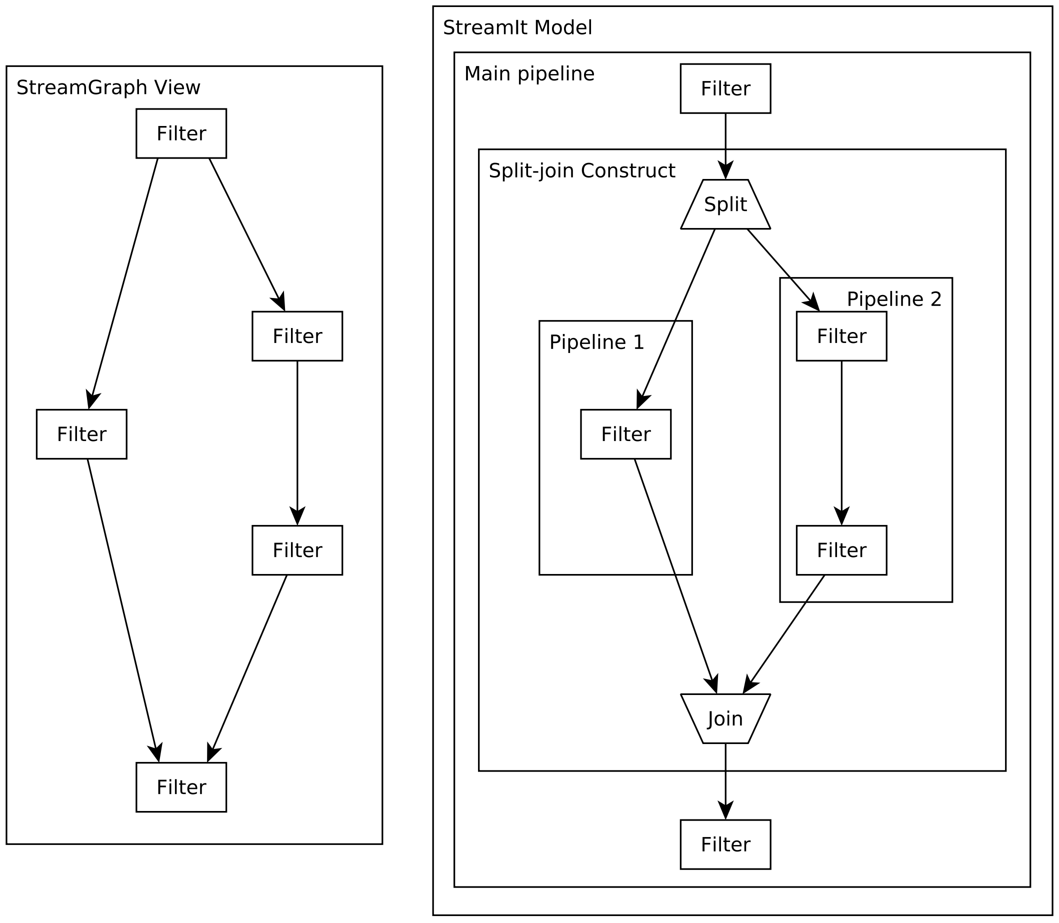

Fig. 9. Example of StreamGraph to StreamIt conversion

contained construct is finished the code generation of the containing construct is continued.

The code generation arranges filters and topological constructs according to their type. Thus StreamIt code generated by StreamGraph lists firstly all filters, then all pipelines and lastly all split-join constructs.

G. Improvment and Extension

1) Features: The StreamGraph project met nearly many of its goals but there are still possibilities for future improvements and Bugs that should be fixed.

- Complete representation
  The StreamGraph project aimed to implement a subset of the functionality of StreamIt. In the future it would be possible to implement the StreamIt features that weren't in the original scope of the project. To be specific these would be the FeedbackLoop, message-passing and the re-initialization feature.

- Further automatic compatibility
  StreamGraph builds StreamIt code from its graph. Before code generation is started made the graph is made compatible with the StreamIt topology. At the moment only minor adjustments can be made as described in II-D. It is possible to make more far-reaching transformations to the graph like the automatic adding of splits or joins in certain special cases. The main case for that can be seen in fig. 5.
- Dynamic subgraphs
  Currently when transforming the graph for code generation StreamGraph loads subgraphs of a graph from their file. As such changes in a subgraph are only recognized in the main graph if the subgraph is saved before generating the code. A desirable feature would be to be able to dynamicly update subgraphs in the main graph without the need to save the subgraph.

2) Known bugs: The implementation of the StramGraph project is not perfect and as such includes bugs and the need for minor additions which would be necessary for a finished product.

- No incorrect graphs
  At the moment it is possible to build non-correct graphs in the view. Optimally the user interface would correct any attempts to build an incorrect graph. The StreamGraph project already prohibits easy to detect errors while

connecting nodes.

III. Conclusion

StreamGraph aimed to simplify the use of StreamIt. Specifically to remove the need for deep understanding of the syntax needed to represent topological constructs. As a graph-based language it inherits some of the advantages of graphical programming, namely a better overview of overall structure of a program and more intuitive access for beginners. It also tried to translate a pipeline-centered to a directed-graph-based paradigm. Enforcing the symmetry required of a pipeline structure is daunting and was not completely realized (See II-G1 and II-G2).

At this point the StreamGraph project was still quite successful in showing that a fairly effective graphical representation and editing environment can be created for StreamIt. Many basic constructs could be realized and intuitively represented and a complete toolchain was created. It is possible to start from an empty graph, add and edit graph constructs, combine several graphs and then create a fully running executable, all from within StreamGraph.

From a practical standpoint, in the current state, StreamGraph is certainly limited though and its usefulness as an actual develpment tool is debatable. It does not have enough ease-of-use and completeness to be satisfying as a regular environment for coding. Add to this the StreamIt language itself, which, while fairly useful in it's area of expertise, is not being very actively developed anymore and showing its age.

However, given the time frame of the project of around 4 months, and being written by 3 people, it seems like it certainly has value. As a learning experience and a study and possibly an inspiration for the development of future graph-based programming tools.

Acknowledgment

The authors would like to thank the section *Kommunikations- und Betriebssysteme* (KBS) of the TU Berlin for the opportunity for this project.
We would especially like to thank Stefan Sydow for the guidance, help and tips throughout the process (and for telling us to cut some features).

Appendix A
Installation

Contains installation instructions on how to get StreamGraph up and running.

Appendix B
Developer documentation

Provides code documentation for the majority of the Perl modules of the StreamGraph project.

References

[1] William Thies, Michael Karczmarek, and Saman Amarasinghe, *StreamIt: A Language for Streaming Applications*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

[2] http://search.cpan.org/~hemlock/Gtk2-Ex-MindMapView-0.000001/lib/Gtk2/Ex/MindMapView.pm

Installation

Prerequisites

StreamGraph

- Download: https://gitlab.tubit.tu-berlin.de/streamgraph/streamgraph/repository/archive.zip?ref=master
- or `git clone` https://gitlab.tubit.tu-berlin.de/streamgraph/streamgraph.git

the latest version of this repository and extract it anywhere you like. This location will be referred to as `streamgraph/`.

The rest of the downloads (*Java 5* & *Streamit*) can be downloaded/extracted into that directory (it's not mandatory though).

Java 1.5

StreamIt is old software and needs a version 5 JDK.

