UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

# The ffgraph Library

Carsten Friedrich

Lehrstuhl für Informatik
Universität Passau
Innstr. 33
D-94032 Passau (Germany)
tel: +49 851 509.3090
fax: +49 851 509.3092
friedric@fmi.uni-passau.de

MIP-9520
Dezember 1995

# Contents


**1 Introduction** **4**

**2 Installing the library** **5**

**3 Using the graph library** **6**

**4 Compiling your program** **7**
- 4.1 Using graphs only . . . 7
- 4.2 Using graphs and graphic . . . 8
- 4.3 Using graphs, graphic and layouters . . . 8

**5 Labels** **8**
- 5.1 Using labels . . . 8
- 5.2 Writing labels . . . 11

**6 Inspectors** **15**
- 6.1 Using inspectors . . . 15
- 6.2 Writing inspectors . . . 15

**7 Graphic contexts** **16**
- 7.1 Using graphic contexts . . . 19
- 7.2 Writing graphic contexts . . . 19

**8 Layouters** **19**
- 8.1 Using layouters . . . 20
- 8.2 Writing layouters . . . 20

**9 The important classes in detail** **21**
- 9.1 Standard classes . . . 21
  - 9.1.1 FFGraph . . . 21
  - 9.1.2 FFGraphClass . . . 21
  - 9.1.3 FFNodeClass . . . 26
  - 9.1.4 FFEdgeClass . . . 28
- 9.2 Label classes . . . 30
  - 9.2.1 FFLabelClass . . . 30
  - 9.2.2 FFGraphLabel . . . 30
  - 9.2.3 FFNodeLabel . . . 31
  - 9.2.4 FFEdgeLabel . . . 31
  - 9.2.5 FFSubGraphLabel . . . 32
  - 9.2.6 FFLayoutLabel . . . 32
  - 9.2.7 FFSugi . . . 32
  - 9.2.8 FFSpring . . . 34
- 9.3 Graphic classes . . . 36
  - 9.3.1 FFWindow . . . 36
  - 9.3.2 FFGraphWindow . . . 36
  - 9.3.3 FFGraphicNode . . . 40
  - 9.3.4 FFGraphicEdge . . . 43
  - 9.3.5 FFInspector . . . 45
  - 9.3.6 FFTclInspector . . . 45
  - 9.3.7 FFSugiInsp . . . 45
  - 9.3.8 FFSpringInsp . . . 47
  - 9.3.9 FFTclBasicWindow . . . 48

9.3.10 FFTclWindow . . . . . . 51
9.3.11 FFTclSubWindow . . . . . . 52
9.3.12 FFPoint . . . . . . 53
9.3.13 FFMatrix . . . . . . 53

**10 The ffgraph graph editor 54**

**A The extended sgraph file format 54**

**B The class hierarchy 57**

# 1 Introduction

ffgraph is a C++ class library to create, manipulate and display directed graphs. It was developed in the context of the META-Frame[SACTM95] project to visualize the various data structures used in this project.
There already exist many graph-tools, so the question arises why to write a new one instead of using an existing one. Many of the existing ones are programs like GraphEd[Him90] or the VCG tool[San94] so you can use them to display your data, but you have to implement a graph data structure in your program anyway. If you use external programs to display your graphs it is often impossible to dynamically change the graph in your program and update the display, or to manipulate it graphically and pass the update back to your program. On the other hand existing graph libraries, like LEDA[Näh94] offer a powerful graph data structure, but have less powerful means for displaying the data. Other tools are specialised on a small subset of graphs, like dataflow graphs etc., and are thus not usable for us. So we decided to write a graph library on our own, which offers you a graph data structure and powerful means to display it.
Graphs consist of a set of nodes and a set of edges, where edges have exactly one target and one source. Access to the library is provided through the classes described in this paper. Graphs can be loaded and saved in a slightly extended sgraph[Him93] file format[1]. The main purpose of this library is not speed or the ability to handle huge graphs, but to give an easy to use and save interface, the possibility to bind user specific data to graphs, nodes, and edges and to provide a powerful and flexible mechanism to display graphs.
The library allows the user to bind semantic labels to graphs, nodes, and edges. Labels are C++ classes which may contain any user specific methods or data and get informed of any changes to the structure of the graph. Some labels are included in the distribution to give an example on how to use labels[2].
Special graph labels which are subclasses of **FFGraphWindow** may be assigned to a graph in order to display it in up to 3 dimensions. The user may provide his/her own window labels to display the graph the way it best suits his/her demands. The distribution contains window labels which display 2 or 3 dimensional graphs using the tcl/tk[Ous94] widget library for X11 systems[3].
To display the contents of labels, inspectors are used. The class **FFTclWindow** contains an inspector that allows one to inspect the content of a label or to invoke the inspector of a label if the label has an inspector on its own.
A special interface is provided for labels that implement layout algorithms. Two layout labels implementing a 3 dimensional spring-embedder and a sugiyama layout algorithm[4] are included in the distribution.
A graph editor is included in the distribution to give an example of how to use the library.
There are several standard template container classes like sets, maps, etc. included in the distribution. These classes are naively modified and debugged versions of the libg++ container classes and are to be replaced soon by more efficient implementations and in some cases by better types of containers. For more information on how to use the container classes see the libg++ documentation.
The library also contains a slightly modified version of a Xpm extension to tk 4.0 by Mark Crimmins (markcrim@umich.edu) which is based on a work by Harald Albrecht (albrecht@igpm.rwth-aachen.de).