- In Arch-Linux-based distributions you can install `jdk5` from the AUR. Don't set it as default Java version, just use the location `/usr/lib/jvm/java-5-jdk/` when configuring below.
- For other distributions download Java 5 via these `curl` commands. Choose the 64- or 32bit download.

```
curl -b oraclelicense=a -LO \
 http://download.oracle.com/otn-pub/java/jdk/1.5.0_22/jdk-1_5_0_22-linux-amd64.bin

curl -b oraclelicense=a -LO \
 http://download.oracle.com/otn-pub/java/jdk/1.5.0_22/jdk-1_5_0_22-linux-i586.bin
```

(Note that the `-b ...` flag implies that you accept the Oracle Binary Code License Agreement!)

Run the downloaded file and the JDK will self-extract after you accept its license. The resulting location will be referred to as `jdk1.5.0_22/`.

Streamit

- Download the *StreamIt* binary release http://groups.csail.mit.edu/cag/streamit/restricted/streamit-2.1.1.tar.gz and extract the containing folder. This location will be referred to as `streamit-2.1.1/`.

- Since the *StreamIt* "binary" release isn't actually all binary, you still have to build some of it—and patch it first (edit the paths!):

  ```
  cd streamit-2.1.1
  patch -p0 -i streamgraph/streamit-build/streamit211-fix-init_instance-save_state.patch
  export JAVA=jdk1.5.0_22/bin/java
  export STREAMIT_HOME=/streamit-2.1.1/ # This path needs to be ABSOLUTE!
  ./configure
  make CFLAGS='-fpermissive -O2 -I.' # Add -fpermissive flag
  ```

Note that *StreamGraph* doesn't actually use *StreamIt*'s `streamit-2.1.1/strc` compiler perl script, but a slightly customized version, located at `streamgraph/source/resources/sgstrc`.

Perl Packages

You will need to install the following perl packages:

Module	Debian Package	Arch Package
Gtk2	libgtk2-perl	gtk2-perl

Module	Debian Package	Arch Package
Gnome2::Canvas	libgnome2-canvas-perl	gnomecanvas-perl
Graph	libgraph-perl	perl-graph *(AUR)*
Glib	libglib-perl	glib-perl
Moo	libmoo-perl	perl-moo
GraphViz	libgraphviz-perl	perl-graphviz
YAML	libyaml-perl	perl-yaml
Set::Object	libset-object-perl	perl-set-object *(AUR)*
Data::Dump	libdata-dump-perl	perl-data-dump

- Debian: `libgtk2-perl libgnome2-canvas-perl libgraph-perl libglib-perl libmoo-perl libgraphviz-perl libyaml-perl libset-object-perl libdata-dump-perl`
- Arch: `gtk2-perl gnomecanvas-perl perl-graph glib-perl perl-moo perl-graphviz perl-yaml perl-set-object perl-data-dump`

Configuration

- Create a file at `streamgraph/source/streamgraph.conf` and fill in the absolute paths to the locations you created above so the file looks like this (`---` must be included):

  ```
  ---
  java_5_dir: 'jdk1.5.0._22'
  streamit_home: 'streamit-2.1.1'
  ```

- (Alternatively you can just start *StreamGraph* at this point and it will create an empty configuration in the right place for you. *StreamGraph* will run, but it will not compile with *StreamIt* until you fill in the paths.)

Result

Your *StreamGraph* directory might look somewhat like this now:

```
streamgraph
   jdk1.5.0_22
   streamit-2.1.1
   streamit-build
   source
       bin
           streamgraph.pl
       lib
       helloworld.sigraph
       streamgraph.conf
       ...
   doc
   README.md
   ...
```

streamgraph.pl

This is the main executable of **StreamGraph** which contains most of the UI control and the necessary boilerplate. Usage:

```
$ perl path/to/streamgraph.pl path/to/graph.sigraph
```

Methods

`create_window($file, $isSubgraph, $parents, $parent_item)`

This method creates the main window and all other GUI elements. All elements are included in the %main_gui hash:

Main window $main_gui{window} = (Gtk2::Window)

Scroller window which includes the view. Allows to scroll the view. $main_gui{scroller} = (Gtk2::ScrolledWindow)

View shows the graph items. $main_gui{view} = StreamGraph::View->new(aa=>1);

Main box inside the main window which includes menu bar, scroller and terminal box. $main_gui{menus} = (Gtk2::VBox)

Item Factory allows to create new graph items. $main_gui{factory} = (StreamGraph::View::ItemFactory)

Node Factory allows to create StreamGraph::Model::Node objects. $main_gui{nodeFactory} = (StreamGraph::Model::NodeFactory)

Config parameters. $main_gui{config} = StreamGraph::Util::Config->new();

Save file dir. $main_gui{saveFile} = (String)

`scroll_to_center($main_gui)`

This method sets the (x,y)=(0,0) position to the center of the window.

`_test_handler($main_gui, $item, $event)`

This method handles mouse events on graph items. Actions which are realised: Open property window, mark or drag the item, open new main window for Subgraphs.

`_window_handler($main_gui, $item, $event)`

This method handles mouse events on the main window. Actions which are realised: Open pop up menu, scroll the view, close a temporal connection by lieving the window, draw a select rectangle.

`create_menu($main_gui,$isSubgraph)`

This method creates a Gtk2::VBox with included menu bar.

StreamGraph

StreamGraph is a graphical frontend to the StreamIt language. This is the main package which contains all code apart from the executable. The code is structured as such:

Model

Modules in the Model package are mostly data classes and present the structural layout of this code. Node objects handle the various types of graph items that StreamGraph has and CodeObject is the basis for the work that CodeGen does when generating StreamIt code. You could see them as "before" and "after" representations of the code.

View

The code in the View package is mostly the Gtk2::Ex::MindMapView which was heavily modified in a few places (e.g. `View`, `Graph`, `HotSpot`) and mostly left as-is in others (e.g. `Border`) and sometimes simply disabled and/or deleted (e.g. the automatic layouting).

The classes that were not modified by us are left out of the technical documentation, but are still documented in the source and (in their original form) at the Gtk2::Ex::MindMapView project and can be looked up there.

Owing to the integrated nature of Gtk2::Ex::MindMapView the View also takes on much of the control and data flow tasks. View::Graph is the central graph topology module and a wrapper around the Graph- and Graph::Directed modules. View::HotSpot and View::Connection are responsible for most of the work needed to connect two nodes. View::Item is a display and topology wrapper for each node in the graph.

Code Generation

The modules GraphCompat, CodeGen and CodeRunner take care of the backend of this program and transform and run the graph that is created via the GUI.

Util

Various utility functions and classes are collected here – mostly file I/O and basic data structure operations, as well as debugging helpers.

StreamGraph::Model::Saveable

Base class for objects that should be serialized when saving the graph.

Properties

`saveMembers` (list[String])
: A string list of members that should be saved.

Methods

`StreamGraph::Model::Saveable->new()`
: Create a StreamGraph::Model::Saveable.

`yaml_dump()`
: `return` a blessed YAML wrapped version of `$self` with knowledge of members to be saved and their order.

  See documentation of YAML module for details.

StreamGraph::Model::Node

Base class for Streamgraph graph objects

Properties

`name` (String)
: The display name of the node

`id` (String)
: The random unique ID for this node.

`x` (Integer)
: The horizontal position of this node in the editing area.

`y` (Integer)
: The vertical position of this node in the editing area. Higher is further down.

`saveMembers` (list[String])
: The properties which are saved when the graph is saved to a file.

Inherited from StreamGraph::Model::Saveable

None.

Methods

`StreamGraph::Model::Node->new($name=>$name, $id=>$id, $x=>$x, $y=>$y)`
: Create a StreamGraph::Model::Node.

`isFilter()`
: `return` Boolean

  Check if node is instance of subclass StreamGraph::Model::Node::Filter.

`isSubgraph()`
: `return` Boolean

  Check if node is instance of subclass StreamGraph::Model::Node::Subgraph.