[1] Compare section A on page 54
[2] E.g. there is a label implementing nested graphs included in the distribution.
[3] See classes **FFTclBasicWindow**, **FFTclWindow** and **FFTclSubWindow**
[4] Written by Falk Schreiber (schreibe@fmi.uni-passau.de)

# 2 Installing the library

The library has already been successfully installed on the following platforms:

- Solaris 2.4 using gcc 2.6.8, tcl 7.3 and tk 3.6
- Linux 1.2.13 using gcc 2.7.0. tcl 7.4 and tk 4.0
- Sinix, using the Sinix C++ compiler and the motif widget set (although I have never seen this one myself up to now)

To install the library you need the following programs and libraries installed on your system:

- g++ version 2.6.3 or higher.
- libg++ according to the compiler.
- X11R5 or higher

either

- tcl 7.4 and tk 4.0 (recommended)

or

- tcl 7.3 and tk 3.6 optional with the tkPixmap extension. (will no longer be supported)

If you want to compile the .y and .l files you need[5]:

- bison++ 1.21-8
- flex++[6] 2.3.8-7

If you want to change some of the configure files you also need:

- autoconf
- autoheader

If you want to be able to use Xpm images you also need the

- Xpm library (recommended)

To install the library do the following:

1. You might want to edit the top section of `./include/ffgraph/ffGlobdefs.h.in`.
2. Type `./configure`. You may want to pass some options like `--prefix=/usr/local` to configure.
3. `make dependall` (optional, but recommended)
4. `make`
5. `make install`
6. `make cleanall` (if you want to save disk-space)
7. If all worked fine, stand up and pat yourself on the shoulder

[5]Precompiled .cc and .h files are included in the distribution

[6]The *real* flex++, not the one from the flex distribution !!!

**Troubleshooting:**
If something went wrong, maybe one of the following could help:

- Error:

  **make: Warning: Can't find 'Makefile.dependencies'**

  Solution:

  Somehow a dependencies file got lost. This can happen if you interrupt **make** at a wrong moment. Go to the corresponding directory and type **touch Makefile.dependencies**.

- Error:

  **gcc: tk.h: No such file or directory**

  Solution:

  The tcl/tk header files are not where the library expects them. Add the directory to **INCLUDES** in **Makefile.head.common.in** and rerun **configure**.

- Error:

  **ld: fatal: library -lxxx: not found**

  Solution:

  The library xxx ist not where the ffgraph library expected it. Add the apropriate directory to **LDFLAGS** in **Makefile.head.common.in** and rerun **configure**.

- Error:

  Any other error

  Solution:

  – Find the error, fix it and send a patch to **friedric@fmi.uni-passau.de**
  – Send a bug report to **friedric@fmi.uni-passau.de** containing:
    * Kind of computer you use
    * Kind and version of operating system you use
    * Kind and version of compiler you use
    * Version of ffgraph library you use
    * Detailed description of the error, including a description of how to reproduce it.

# 3 Using the graph library

There are two ways to use the library. The first one is to use it as a new graph data-type for your graph algorithms. The second one is to bind data to a graph. The second way is normally used if you have any data which skeleton structure is a graph, e.g. data-flow graphs. The library helps you to maintain and work with your data and gives you powerful features to display the data. The main classes to access the library are **FFGraph**, **FFGraphClass**, **FFNodeClass** and **FFEdgeClass**. In addition there are two data types **FFNode** and **FFEdge** which are pointers to **FFNodeClass** and **FFEdgeClass**. **FFGraph** is a container for **FFGraphClass** objects. The user has no direct access to the class **FFGraphClass**. The user can call methods on **FFGraphClass** objects by using the `->` operator on **FFGraph** objects. E.g. to create a new graph and add a node to the graph you would write:

```
FFGraph g;
g->addNode();
```

The reason for this is the overloaded `->` operator in **FFGraph** which passes all method calls to the underlying **FFGraphClass** object. Copies of **FFGraph** objects share the same **FFGraphClass** object. This allows an

efficient garbage collection and protects the user from making any of the usual pointer errors like uninitialized pointers or pointers to invalid addresses. If you want a copy of a graph you can use the method **FFGraphClass::copy(...)**. However **FFNode** and **FFEdge** are only pointers, for efficiency reasons so beware[7]!
For a detailed description of all methods of the classes **FFGraph**, **FFGraphClass**, **FFNodeClass** and **FFEdgeClass** see sections 9.1 on page 21.
The following data types which are not C++ classes are defined in the library.

- **BOOL**

  **BOOL** is a boolean data type with the two values **YES** and **NO**.

- **FFCoord**

  **FFCoord** is a floating point data type.

If you have a label that should be bound to every graph, node or edge, you can use the methods

- **void addDefGraphLabel(FFGraphLabel *)**
- **void addDefNodeLabel(FFNodeLabel *)**
- **void addDefEdgeLabel(FFEdgeLabel *)**

to specify default labels for them.

The library uses two kinds of error handlers:

- **ffError(errGrade, errKind, char *file, int line)**
- **ffError(errGrade, char *msg, char *file, int line)**

The library calls these functions whenever an error **kind** or **msg** of grade **grade** occurred in file **file** at line **line**. The values of **grade** and **kind** are defined in **ffError.h**. If you want to display errors differently, you can tell the library to use your own error-handler by calling:

- **setErrorHandler(kindErrorHandler, charErrorHandler)**

  **kindErrorHandler** and **charErrorHandler** are defined in **ffError.h**. If you pass **NULL** as a handler, the handler remains unchanged.

# 4 Compiling your program

This section describes how to compile a program using the various features of the library. It is assumed that you already have installed the library on your system. See section 2 on page 5 for further information on how to install the library.

## 4.1 Using graphs only

To compile C++ files using the graph library you must have the following programs and libraries installed on your system:

- g++ Version 2.6.3 or higher.
- libg++ according to the g++ version.

You must include some or all of the include files **FFGraph.h**, **FFGraphClass.h**, **FFEdgeClass.h**, **FFNodeClass.h** and **ffError.h** in you C++ files, depending on which methods you use. You might also need to tell the compiler where to find these files[8]. To link a program using the graph library you must add **-lFFGraph -lffcontainer** to the linker options. You might also have to tell the linker where to find the library[9].