`isDataNode()`
: `return` Boolean

  Check if node is instance of subclass StreamGraph::Model::Node::Filter or StreamGraph::Model::Node::Subgraph.

`isParameter()`
: `return` Boolean

  Check if node is instance of subclass StreamGraph::Model::Node::Parameter.

`isComment()`
: `return` Boolean

  Check if node is instance of subclass StreamGraph::Model::Node::Comment.

`is_split()`

`return` Boolean

Check if node has more than one successor.

`is_join()`

`return` returnvalue

Check if node has more than one predecessor (only filters are considered).

`resetId()`

`return` new ID (String)

Set a new random ID for this node.

Inherited from StreamGraph::Model::Saveable

`yaml_dump`

StreamGraph::Model::NodeFactory

Create general Nodes and some standardized nodes for StreamGraph::GraphCompat.

Properties

None.

Methods

`StreamGraph::Model::NodeFactory->new()`

Create a StreamGraph::Model::NodeFactory.

`createNode(@attributes)`

`return` a new node of type `$attributes{"type"}`.

Takes a hash of `%attributes` to pass to the new node.

`createIdentity($datatype)`

`return` a new node of type StreamGraph::Model::Filter.

The created filter will have the work section of

```
push(pop());
```

and so will pass on all data packets unchanged.

This function is used by StreamGraph::GraphCompat.

`createVoidEnd($type, $count)`

`return` a new node of type StreamGraph::Model::Filter or 0 on error if the `$type` is invalid.

`$type` is a string and can either be `"source"` or `"sink"`.

`$count` determines how many outputs (source) or inputs (sink) the node will have.

StreamGraph::Model::Node::Filter

The basic code component in StreamGraph. Joins data from its multiple inputs and processes it in its work code. Then the data gets distributed over its outputs.

This class is mainly a data structure and holds all properties required to generate the filter.

Properties

`initCode` (String)
: The StreamIt code of the init function written in the filter from the user.

`workCode` (String)
: The StreamIt code of the work function written in the filter from the user.

`globalVariables` (String)
: The global variables text written in the filter from the user.

`timesPush` (Integer)
: The number of times data packets are pushed to the output in one cycle.

`timesPop` (Integer)
: The number of times data packets are popped from the input in one cycle.

`timesPeek` (Integer)
: The highest number which is peeked in one cycle.

`joinType` (String)
: The (StreamIt) description of the join. May be void or round robin.

`splitType` (String)
: The (StreamIt) description of the split. May be void, round robin or duplicate.

`inputType` (String)
: The StreamIt type of the input.

`inputCount` (Integer)
: The number of incomming data connections.

`outputType` (String)
: The StreamIt type of the output.

`outputCount` (Integer)
: The number of outgoing data connections.

`saveMembers` (list[String])
: The properties which are saved when the graph is saved to a file.

Inherited from StreamGraph::Model::Node

See the documentation of StreamGraph::Model::Node for descriptions.

`$name` (String)
`$id` (String)
`$x` (Integer)
`$y` (Integer)

Inherited from StreamGraph::Model::Saveable

None.

Methods

```
StreamGraph::Model::Node::Filter->new(
                      initCode=>$initCode,
                      workCode=>$workCode,
                      globalVariables=>$globalVariables,
                      timesPush=>$timesPush,
                      timesPop=>$timesPop,
                      timesPeek=>$timesPeek,
                      joinType=>$joinType,
                      splitType=>$splitType,
                      inputType=>$inputType,
                      inputCount=>$inputCount,
                      outputType=>$outputType,
                      outputCount=>$outputCount)
```

Create a StreamGraph::Model::Node::Filter.

`get_parameters($graph, $parameterTypeFlag)`

`return` list[StreamGraphModel::Node::Parameter] or

list[StreamGraph::Model::CodeObject::Parameter] as specified.

If the `$parameterTypeFlag` is not given or true a

list[StreamGraphModel::CodeObject::Parameter] is returned. Otherwise a list[StreamGraphModel::Node::Parameter] is returned. The returned list has all parameters that are connected to the filter in the `$graph`.

`get_edge_data_to($target, $graph)`

`return` StreamGraph::Model::ConnectionData

Get the data attribute of the connection to the `$target`

`get_edge_data_from($source, $graph)`

`return` StreamGraph::Model::ConnectionData

Get the data attribute of the connection from the `$source`

`set_edge_attribute_to($target, $graph, $key, $value)`

Set the generic attribute for the edge to the `$target`.

`set_edge_attribute_from($source, $graph, $key, $value)`

Set the generic attribute for the edge from the `$source`.

`set_edge_data_to($target, $graph, $inMult, $outMult)`

Set the data for the edge to the `$target`. Requires input and output multiplicities for the connection.

`set_edge_data_from($source, $graph, $inMult, $outMult)`

Set the data for the edge from the `$source`. Requires input and output multiplicities for the connection.

Inherited from StreamGraph::Model::Node

```
isFilter()
isSubgraph()
isDataNode()
isParameter()
isComment()
is_split()
is_join()
resetId()
```

Inherited from StreamGraph::Model::Saveable

```
yaml_dump
```

StreamGraph::Model::Node::Parameter

The parameter in the StreamGraph view have this data structure.

Properties

`outputType` (String)
: The StreamIt type of the parameter.

`value` (Var)
: The value of the parameter.

`saveMembers` (list[String])
: The properties which are saved when the graph is saved to a file.

Inherited from StreamGraph::Model::Node

See the documentation of StreamGraph::Model::Node for descriptions.

`$name` (String)
`$id` (String)
`$x` (Integer)
`$y` (Integer)

Inherited from StreamGraph::Model::Saveable

None.

Methods

`StreamGraph::Model::Node::Parameter->new(outputType=>$outputType, value=>$value)`
: Create a StreamGraph::Model::Node::Parameter.

Inherited from StreamGraph::Model::Node

```
isFilter()
isSubgraph()
isDataNode()
isParameter()
isComment()
is_split()
is_join()
resetId()
```

Inherited from StreamGraph::Model::Saveable

```
yaml_dump
```

StreamGraph::Model::Node::Comment

The Implementation of comments in the StreamGraph view (a pretty uninteresting one if you ask me)

Properties

`string` (String)
: The comment text.

`saveMembers` (list[String])
: The properties which are saved when the graph is saved to a file.

Inherited from StreamGraph::Model::Node

See the documentation of StreamGraph::Model::Node for descriptions.

`$name` (String)
`$id` (String)
`$x` (Integer)
`$y` (Integer)

Inherited from StreamGraph::Model::Saveable

None.

Methods

`StreamGraph::Model::Node::Comment->new(string=`$string`)>`
: Create a StreamGraph::Model::Node::Comment.

Inherited from StreamGraph::Model::Node

`isFilter()`
`isSubgraph()`
`isDataNode()`
`isParameter()`
`isComment()`
`is_split()`
`is_join()`
`resetId()`

Inherited from StreamGraph::Model::Saveable

`yaml_dump`

StreamGraph::Model::Node::Subgraph

A node containing another graph.

Properties

`filepath` (String)
: The path to the file in which the subgraph is saved.

`unsaved` (Boolean)
: Flag which specifies if the graph was never saved.

`visible` (Boolean)
: Flag which specifies if the View with the subgraph is open.

`joinType` (String)
: The (StreamIt) description of the join. May be void or round robin.

`splitType` (String)
: The (StreamIt) description of the split. May be void, round robin or duplicate.

`inputType` (String)
: The StreamIt type of the input.

`inputCount` (Integer)
: The number of incomming data connections.

`outputType` (String)
: The StreamIt type of the output.

`outputCount` (Integer)
: The number of outgoing data connections.