[7] Using the container mechanism for FFNode and FFEdge would slow down the library by 30%
[8] Normally using the **-I** option, see your compiler manual for details
[9] Normally using the -L option, see your compiler manual for details

### 4.2 Using graphs and graphic

If you want to use the graphic facilities provided by the library you must have the following additional libraries installed on your system:

- The X11R5 library or higher
- either
  - tcl 7.4 with tk 4.0 (recommended)

  or

  - tcl 7.3 with tk 3.6 (will no longer be supported in the future)
- optional but highly recommended: The Xpm library.
- the tkPixmap extension, if you have tk 3.6 and want to use Xpm images.

You may have to include some or all of the files **FFTclWindow.h**, **FFGraphicNode.h**, **FFGraphicEdge.h** and **tk.h** depending on which methods you use. You also have to add **-lFFGraphic -ltk -ltcl -lXpm -lX11 -lm** to your library list. If you use tk 4.0 and do not have the Xpm library installed on your system you must leave out **-lXpm**. Or better install the Xpm library first.

### 4.3 Using graphs, graphic and layouters

You must do all the above and additionally include the files **FFSugi.h** and **FFSpring.h**. You also have to add **-lflayout** to your library list.

## 5 Labels

The user can bind his/her own C++ classes to graphs. These classes are called labels. You can bind your labels to graphs, nodes, and edges using graph-labels, node-labels or edge-labels. If you bind a node-label or an edge-label to a graph, a prototype of the label is bound to every node or edge in the graph and will be bound to every node or edge that will be added to the graph in the future. There are 4 basic label classes included in the library. **FFGraphLabel**, **FFNodeLabel** and **FFEdgeLabel** which are all subclasses of **FFLabelClass**. All graph-labels are subclasses of **FFGraphLabel**, **FFNodeLabel** for nodes and **FFEdgeLabel** for edges. When a graph is loaded or saved the content of the labels may be loaded or saved as well. Whether a label is saved/loaded depends on the implementation of the label. It is even possible that only the labels of certain nodes or edges are saved/loaded. When a graph is copied the contents of the labels may be copied as well. Whether a label is copied depends on the implementation of the label. Either all or no labels of a certain type are copied. Labels get informed of any structural changes to the graph by events that are sent to the labels. A label assigned to a graph must have a unique type. Only one label class of each type can be assigned to a certain graph. The type is determined by the result of the method **const String& FFLabelClass::type() const** which each label class provides. The type may not correspond to the C++ type in any way. The type of a label usually consists of an arbitrary string preceded by the type of its superclass. The two strings are separated by ":".

### 5.1 Using labels

Labels are subclasses of the classes **FFGraphClass**, **FFNodeClass** or **FFEdgeClass** which are subclasses of the class **FFLabelClass**. The library contains methods to add, access and delete labels from graphs.
To add labels to a graph use the methods

- **FFGraphClass::addGraphLabel(FFGraphLabel *label)**

- **FFGraphClass::addNodeLabel(FFNodeLabel *label)**
- **FFGraphClass::addEdgeLabel(FFEdgeLabel *label)**

**label** must be created using the **new** operator[10]. After calling one of these methods **label** belongs to the graph library and will usually be freed by it using the **delete** operator. Copies of node-labels and edge-labels will automatically be assigned to every node and edge in the graph and to all nodes and edges that will be added to the graph in the future.
The return value of these methods is the index of the added label. This index is local to a graph and may differ from graph to graph. You can access a label through its type or its index. Access through index is recommended as it is much faster[11]. To access a label by its index or type you can use one of the following methods:

- **FFGraphLabel *FFGraphClass::getGraphLabel(int i)**

  Returns the graph label with index **i**, or **NULL** if the graph has no label with index **i**.

- **FFGraphLabel *FFGraphClass::getGraphLabel(const String& type, AVLSet<FFGraphLabel *> *set=NULL)**

  If **set == NULL** the label with type **type** is returned, or **NULL** if the graph has no graph label of type **type**.

  If **set != NULL** all labels that contain **type** in their type are added to **set**. The first label found is returned. If no graph label-type contains **type**, **NULL** is returned.

- **const MPlex<FFGraphLabel *>& FFGraphClass::getGraphLabels() const**

  Returns a MPlex containing all graph labels.

- **const MPlex<FFNodeLabel *>& FFGraphClass::getNodeLabels() const**

  Returns a MPlex containing the prototypes of all node labels.

- **const MPlex<FFEdgeLabel *>& FFGraphClass::getEdgeLabels() const**

  Returns a MPlex containing the prototypes of all edge labels.

- **FFNodeLabel *FFNodeClass::getLabel(int i)**

  Returns the node label with index **i** or **NULL** if the node has no label with index **i**.

- **const MPlex<FFNodeLabel *> *FFNodeClass::getLabels() const**

  Returns a MPlex containing all labels bound to the node.

- **FFEdgeLabel *FFEdgeClass::getLabel(int i)**

  Returns the edge label with index **i** or **NULL** if the edge has edge label with index **i**.

- **const MPlex<FFEdgeLabel *> *FFEdgeClass::getLabels() const**

  Returns a MPlex containing all labels bound to the node.

If you did not store the index you got by adding a label, you can retrieve its index by using one of the following methods:

- **int FFGraphClass::getGraphLabelIndex(const String& type, AVLSet<int>*set=NULL) const**

  If **set == NULL** the index of the graph label with type **type** is returned or **-1**, if the graph has no graph label of type **type**.

  If **set != NULL** the indexes of all graph labels that contain **type** in their type are added to **set**. The first index found is returned. If no graph label type contains **type**, **-1** is returned.

[10] Unless the manual of a certain label says anything else
[11] index: $O(1)$, type: $O(|labels|)$

- **int FFGraphClass::getNodeLabelIndex(const String& type, AVLSet<int>*set=NULL) const**

  If **set == NULL** the index of the node label with type **type** is returned or **-1**, if the graph has no node label of type **type**.

  If **set != NULL** the indexes of all node labels that contain **type** in their type are added to **set**. The first index found is returned. If no node label type contains **type**, **-1** is returned.

- **int FFGraphClass::getEdgeLabelIndex(const String& type, AVLSet<int>*set=NULL) const**

  If **set == NULL** the index of the edge label with type **type** is returned or **-1**, if the graph has no edge label of type **type**.

  If **set != NULL** the indexes of all edge labels that contain **type** in their type are added to **set**. The first index found is returned. If no edge label type contains **type**, **-1** is returned.

- **int FFNodeClass::getLabelIndex(const String& type, AVLSet<int>*set=NULL) const**

  If **set == NULL** the index of the node label with type **type** is returned or **-1**, if the node has no node label of type **type**.

  If **set != NULL** the indexes of all node labels that contain **type** in their type are added to **set**. The first index found is returned. If no node label type contains **type**, **-1** is returned.

- **int FFEdgeClass::getLabelIndex(const String& type, AVLSet<int>*set=NULL) const**

  If **set == NULL** the index of the edge label with type **type** is returned or **-1**, if the edge has no edge label of type **type**.

  If **set != NULL** the indexes of all edge labels that contain **type** in their type are added to **set**. The first index found is returned. If no edge label type contains **type**, **-1** is returned.

If you want to change a label, you can get the label using one of the above methods and change the label by calling its methods.
If you want to assign the contents of one label to another, you can either

- Get the label and call the method **void FFLabelClass::assign(const FFLabelClass*)** which every label provides. Of course you cannot be sure that the implementation of this method really does anything at all.

or

- Use one of the following methods:

  - **void FFGraphClass::assignGraphLabel(const FFGraphLabel *label, int i=-1)**
    calls the method **void FFLabelClass::assign(const FFLabelClass*)** of the graph label with index **i** and argument **label**. If **i** == -1 the graph checks whether it is labeled with a label of the same type as **label** and calls **assign** if it finds one.
  - **void FFGraphClass::assignGraphLabel(const MPlex<FFGraphLabel *>& set)**
    For each label in **set** the graph checks if it is labeled with a label of the same type. If it finds one, the method **void FFLabelClass::assign(const FFLabelClass*)** is called with the label from **set** as argument.
  - **void FFGraphClass::setGraphLabel(FFGraphLabel *label, int index=-1, BOOL doFree=YES)**
    Replaces the graph label of index **index** with **label**. If **doFree == YES** the old label is deleted using the **delete** operator. If **index == -1** the graphs checks whether it is labeled with a label of the same type as **label** and does the replacement if it finds one. After calling this method **label** belongs to the graph and will be freed by the graph using the **delete** operator when it is no longer needed.

- **void FFGraphClass::setGraphLabel(const MPlex<FFGraphLabel *>& set, BOOL doFree=YES)**
  For each label in **set** the method replaces the graph label of the same type with the label from **set**. If **doFree == YES** the old label is deleted using the **delete** operator. After calling this method all labels of **set** belong to the graph and will eventually be freed by the graph using the **delete** operator.
- Use one of the corresponding methods for **FFNodeClass** and **FFEdgeClass**.

If you no longer need a label, you can remove it from the graph using one of the following methods:

- **void FFGraphClass::delGraphLabel(int index, BOOL doFree=YES)**
  **void FFGraphClass::delGraphLabel(const String& type, BOOL doFree=YES)**
  Deletes the graph label with index **index** or type **type** from the graph. If **doFree == YES** the label is freed using the **delete** operator.
- **void FFGraphClass::delEdgeLabel(int index, BOOL doFree=YES)**
  **void FFGraphClass::delEdgeLabel(const String& type, BOOL doFree=YES)**
  Deletes the edge labels with index **index** or type **type** from all edges of the graph. If **doFree == YES** the labels are freed using the **delete** operator.
- **void FFGraphClass::delNodeLabel(int index, BOOL doFree=YES)**
  **void FFGraphClass::delNodeLabel(const String& type, BOOL doFree=YES)**
  Deletes the node labels with index **index** or type **type** from all nodes of the graph. If **doFree == YES** the labels are freed using the **delete** operator.

Graph-labels can contain an inspector to display their content. Inspectors are subclasses of **FFInspector**. The kind of information displayed by the inspector depends on the implementation of the inspector. E.g. the inspector contained in **FFTclWindow** displays the result of the **String FFLabelClass::unparse()** method for graph, node or edge labels.
To determine whether a label has its own inspector, you can use the method:

- **BOOL FFGraphLabel::hasOwnInspector()**

To display an inspector you can use the method:

- **void FFGraphLabel::showInspector()**

To update the display of the inspector you can use the method:

- **void FFGraphLabel::updateInspector()**

## 5.2 Writing labels

If you want to use your own label, you have to write a subclass of **FFGraphLabel**, **FFNodeLabel** or **FFEdgeLabel** with any additional data and methods you want. You then can add it to the graph using one of the **add...Label(...)** methods. Labels get informed when the structure of the graph is changed. To overwrite the method **void event(labelEvent, labelEventStruct *)** is the easiest way to respond to these changes to the graph. To enable the library to handle your label you must implement some additional methods. The methods you must implement are:

- Write a constructor for your class. In the constructor you have to call the method **void FFLabelClass::_setType(const String& s)** with an arbitrary String **s**. **s** must not contain ":" and must be unique. Using the C++ type of the label class often is a good choice.
- Depending on what kind of label you are writing, one of

  - **FFGraphLabel *YourLabelClass::proto()**
  - **FFNodeLabel *YourLabelClass::proto()**
  - **FFEdgeLabel *YourLabelClass::proto()**

  This method must return a new instance of your label class. The new instance should be allocated using the **new** operator. This method is needed to assign labels to nodes, edges and copies of graphs.