`saveMembers` (list[String])
: The properties which are saved when the graph is saved to a file.

Inherited from StreamGraph::Model::Node

See the documentation of StreamGraph::Model::Node for descriptions.

`$name` (String)
`$id` (String)
`$x` (Integer)
`$y` (Integer)

Inherited from StreamGraph::Model::Saveable

None.

Methods

```
StreamGraph::Model::Node::Subgraph->new(
                  filepath=>$filepath
                  unsaved=>$unsaved
                  visible=>$visible
                  joinType=>$joinType
                  splitType=>$splitType
                  inputType=>$inputType
                  inputCount=>$inputCount
                  outputType=>$outputType
                  outputCount=>$outputCount
                  saveMembers=>$saveMembers)
```

Create a StreamGraph::Model::Node::Subgraph.

Inherited from StreamGraph::Model::Node

```
isFilter()
isSubgraph()
isDataNode()
isParameter()
isComment()
is_split()
is_join()
resetId()
```

Inherited from StreamGraph::Model::Saveable

```
yaml_dump
```

StreamGraph::Model::CodeObject

The StreamGraph::Model::CodeObject class is a wrapper class for all implemented topological constructs of the StreamIt language and Parameters.

Properties

`name` (String)
: The name of the topological construct.

`outputType` (String)
: The type of the output of the topological construct.

Methods

`StreamGraph::Model::CodeObject->new(name=>$name, outputType=>$outputType)`
: Create a StreamGraph::Model::CodoeObject. Do not use CodeObject is an abstract class.

StreamGraph::Model::CodeObject::SplitJoin

The StreamGraph::Model:CodeObject::SplitJoin class implements the split-joins of StreamIt.

Properties

`codeObjects` (list[StreamGraphModel::CodeObject])
: list[StreamGraphModel::CodeObject] which are directly nested in the split-join construct.

`split` (StreamGraph::Model::Node::Filter)
: The filter on which the splitting is occuring.

`join` (StreamGraph::Model::Node::Filter)
: The filter on which the splitted streams are converging.

`next` (StreamGraph::Model::Node::Filter)
: The filter which is the next to be processed after completely building the codeObject.

`code` (String)
: The StreamIt code for the codeObject.

`parameters` (list[StreamGraphModel::CodeObject::Parameter])
: The parameters of all filters nested in the codeObject.

`inputType` (String)
: The type(a type in StreamIt) in which the input of the codeObject is given.

`graph` (StreamGraph::GraphCompat)
: The graph in which the split-join is located.

Inherited from StreamGraph::Model::CodeObject

See the documentation of StreamGraph::Model::CodeObject for descriptions.

`name` (String)
`outputType` (String)

Methods

`StreamGraph::Model::CodeObject::SplitJoin->new(first=>$first, graph=>$graph)`
: Create a StreamGraph::Model::CodeObject::SplitJoin starting on `$first`(a StreamGraph::Model::Node::Filter) in the `$graph` (a StreamGraph::GraphCompat). While creating the split-join it is necessary to create all nested CodeObjects within it. Therefore these are also created which assures the StreamIt typical hierarchical structure, as well as the complete generation of all necessary codeObjects.

`getSplitCode($view)`
: `return` Code for the split or Error if a failure occured.

Generates the code for the split part of the split-join including all necessary multiplicities.

`getJoinCode($view)`

`return` Code for the join or Error if a failure occured.

Generates the code for the join part of the split-join including all necessary multiplicities.

`generate($view)`

`return` 1 if no error occured, otherwise Error

Generates the code for the split-join, as well as all nested CodeObjects and fills the code property.

`buildCode($pipelinesCode, $splitJoinesCode)`

`return` The complete code for the codeObject and all nested codeObjects as a tuple of code for pipelines and split-joines.

Gets all the code necessary for a StreamIt program out of the codeObject and all nested codeObjects.

Inherited from StreamGraph::Model::CodeObject

None.

StreamGraph::Model::CodeObject::Parameter

Implements parameters for usage in the code generation.

Properties

`associatedParameter` (StreamGraph::Model::Node::Parameter)
: The node in the graph which represents the parameter.

`value` (Var)
: The value of the parameter.

Inherited from StreamGraph::Model::CodeObject

See the documentation of StreamGraph::Model::CodeObject for descriptions.

`name` (String)
`outputType` (String)

Methods

`StreamGraph::Model::CodeObject::Parameter->new(first=>$node)`
: Create a StreamGraph::Model::CodeObject::Parameter. Expects a StreamGraph::Model::Node::Filter as `$node`.

`updateValues()`
: Updates the values of the codeObject with those of the associated parameter.

Inherited from StreamGraph::Model::CodeObject

None.

StreamGraph::Model::CodeObject::Pipeline

The StreamGraph::Model::CodeObject::Pipeline class implements StreamIt pipeplines

Properties

`codeObjects` (list[StreamGraph::Model::CodeObject, StreamGraph::Model::Node::Filter])
: The objects which are in the pipeline in the order in which a packet of data would be processed by them. They may be a StreamGraph::Model::Node::Filter or a StreamGraph::Model::CodeObject (most of the time the StreamGraph::Model::CodeObject will be a StreamGraph::Model::CodeObject::SplitJoin). Since StreamGraph::Model::Node::Filter already has all relevant infromation it was not necessary to create a filter as a codeObject, as a result of that this list has items of both StreamGraph::Model::Node::Filter and StreamGraph::Model::CodeObject.

`next` (StreamGraph::Model::Node::Filter)
: The filter which is the next to be processed after completely building the codeObject.

`code` (String)
: StreamIt code for the pipeline.

`parameters` (list[StreamGraphModel::CodeObject::Parameter])
: The parameters of all filters nested in the codeObject.

`inputType` (String)
: The type(a type in StreamIt) in which the input of the codeObject is given.

`graph` (StreamGraph::GraphCompat)
: The graph in which the pipeline is located.

Inherited from StreamGraph::Model::CodeObject

See the documentation of StreamGraph::Model::CodeObject for descriptions.

`name` (String)
`outputType` (String)

Methods

`StreamGraph::Model::CodeObject::Pipeline->new(first=>$first, graph=>$graph)`
: Create a StreamGraph::Model::CodeObject::Pipeline starting on `$first` (a StreamGraph::Model::Node::Filter) in the `$graph` (a StramGraph::GraphCompat). While creating a pipeline a node with multiple outputs may be detected. In that case it is necessary to start the generation of a split-join-construct. Since all directly nested constructs are needed for the later generation these nested constructs will also be created. This assures the StreamIt typical hierarchical structure, as well as the complete generation of all necessary codeObjects.

`generate($view, $mainFlag)`

> `return` 1 if no error occured, otherwise Error
>
> Generates the code for the pipeline, as well as all nested CodeObjects and fills the code property. If `$mainFlag` is set to true the generated pipeline will be treated as the main pipeline, which means, that its name will be the name of the project. This property will not be carried on to nested codeObjects, so that there exists only one main pipeline in every project.

`buildCode($pipelinesCode, $splitJoinesCode)`

> `return` The complete code for the codeObject and all nested codeObjects as a tuple of code for pipelines and split-joines.
>
> Gets all the code necessary for a StreamIt program out of the codeObject and all nested codeObjects.

Inherited from StreamGraph::Model::CodeObject

None.

StreamGraph::Model::Namespace

Namespaces are used by StreamGraph::GraphCompat to separate potentially identically named parameters in subgraphs from those passed in from the parent graph. Instances of this module let you `register` names and then, given a graph, will replace all references to this parameter in all filters inside that graph.

Properties

`$filepath` (String)
: The subgraph filename, used to uniquely identify and prefix a namespace.