  If you e.g. write a graph label usually writing something like

```
FFGraphLabel *YourLabelClass::proto()
{
  return new YourLabelClass;
}
```

  is sufficient.

There are also some methods you might want to overwrite, but you do not have to:

- **BOOL FFLabelClass::isSaveable() const**

  If this method returns **YES** the result of the method **String FFLabelClass::unparse()** is stored in a file when a graph is saved. When a graph is loaded the method **BOOL FFLabelClass::parse()** is called with the string saved before.

  If this method returns **NO** the label is not stored in the file. The results of the method may differ between two nodes or edges in the graph for the same label.

  This method defaults to return **NO**.

- **BOOL shouldCopy() const**

  If a graph gets copied and this method returns **YES** the new graph is labeled with the label as well. The **assign** method is called for the label in the new graph with this label as argument. For node and edge labels **assign** is called for labels in the corresponding node or edge in the new graph.

  This method defaults to return **YES**.

- **BOOL isInspectable() const**

  Inspector classes are used to display the contents of a label on screen. Usually the result of calling the method **String FFLabelClass::unparse()** is displayed. If this method returns **YES** it can be displayed by an inspector.

  This method defaults to return **NO**.

- **BOOL compare(const FFLabelClass *label) const**

  This method should return if **label** is equal to this label.

- **void assign(const FFLabelClass *label)**

  This method should assign the content of **label** to **this**.

- **BOOL parse(const String& str)**

  This method should be able to initialize itself according to **str**. **str** is usually the result of a previous call to **String FFLabelClass::unparse()**. This method is e.g. called when a graph is loaded from a file or by an inspector to display the content of the label. The method should return whether it could successfully parse **str**.

- **String unparse()**

  This method should return the content of the label as a **String**. An instance of the label class should be able to initialize itself if the result of invoking **unparse()** is passed to **BOOL parse(const String& str)**. This method is e.g. called when a graph is saved to a file or by an inspector.

- **void event(labelEvent kind, labelEventStruct *data)**

  Whenever there is a structural change to the graph, the **event** method is invoked. **kind** specifies the kind of event that happened and the union **data** contains some additional information that varies between different kinds of events. **event** is not always invoked in all labels, but there are some events that are e.g. only invoked in graph labels etc.

  **labelEventStruct** is defined as follows:

```
union {
  FFNode node;
  FFEdge edge;
  int index;
} labelEventStruct;
```

  The following kinds of events may occur:

  - **ADD_NODE**

    Is invoked after a new node has been added to the graph. **data->node** contains the new node.

    Is delivered to all graph labels and to all node labels, including the new node.

  - **ADD_EDGE**

    Is invoked after a new edge has been added to the graph. **data->edge** contains the new edge.

    Is delivered to all graph labels and to all edge labels, including the new edge.

  - **DEL_NODE**

    Is invoked before a node is removed from the graph. **data->node** contains the node that will be removed.

    Is delivered to all graph labels and to all node labels, including the node that will be deleted.

  - **DEL_EDGE**

    Is invoked before an edge is removed from the graph. **data->edge** contains the edge that will be removed.

    Is delivered to all graph labels and to all edge labels, including the edge that will be deleted.

  - **DEL_GRAPH**

    Is invoked before a graph is deleted.

    Is delivered to all graph labels.

  - **REV_EDGE**

    Is invoked after the direction of an edge has changed.

    Is delivered to all labels of this edge. **data->edge** contains the edge that has been reverted.

  - **SET_STARTNODE**

    Is invoked after a node has been marked as being a start-node.

    Is delivered to all graph labels and to all labels of the new start-node. **data->node** contains the new start-node.

- **DEL_STARTNODE**

  Is invoked before a node will be marked as no longer being a start-node.

  Is delivered to all graph labels and to all labels of the former start-node. **data**−>**node** contains the former start-node.

- **SET_ENDNODE**

  Is invoked after a node has been marked as being an end-node.

  Is delivered to all graph labels and to all labels of the new end-node. **data**−>**node** contains the new end-node.

- **DEL_ENDNODE**

  Is invoked before a node will be marked as no longer being an end-node.

  Is delivered to all graph labels and to all labels of the former end-node. **data**−>**node** contains the former end-node.

- **NAME_CHANGE**

  Is invoked after the name of a graph, node or edge has changed.

  Is delivered to either all graph labels, all node labels for the node which changed or all edge labels for the edge which changes. **data** contains the according node or edge.

- **ADD_NODELABEL**

  Is invoked after a new node label has been added to the graph.

  Is delivered to all graph labels and to all node labels. **data**−>**index** contains the index of the new label.

- **ADD_EDGELABEL**

  Is invoked after a new edge label has been added to the graph.

  Is delivered to all graph labels and to all edge labels. **data**−>**index** contains the index of the new label.

- **ADD_GRAPHLABEL**

  Is invoked after a new graph label has been added to the graph.

  Is delivered to all graph labels, to all edge labels, and to all node labels. **data**−>**index** contains the index of the new label.

- **DEL_NODELABEL**

  Is invoked before a node label will be removed from the graph.

  Is delivered to all graph labels and to all node labels, including the one that will be removed. **data**−>**index** contains the index of the label that will be removed.

- **DEL_EDGELABEL**

  Is invoked before an edge label will be removed from the graph.

  Is delivered to all graph labels and to all edge labels, including the one that will be removed. **data**−>**index** contains the index of the label that will be removed.

- **DEL_GRAPHLABEL**

  Is invoked before an graph label will be removed from the graph.

  Is delivered to all graph labels, to all node labels, and to all edge labels, including the one that will be removed. **data**−>**index** contains the index of the label that will be removed.

- **SOURCE_CHANGE**

  Is invoked after the source node of an edge changed.

  Is delivered to all labels of the edge that changed. **data**−>**edge** contains the edge that changed.

- **TARGET_CHANGE**

  Is invoked after the source node of an edge changed.

  Is delivered to all labels of the edge that changed. **data**−>**edge** contains the edge that changed.

– **DIRECTEDNESS_CHANGE**
  Is invoked after the directedness of the graph changed.
  Is delivered to all graph labels, to all node labels, and to all edge labels.

If you write a graph label and want to provide an inspector for the label you have to assgin the inspector to the **inspector** member variable. This is usually done in the constructor of the label. The label must be passed to the constructor of the inspector.
The following methods check if the member variable **inspector != NULL** to determine their behaviour or return value:

- **BOOL FFGraphLabel::hasOwnInspector()**
  Returns **YES** to indicate that the the label has its own interpreter if **inspector != NULL**
- **void FFGraphLabel::showInspector()**
  Displays the inspector on the screen if **inspector != NULL**. Usually by calling the method **void FFInspector::show()**.
- **void FFGraphLabel::updateInspector()**
  Updates the display of the inspector if **inspector != NULL**. Usually by calling **void FFInspector::inspect()**

# 6 Inspectors

Inspectors are C++ classes which are usually bound to graph labels. They are normally used to display the contents of a label in a window on the screen.
E.g. the window label **FFTclWindow** uses the class **FFTclGraphInsp** to display the contents of graph, node and egde labels.

## 6.1 Using inspectors

Inspectors are subclasses of **FFInspector**. Inspectors are usually invoked by the graph label they are bound to and stay on the screen until the label is deleted or the window is dismissed by the user. See the corresponding graph label methods for details on how to invoke inspectors.

## 6.2 Writing inspectors

If you write your own graph label, you might also want to write an inspector for it to display its contents. To write an inspector you have to subclass **FFInspector** and implement the following methods:

- **FFInspector(FFGraphLabel *g)**
  Initialize the inspector according to **g**. **g** is the label the inspector is bound to.
- **virtual void show()**
  Should display the inspector on the screen.
- **virtual void hide()**
  Should remove the inspector from the screen.
- **virtual void inspect() = NULL**
  Display the contents of the label the inspector is bound to, or whatever you want to display.

If you want to use the tcl/tk widget set to display the inspector, you might want to subclass **FFTclInspector** which is a subclass of **FFInspector**.

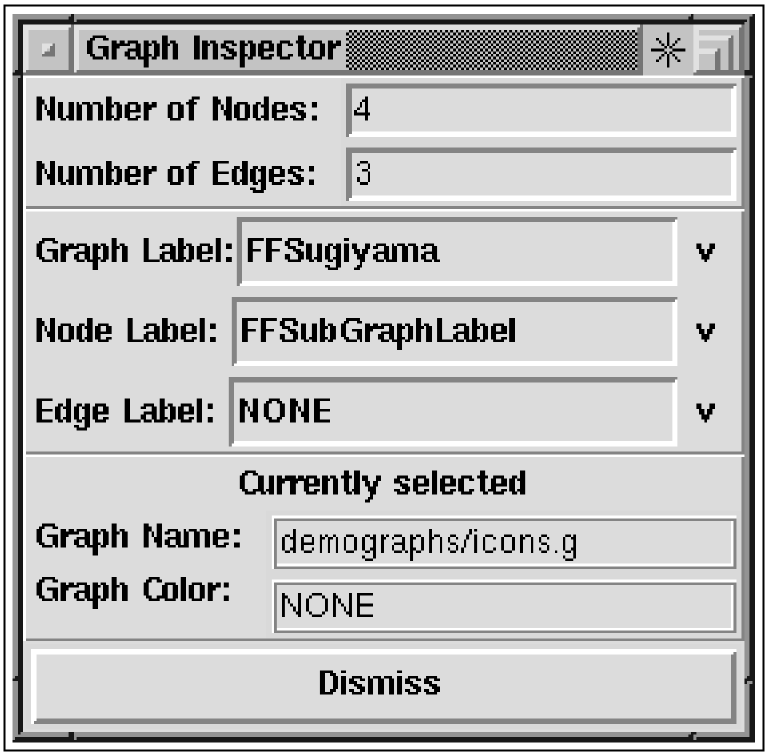


Figure 1: FFTclGraphInsp

# 7 Graphic contexts

To display a graph you can bind a graphic context to a graph. Graphic contexts are C++ classes which are subclasses of **FFGraphWindow** which is a subclass of **FFGraphLabel**. How the graph is displayed depends on the graphic context you use. This may range from a window on the screen to an output to a printer. When a graphic context is bound to a graph, the graph is automatically labeled with the node label **FFGraphicNode** and the edge label **FFGraphicEdge** which contain graphic information like coordinates, colors, etc. If you just want the graphic information but do not want to actually display the graph, e.g. to do some layouting, you can use the special graphic context **NULL**.
The library contains the class **FFTclWindow** which is able to display 2 and 3 dimensional graphs.
It is also able to display nested graphs:

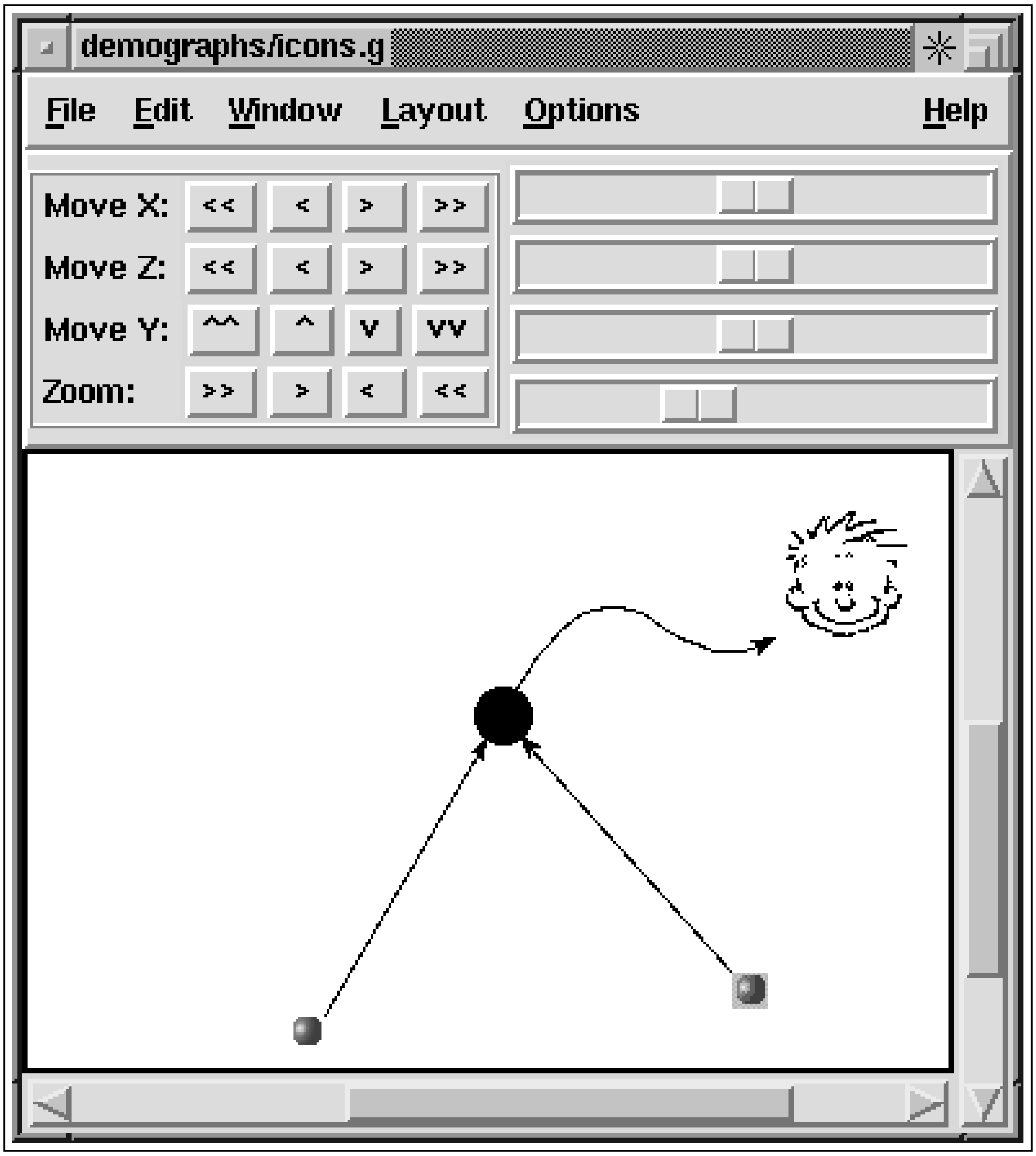


Figure 2: Graph containing icons

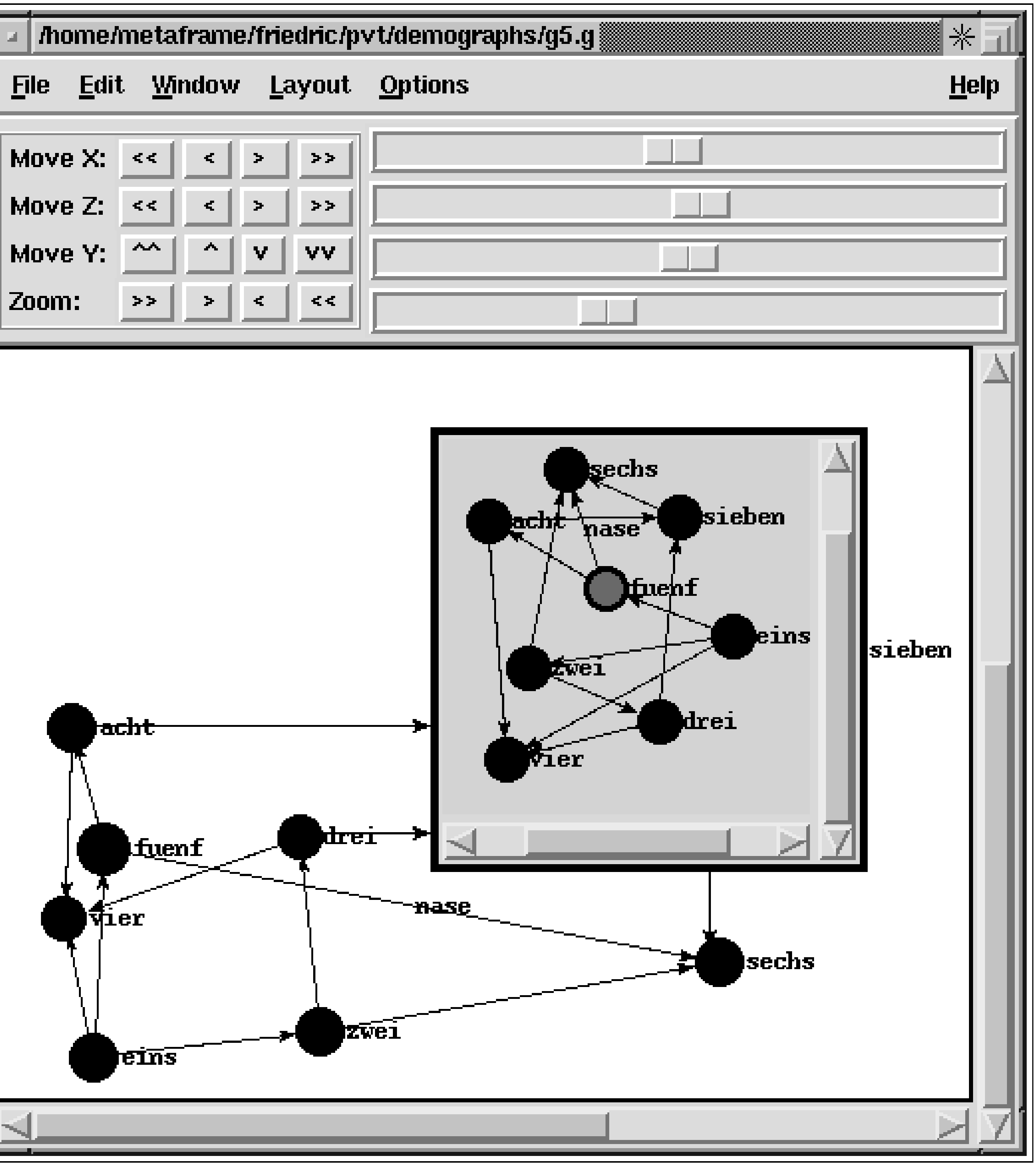


Figure 3: Graph containing a subgraph

## 7.1 Using graphic contexts

To assign a graphic context to a graph you should use the method

- **void FFGraphClass::createGraphic(FFGraphWindow *window)**.

E.g. to display a graph using the graphic context **FFTclWindow** you could write:

```
FFGraph g;
g->createGraphic(new FFTclWindow);
```

If you want to use graphic information without actually displaying the graph, you can call createGraphic with **window == NULL**.
If a graphic context is assigned to a graph, it can be retrieved using the method

- **FFGraphWindow *FFGraphClass::getWindow()**

At the moment only one subclass of **FFGraphWindow** can be assigned to a graph. If you no longer want to display the graph, you can use the method

- **void FFGraphClass::removeGraphic()**

which removes the graph-window label and the labels **FFGraphicNode** and **FFGraphicEdge**. The labels are freed using the **delete** operator. If you do not want the graph window label to be freed, e.g. to use it for another graph, you can use the method

- **FFGraphWindow *FFGraphClass::leaveWindow()**

## 7.2 Writing graphic contexts

To write your own graphic context, you can either write a new one from scratch, or subclass an existing one and only overwrite the methods you want to change. Graphic contexts are designed to be able to display 3 dimensional graphs. If you do not want to display 3D graphs, just implement the appropriate methods to ignore 3D information. A graphic context must be a subclass of **FFGraphWindow** and must implement all methods **FFGraphWindow** defines. See section 9.3.2 on page 36 for details on what methods you have to implement. When **FFGraphClass::createGraphic(...)** is invoked with your window label as argument, the graph labels itself with the node label **FFGraphicNode** and the edge label **FFGraphicEdge**. You should use these labels to store and get the graphic information you need to display the graph. See sections 9.3.3 on page 40 and 9.3.4 on page 43 for further information on how to use these labels.

# 8 Layouters

Layouters are used to arrange the nodes and the edges of a graph on the screen to optimally suit the representation of the data the graph contains. As there are a huge number of different data a graph may contain and an even larger number of opinions on what an optimal representation of this data is, there exists a huge amount of different layout algorithms. In the ffgraph library layout algorithms are implemented as graph labels. By this you can add specific layouters to specific graphs and other classes like windows can determine at runtime which layouters are available. These classes can even use layouters they did not know about when they were written. E.g. the graph-window class **FFTclWindow** included in the distribution automatically adds all layouters available for a graph to its layout menu.