`$names` (list[String])
: The list of parameters to replace. This list is used internally and while you could set this in the constructor, it's probably more useful to use `register($name)` for adding parameters.

Methods

`StreamGraph::Model::Namespace->new($filepath=>$filepath)`
: Create a StreamGraph::Model::Namespace.

`register($name)`
: Registers one string parameter `$name` into the namespace.

`newname($name)`
: `return` a string with the namespace-prefixed name.

  The namespace-prefixed string will have the format `basename($filepath)."_".$name`.

`replace($graph, $name)`
: Replaces one `$name` inside the given `$graph`. The name doesn't have to be registered and it won't be after this method is called. Most likely `replaceAll()` is what you want.

`replaceAll($graph)`
: Replaces all occurrences of all registered parameter names in all filters of the `$graph`.

StreamGraph::Model::ConnectionData

The StreamGraph::Model::ConnectionData is the implementation of StreamIt's multiplicities for split-join connections.

Properties

`inputMult` (Integer)
: The number of packages inputted from the split in one cycle.

`outputMult` (Integer)
: The number of packages inputted into the join in one cycle.

`saveMembers` (list[String])
: The properties which are saved, when a graph is saved to a file.

Inherited from StreamGraph::Model::Saveable

None.

Methods

`StreamGraph::Model::ConnectionData->new(inputMult=>$inputMult, outputMult=>$outputMult)`
: Create a StreamGraph::Model::ConnectionData.

`createCopy()`
: **`return`** A new StreamGraph::Model::ConnectionData with the exact same entries.

  Copy the entries of the StreamGraph::Model::ConnectionData to a new StreamGraph::Model::ConnectionData.

Inherited from StreamGraph::Model::Saveable

`yaml_dump`

StreamGraph::GraphCompat

The StreamGraph::GraphCompat class assures that a graph is StreamIt compatible. It inherits from the StreamGraph::View::Graph class.

Properties

None.

Inherited from StreamGraph::View::Graph

`$graph`

`$root`

Methods

`StreamGraph::GraphCompat->new($graph)`
: Create a StreamGraph::GraphCompat from the given StreamGraph::View::Graph. This is essentially a deep copy of the given `$graph` with its StreamGraph::View::Items unpacked into plain StreamGraph::Model::Nodes.

`graph()`
: `return` graph (Graph::Directed)

`source()`
: `return` source (StreamGraph::Model::Node::Filter)

`sink()`
: `return` sink (StreamGraph::Model::Node::Filter)

`factory()`
: `return` factory (StreamGraph::Model::NodeFactory)

`success()`
: `return` success (Boolean)

`_addIdentities()`
: Adds identities throughout the graph for all connections whose two nodes have other paths between them as well.

`_addVoidSource()`
: `return` the graph's source (StreamGraph::Model::Node::Filter) or 0 on error.

  If there are multiple sources in the graph, adds a new filter with a void- typed input joining all sources.

  If there is only one source, then return this source filter unchanged.

`_addVoidSink()`
: `return` the graph's sink (StreamGraph::Model::Node::Filter) or 0 on error.

  If there are multiple sinks in the graph, adds a new filter with a void-typed output joining all sinks.

  If there is only one sink, then return this sink filter unchanged.

`_addVoidEnd($type, $getEndNodes, $getIOType)`

`return` the created end node(StreamGraph::Model::Node::Filter)

Takes a `$type`(String), either `"source` or `"sink"` and two functions, `$getEndNodes` to filter the relevant end nodes, and `$getIOType` to get the type of the relevant pins. It then checks whether there are multiple sources or sinks and if there are creates a void end, connects it and returns it.

`_subgraphs()`

Flatten all subgraph nodes (and their children recursively) into the parent graph structure. See `_subgraph()` for details.

`_subgraph($n)`

Flatten a subgraph node `$n` (StreamGraph::Model::Node) into the parent graph structure.

The subgraph is loaded, parsed and checked for errors. If there are errors in the subgraph, it is skipped and this parent graph is also marked as failed.

The loaded subgraph's nodes and connections are inserted into the parent graph and connected inline.

`_copyData($graph)`

Copies nodes, connections and connection-attributes from the constructor argument `$graph` to this $graph.

`_loadFile($filepath)`

`return` a graph (StreamGraph::GraphCompat) loaded from `$filepath`.

Static method that loads and parses a file into a temporary StreamGraph::GraphCompat instance and namespaces all it's parameters (See StreamGraph::Model::Namespace for details).

Inherited from StreamGraph::View::Graph

```
add_vertex()
add_edge()
get_edge_attribute()
set_edge_attribute()
get_root()
has_item()
get_items()
get_connections()
num_items()
predecessors()
remove_vertex()
remove_edge()
successors()
all_successors()
topological_sort()
all_non_predecessors()
is_predecessor()
is_successor()
successorless_filters()
predecessorless_filters()
set_root()
traverse_postorder_edge()
traverse_preorder_edge()
same()

sameErr()
connected()
circle()

typeCompatible()
directlyConnected()
nextSplitJoin()
crossConnection()
lca()

connectable()
connectableErr()
connectableQuiet()
```

StreamGraph::CodeGen

Wrapper file for the code generation into StreamIt

Functions

`generateCode($view, $graph, $configFile, $fileName)`

> `return` Generated code or Error message if the generation failed.
>
> This is a wrapper function for the code generation which gets the $view to print error messages, the `$graph` for which the code should be generated, the $configFile to write into the tmp directory and optionally the `$fileName`. If the $fileName is not given the name main is assumed.

`generateCommentary($commentText)`

> `return` comment text
>
> Generates a comment in StreamIt code with the given `$commentText`.

`generateMultiLineCommentary($commentaryText)`

> `return` comment text
>
> Generates a multiline commentary in StreamIt code with the given (multiline) `$commentText`.

`generateSectionCommentary($commentText)`

> `return` comment text
>
> Generates a Section heading as a StreamIt commentary with the given `$commentText`.

`generateWork($data)`

> `return` code for the work function of the given filter(`$data`)
>
> Expects a StreamGraph::Model::Node::Filter as input. Generates the code for the work function of the given filter.

`generateInit($data)`

> `return` code for the init function of a filter
>
> Expects a StreamGraph::Model::Node::Filter as input. Generates the code for the init function of a filter.

`generateParameters($parameterListPointer, $typeFlag, $bracketFlag, $valueFlag, $listFlag)`

> `return` string or list of parameter names, types and values as specified through parameters
>
> Expects a pointer to a list[StreamGraphModel::Node::Parameter] as first parameter. The `$listFlag` specifies if the returned value should be a list if the flag is true or a complete string. If the `$typeFlag` is true then the type of the parameters in the list of parameters(`$parameterListPointer`) is included in the returned value. The `$bracketFlag` enables brackets in the begin and the end of the returned string when the `$listFlag` is false, otherwise it is irrelevant. The `$valueFlag` if true adds the value of the parameters of the list(`$parameterListPointer`) to the returned value.

`generateFilter($filterNode, $graph)`

`return` code of the filter in StreamIt or error message.

Expects a StreamGraph::Model::Node::Filter and a StreamGraph::GraphCompat as input parameters. Generates the complete code of a filter including it's parameters, the init-function, the work-function and it's filter-global variables.

`updateNodeName($filterNode)`

`return` Error or nothing

Expects a StreamGraph::Model::Node::Filter as input. Updates the name of the Node to be unique through adding a '_gen_name' field with the updated name.

`getTopologicalConstructName($mainFlag, $splitJoinText)`

`return` name of the topological construct.

Generates the name of a construct. If `$mainFlag` is true returns name of the file, since StreamIt expects a construct with the name of the file as the main construct. The `$splitJoinText` is optional, but the function assumes that without an input of it (`$splitJoinText`) the construct for which the name should be generated is a pipeline. Otherwise the function assumes it is a split-join.

StreamGraph::CodeRunner

Compile and run the generated StreamIt code.

All results are written to files, including status codes etc., because the compilation and execution is run asynchronously.

Properties

`config` (StreamGraph::Util::Config)
: The StreamGraph config file.

`source` (String)
: The source code file.

`ccPid` (Integer)
: The compiler process' PID.

`ccResult` (list[String])
: The compiler output, line by line.

`ccResultFile` (String)
: The compiler log filename.

`ccSuccess` (Integer)
: The compiler process' exit code.

`ccSuccessFile` (String)
: The filename for storing the compiler process' exit code.

`binary` (String)
: The binary file.

`rPid` (Integer)
: The runner process' PID.

`rResult` (list[String])
: The runner output, line by line.

`rResultFile` (String)
: The runner log filename.

`rSuccess` (Integer)
: The runner process' exit code.

`rSuccessFile` (String)
: The filename for storing the runner process' exit code.

Methods

`StreamGraph::CodeRunner->new(config=`$config)`>`
: Create a StreamGraph::CodeRunner. The `config` parameter is required.

`setStreamitEnv()`
: Set the environment from the StreamGraph config.

`compileAndRun($filename, $callback)`
: Compile the StreamIt code and then execute it. When finished run `$callback`. Returns immediately.

`compile($filename, $callback)`
: Compile the StreamIt code. When finished run `$callback`. Returns immediately.

`run($callback)`
: Run the StreamIt binary produced by the compiler. When finished run `$callback`. Returns immediately.

`isCompiling()`
: `return` Boolean

  Check if compiler is still running.

`isRunning()`
: `return` Boolean

  Check if runner is still running.

`compileResult($lines)`
: `return` the result of the compiler (list[String])

  If `$lines` (Integer) is given, return only the first `$lines` result lines.

`compileSuccess()`
: `return` the compiler exit code (Integer)

`compileErrors()`
: `return` errors in the compiler output (list[String])

  Returns only the lines from the output that are actual error locations and not the java stacktrace.

`runResult($lines)`
: `return` the result of the runner (list[String])

  If `$lines` (Integer) is given, return only the first `$lines` result lines.

`runSuccess()`
: `return` the runner exit code (Integer)

`_compile()`
: Fork and execute the compilation process.

`_updateCC()`
: Update all compiler relevant fields (`$self->cc*`).

`_run()`
: Fork and execute the runner process.

`_updateRun()`
: Update all runner relevant fields (`$self->r*`).

`_getResult($result, $lines)`
: `return` the first `$lines` (Integer) lines from the `$result` (list[String]).

`_after($pid, $callback)`
: Wait for the forked process with `$pid` (Integer) and then call `$callback` (Function).

`_killIfNecessary($process)`
: Kill the process if the PID returned by `$process-`($self)`>` exists.

  `$process` has to be a code ref to either `ccPid()` or `rPid()`.

StreamGraph::View

The StreamGraphView draws a graph on a Gnome2::Canvas.

The StreamGraphView is an extension of the Gnome2::Canvas which is a Gtk2::Widget, so it can be placed in any Gtk2 container.

Properties

'aa' (boolean : readable / writable /construct-only)
: The antialiasing mode of the canvas.

'connection_colors_gdk' (Gtk2::Gdk::Color : readable / writable)
: The default colors to apply to connection objects of various types as they are created.

'connection_arrows' (string : readable / writable);
: The type of arrow to use when creating a connection object. May be one of: 'none', 'one-way', or 'two-way'.

Methods

`new(aa=>1)`
: Construct an anti-aliased canvas. Aliased canvases look just awful.

`INIT_INSTANCE`
: This subroutine is called by Glib::Object::Subclass as the object is being instantiated. You should not call this subroutine directly. Leave it alone.

`SET_PROPERTY`
: This subroutine is called by Glib::Object::Subclass to set a property value. You should not call this subroutine directly. Leave it alone.

`add_item ($item)`
: Add the root item to the stream graph. This is the node off of which all other nodes are attached. The item must be a StreamGraph::View::Item.

`add_item ($predecessor_item, $item)`
: Add an item to the stream graph. The item is linked to its predecessor item. The item must be a StreamGraph::View::Item.

`clear()`
: Clear the items from the stream graph.

`layout()`
: Layout the stream graph. The map is redrawn on the canvas.

`predecessors ($item)`
: Returns an array of StreamGraph::View::Items that are the items that link to the item argument you have specified. Each item in the stream graph may have zero or more predecessors.

`remove_item ($item)`
: Remove the item from the graph including all connections.

`remove_connection($connection)`
: Remove the connection from the graph.

`set_root ($item)`
: Change the root StreamGraph::View::Item in the underlying graph, and revise the visible connections in the stream graph.

`println($str,$type)`
: This methid prints a String on the terminal and if type is defined also shows a image. Type is a string and has a format from Gtk2::Stock without gtk-.

`connect($predecessor_item, $item, $connection_data)`
: This method creates a new connection between two items.

`deselect()`
: This method deselects all items.

StreamGraph::View::Graph

This is internal to StreamGraph::View. It's a wrapper around Jarkko Heitaniemi's nice Graph module. This module is instantiated by StreamGraph::View.

`StreamGraph::View::Graph->new()`
: Create a StreamGraph::View::Graph.

`add_vertex($item)`
: Add a vertex to the graph. Only one of these may be added, or you will get an error.

`add_edge($predecessor_item, $item)`
: Add an edge between `$predecessor_item` and `$item` to the graph. If one or both do not exist in the graph they will be added.

`get_root()`
: Return the root item of the graph.

`has_item($item)`
: Return true if the graph contains the item.

`num_items($item)`
: Return the number of items in the graph.

`predecessors($item)`
: Return the predecessor items of a given StreamGraph::View::Item.

`remove_vertex($item)`
: Remove a StreamGraph::View::Item from the graph. Remove any connections that the item may have had.

`set_root($item)`
: Change the root item in the graph. An new graph is created with the new root.

`successors($item)`
: Return the successor items of a given StreamGraph::View::Item.

`traverse_preorder_edge($predecessor_item, $item, $callback)`
: Perform a depth first traversal and pass back the predecessor item as well as the item to the callback. Need to do something about all these traversal routines.

`traverse_postorder_edge($predecessor_item, $item, $callback)`
: Perform a depth first traversal and pass back the predecessor item as well as the item to the callback. Need to do something about all these traversal routines.

`get_edge_attribute($source, $target, $key)`
: `return` Var

  Returns the edge attribute of the edge from `$source` to `$target` with the `$key`.

`set_edge_attribute($source, $target, $key, $value)`
: Set the edge attribute of the edge from `$source` to `$target` with the `$key` to `$value`.

`get_items()`
: `return` list[StreamGraph::View::Item]

  Returns all items of the graph in non particular order.

`get_connections()`
: `return` list[tuple[StreamGraph::View::Item, StreamGraph::View::Item]]

  Returns all connections of the graph.

`remove_vertex($item)`
: Removes the `$item` and all connections to and from it.

`remove_edge($source, $target)`
: Removes the edge between `$source` and `$target`.

`all_successors($item)`
: `return` list[StreamGraph::View::Item]

  Returns successors of the `$item` and all of their successors.

`topological_sort()`
: `return` list[StreamGraph::View::Item]

  Returns all items of the graph sorted in topological order.

`all_non_predecessors($item)`
: `return` list[StreamGraph::View::Item]

  Returns all nodes which are not predecessors. Includes unconnected nodes.

`is_predecessor($item, $potential_predecessor)`
: `return` Boolean

  Checks if `$potential_predecessor` is a predecessor of `$item`. Returns true if it is.

`is_successor($item, $potential_successor)`
: `return` Boolean

  Checks if `$potential_successor` is a successor of `$item`. Returns true if it is.

`successorless_filters()`
: `return` list[Var]

  Returns all filters with no successor.

`predecessorless_filters()`
: `return` list[Var]

  Returns all filters with no predecessors. Parameters as predecessors are allowed.

`same($node1, $node2)`
: `return` Boolean

  Checks if `$node1` and `$node2` are the same.

`sameErr($node1, $node2)`
: `return` tuple[Boolean, String]

  Helper for same. Checks if `node1` and `$node2` are the same. Returns true if they are. false otherwise, as well as an error message.

`connected($node1, $node2)`

  `return` tuple[Boolean, String]

  Checks if `$node1` and `$node2` are already connected. Returns boolean and error message as a tuple.

`circle($node1, $node2)`

  `return` tuple[Boolean, String]

  Checks if a connection from `$node1` to `$node2` would result in a circle in the graph. Returns boolean and error message.

`typeCompatible($node1, $node2, $direction)`

  `return` tuple[Boolean, String]

  Checks if the types of the connection in the `$direction` between `$node1` and `$node2` are matching. That means it checks if the output type from the source equals the input type of the target.

`directlyConnected($node1, $node2, $direction)`

  `return` Boolean

  Checks if `$node1` and `$node2` are successors of one another in the `$direction`.

`nextSplitJoin($node, $direction)`

  `return` Var

  Tries to find the split for the join (`$node`) if `$direction` equal `"up"`. othwerwise tries to find the join for the split (`$node`). Returns the Split or join respectivly

`crossConnection($node1, $node2, $direction)`

  `return` tuple[Boolean, String]

  Given two nodes, return true iff there exists any path between a successor of the would-be predecessor node and a predecessor of the would-be successor node.

`lca($node1, $node2, $direction)`

  `return` Var

  Determine the "lowest common ancestor" between two nodes. The LCA is the first node in the specified direction that is a predecessor/successor of the two and returns it. It's possible that one of the given nodes is the LCA itself (if there is a straight path between them). If no LCA can be determined (unconnected in the specified direction) 'undef' will be returned.

`connectable($node1, $node2, $direction)`

  `return` tuple[Boolean, String]

  Checks if `$node1` and `$node2` can be connected in the `$direction`.

`connectableErr($node1, $node2, $direction)`

  `return` Boolean

  Wrapper for connectable. Prints connectables error message if connectable returned false. Returns boolean given by connectable.

`connectableQuiet($node1, $node2, $direction)`

  `return` Boolean

  Wrapper for connectable. Does not print connectables error message. Returns boolean given by connectable.

StreamGraph::View::Item

StreamGraph::View::Item items contain the border and content that is displayed in the mind map. They may be created using StreamGraph::View::ItemFactory and may be placed into StreamGraph::View.

Properties

`graph` (StreamGraph::View::Graph)
: A reference to the StreamGraph::View::Graph that contains all the StreamGraph::View::Item items.

`border` (StreamGraph::View::Border)
: A reference to the StreamGraph::View::Border that is drawn on the canvas. The border contains a reference to the content.

`visible` (boolean)
: A flag indicating whether or not this item is visible.

`x` (double)
: The upper left x-coordinate of the item.

`y` (double)
: The upper left y-coordinate of the item.

`height` (double)
: The height of the item.

`width` (double)
: The width of the item.

Methods

`INIT_INSTANCE`
: This subroutine is called by Glib::Object::Subclass whenever a StreamGraph::View::Item is instantiated. It initialized the internal variables used by this object. This subroutine should not be called by you. Leave it alone.

`SET_PROPERTY`
: This subroutine is called by Glib::Object::Subclass whenever a property value is being set. Property values may be set using the `set()` method. For example, to set the width of an item to 100 pixels you would call set as follows: `$item->set(width=>100);`

`add_hotspot ($hotspot_type, $hotspot)`
: Add a StreamGraph::View::ItemHotSpot to an item. There are two types of hotspots ('toggle_top','toggle_down').

  The `"toggle"` hotspots correspond to the small circles you see on a view item that allow for connecting two items.

  You should add a hotspot for each hotspot type to an item. If you use the StreamGraph::View::ItemFactory to create items, this will be done for you.

When you add a hotspot the hotspot type is used to position the hotspot on the item. You may only add one hotspot of each type.

`disable_hotspots ()`
: This method is used to disable and hide the "toggle" hotspots, which only appear on an item if they are needed.

`enable_hotspots ($successor_item)`
: This method enables and shows the "toggle" hotspots provided that they are needed by the item. In item needs a toggle hotspot if it has successor items attached to it.

`get_column_no`
: Return the column number that this item belongs to. The column number is used to determine the relative position of items in the layout.

`get_connection_point ($side,$connection)`
: Return the x,y coordinates of the point at which a StreamGraph::View::Connection may connect to. This coordinate is also used to detemine where to place the "toggle" hotspots.

`get_insets ()`
: Return the `($top, $left, $down, $right)` border insets. The insets are used by the Grips and Toggles to position themselves.

`get_min_height()`
: Return the minimum height of this item.

`get_min_width()`
: Return the minimum width of this item.

`get_weight ()`
: Return the "weight" of a view item. The weight is the product of the item height and width. The weight is used by StreamGraph::View::Layout::Balanced to determine the side of the mind map on which to place the item.

`is_visible ()`
: Return true if this item is visible.

`predecessors ()`
: Return an array of predecessor items of this item.

`resize ()`
: Adjust the height and width of the item, and then signal to the toggles, grips and connections to redraw themselves.

`successors ()`
: Return an array of successor items of this item.

`successors ($side)`
: Return an array of items that are on one side of this item. The side may be 'top' or 'down'.

`update ()`
: Sets height and width based on the content size.

`add_connection($self, $side, $connection)`
: This method adds a connection to the list of top or down connections. Evry items has complete list of all top or down connections.

`remove_connection($self, $side, $connection)`
: This method removes a connection from the list of top or down connections.

`set_data($data)`
: This method sets the data node.

`set_view($view)`
: This method sets the $view. This is necessary, because items can access view functions.

`select($switch)`
: This method changes the background color of the item based on the switch value (0 -> white or 1 -> blue).

`get_edge_data_to($target)`
: `return` StreamGraph::Model::ConnectionData

  Get the data attribute of the connection to the `$target`

`get_edge_data_from($source)`
: `return` StreamGraph::Model::ConnectionData

  Get the data attribute of the connection from the `$source`

`set_edge_attribute_to($target, $key, $value)`
: Set the generic attribute with `$key` and `$value` for the edge to the `$target`.

`set_edge_attribute_from($source, $key, $value)`
: Set the generic attribute with `$key` and `$value` for the edge from the `$source`.

`set_edge_data_to($target, $graph, $inMult, $outMult)`
: Set the data for the edge to the `$target`. Requires input and output multiplicities for the connection.

`set_edge_data_from($source, $graph, $inMult, $outMult)`
: Set the data for the edge from the `$source`. Requires input and output multiplicities for the connection.

`get_parameters()`
: `return` list[StreamGraphModel::Node::Parameter] or list[StreamGraph::Model::CodeObject::Para as specified.

  If the `$parameterTypeFlag` is not given or true a list[StreamGraphModel::CodeObject::Parameter] is returned. Otherwise a list[StreamGraphModel::Node::Parameter] is returned. The returned list has all parameters that are connected to the filter in the `$graph`.

`connections($direction)`
: `return` list[StreamGraph::View::Item]

  Returns predecessors if `$direction` equals `"up"`, successors otherwise.

`all_successors()`

: `return` list[StreamGraph::View::Item]

  returns successors and all their successors.

`all_predecessors()`

: `return` list[StreamGraph::View::Item]

  returns predecessors and all their predecessors.

`is_split()`

: `return` Boolean

  Checks if item has more than one predecessor that is not a parameter.

`is_join()`

: `return` Boolean

  Checks if the item has more than one successor.

`isFilter()`

: `return` Boolean

  Checks if the items data field is of the StreamGraph::Model::Node::Filter class.

`isSubgraph()`

: `return` Boolean

  Checks if the items data field is of the StreamGraph::Model::Node::Subgraph class.

`isDataNode()`

: `return` Boolean

  Checks if the items data field is either of the StreamGraph::Model::Node::Filter class or of the StreamGraph::Model::Node::Subgraph class.

`isParameter()`

: `return` Boolean

  Checks if the items data field is of the StreamGraph::Model::Node::Parameter class.

`isComment()`

: `return` Boolean

  Checks if the items data field is of the StreamGraph::Model::Node::Comment class.

StreamGraph::View::HotSpot

This module is internal to StreamGraph::View. Four StreamGraph::View::HotSpots are created for each StreamGraph::View::Item. The hotspots are areas on a mind map item that when clicked, cause an action to be performed on an item. These hotspots allow the user to expand/collapse the items in the mind map, or to resize an item.

Properties

Use the `set` method to set these properties. Accessing them directly will only cause you trouble.

'item' (StreamGraph::View::Item)
: Items and hotspots are rather fond of each other. This item is the one this hotspot is attached to.

'enabled' (boolean)
: If enabled, this hotspot is ready for action. The type of action depends on whether it is a grip or a toggle. Grips are used to resize an item. Toggles are used to expand or collapse paths on the mind map graph.

'fill_color_gdk' (Gtk2::Gdk::Color)
: The color with which to fill in the hotspot.

'outline_color_gdk' (Gtk2::Gdk::Color)
: The color with which to fill in the hotspot outline. Toggles normally have a visible outline, while grips usually have the outline set to the same color as the item fill color.

'hotspot_color_gdk' (Gtk2::Gdk::Color)
: The color of the hotspot once it is engaged. A hotspot becomes engaged when the mouse is placed close to it.

Methods

`new (item=>$item)`
: Instantiates a hotspot that is associated with the StreamGraph::View::Item.

  This module connects to the Gnome2::Canvas::Item "event" event, and depending on the event type will call back to it's StreamGraph::View::Item.

`hotspot_adjust_event_handler`
: This method must be overridden. It handles the "hotspot_adjust" event.

`hotspot_button_press`
: This method may optionally be overridden to handle the "button-press" event.

`hotspot_button_release`
: This method may optionally be overridden to handle the "button-release" event.

`hotspot_engaged`
: This method may optionally be overridden to set the "engaged" flag in a non-standard way.

`hotspot_enter_notify`
: This method may optionally be overridden to handle the "enter-notify" event.

`hotspot_get_image()`
: This method must be overridden. It is used to instantiate a hotspot toggle or grip.

`hotspot_leave_notify`
: This method may optionally be overridden to handle the "leave-notify" event.

`hotspot_motion_notify`
: This method may optionally be overridden to handle the "motion-notify" event.

StreamGraph::Util

Package for various low level utility functions.

Functions

`getNodeWithId($nodes, $id)`
: `return` a node ($treamGraph::Model::Node) with the given `$id` or `undef`.

`getItemWithId($items, $id)`
: `return` an item (StreamGraph::View::Item) with the given `$id` or `undef`.

`unique($list)`
: `return` unique items in `$list` (list[Var]).

`filterNodesForType($listPointer, $type)`
: `return` nodes in `$listPointer` (list[StreamGraph::Model::Node]) having type `$type` (String).

StreamGraph::Util::Config

A utility class that implements the config file and its usage. The configuration file is needed for temporary dumps of the program, as well as and perhaps more importantly for the execution of the generated StreamIt program.

Properties

`configFile` (String)
: The filename of the configuration file.

`config` (Hash)
: The hash with the necessary information, like directories.

`default` (Hash)
: The default entries for the config hash if no configuration file exists.

Methods

`StreamGraph::Util::Config->new(configFile=>$configFile, config=>$config, default=>$default)`
: Create a StreamGraph::Util::Config. Set the config file location or use without parameters to use the default location of `"streamit.conf"`, i.e. in the working directory.

`load()`
: Loads the configuration file specified in the `configFile` field into the `config` field.

`write()`
: Writes the configuration file specified in `config` to the file specified in `configFile`.

`get($key)`
: `return` the value for a given `$key`.

`createDefault()`
: Sets the default config values and writes them. It is used if the given config file does not exist.

StreamGraph::Util::File

File serialization for StreamGraph. Write and read operations for the .sigraph YAML file format and generic file I/O.

Functions

`writeFile($string, $filename)`
: Writes `$string` to `$filename`.

`readFile($filename)`
: `return` contents of `$filename` (String).

`readFileAsList($filename)`
: `return` contents of `$filename` (list[String]) line by line.

`writeStreamitSource($writeString, $filename)`
: Writes `$writeString` to `$filename`, appending ".str" suffix if `$filename` doesn't already have it.

  Defaults to `"a.str"` if `$filename` is not given.

`save($filename, $graph, $window)`
: Serializes and saves a StreamGraph `$graph` and its `$window`'s properties to `$filename`.

`load($filename)`
: `return` a tuple `(%windowdata, @nodes, @connections)` representing the graph.

  Loads a StreamGraph `.sigraph` file into a list of nodes, a list of connections and some window properties that can be easily added to a graph.

  It detects the file format version and dispatches to a version-specific `load_*()` (see below).

`load_v0_2($obj)`
: `return` *same as* `load()`

  Loads the .sigraph YAML file format version 0.2.

`load_v0_1($obj)`
: `return` *same as* `load()`

  Loads the .sigraph YAML file format version 0.1 and older.

`_write($writeString, $filename)`
: Write a string to a file.

`_read($filename)`
: `return` a list of lines (list[String]) from a file.

StreamGraph::Util::PropertyWindow

Property Window functions for graph nodes.

Functions

`show($item,$window)`
: Creates a dialog window, with `$window` as the parent window. Based on `$item`'s type a custom function is called.

`show_connection($con)`
: Creates a dialog window and customizes it for the properties of the connection `$con`.

`show_comment($item, $dialog)`
: Customizes the dialog window `$dialog` for the properties of the comment `$item`.

`show_parameter($item, $dialog)`
: Customizes the dialog window `$dialog` for the properties of the parameter `$item`.

`show_filter($item, $dialog)`
: Customizes the dialog window `$dialog` for the properties of the filter `$item`.

`show_subgraph($item, $dialog)`
: Customizes the dialog window `$dialog` for the properties of the subgraph `$item`.

StreamGraph::Util::ResultWindow

This module shows the results of compiling and executing the generated StreamIt code in a dialog window.

Functions

`show_result($window,$view,$result)`
: Takes a String with the StreamIt run result and shows it in a dialog window.

StreamGraph::Util::DebugGraph

This module creates a GraphViz image of a graph and shows it in a dialog window.

Functions

`export_graph($window, $view,$graph, $dir)`
: Creates a GraphViz image of a graph and shows it in a dialog window.

`name_id($data)`
: Creates a unique name from `$data->name` and `$data->id`.

StreamGraph::Util::DebugCode

This module shows the StreamIt code in a dialog window.

Functions

`show_code($window,$view,$graph,$dir)`
: Gets a String with the StreamIt code and shows it in a dialog window.