## 8.1 Using layouters

Layouters are usually subclasses of **FFLayoutLabel** which is a subclass of **FFGraphLabel**. Of course layouters do not need to be implemented as graph labels. This subsection only describes how to use the layouters included in the library. It should work for other layouters as well, but may not.
In your program you have to add layout labels to all graphs you want to layout using the **FFGraphClass::addGraphLabel(FFGraphLabel *)** method. E.g.

```
FFGraph g;
g->addGraphLabel(new FFSugi);
g->addGraphLabel(new FFSpring);
```

You can then use the layouter using one of the following methods:

- **void FFLayoutLabel::layout()**

  The whole graph is layouted.

- **void FFLayoutLabel::layout(AVLSet<FFNode>& set)**

  Only the subgraph defined by the nodes in **set** is layouted.

If you want to change the parameters of a layouter, you can either use the methods the layouters provide[12], or if you use **FFTclWindow** to display the graph, you can set the parameters at runtime using the inspector included in the library.
If you use the class **FFTclWindow** to display the graph, the new layouter is automatically added to its layout menu.

## 8.2 Writing layouters

The easiest way to write your own layouter is to subclass **FFLayoutLabel** and implement these methods:

- **void FFLayoutLabel::layout()**

  To layout the whole graph.

- **void FFLayoutLabel::layout(AVLSet<FFNode>& set)**

  To layout the subgraph defined by the nodes in **set**.

If you use **FFTclWindow** to display the graph, you only have to add your layouter to the graph using **int FFGraphClass::addGraphLabel(FFGraphLabel *)** and it will automatically be added to the layout menu of **FFTclWindow**. If the menu is invoked, the method **void FFLayoutLabel::layout()** is called. If more than one node is selected, **void FFLayoutLabel::layout(AVLSet<FFNode>)** is called.
That is all you basically have to do. If you want the user to be able to set parameters at runtime the usual way to do this is to use an inspector. To use an inspector you have to overwrite the method

- **BOOL FFLabelClass::isInspectable()**

to return **YES**.
You then have to decide if you want to use your own inspector or if you want to use the inspector included in the library. If you want to use the given inspector, you just have to implement the methods:

- **string FFLabelClass::unparse()**
  and
  **BOOL FFLabelClass::parse(String&)**

  to read and write the layout parameters in string form.

[12]see sections 9.2.7 on page 32 and 9.2.8 on page 34

If you want to use your own inspector, you have to overwrite the method

- **BOOL FFGraphLabel::hasOwnInspector()**

to return **YES**. You also have to write your own inspector class. You can do this subclassing **FFInspector**[13].

# 9 The important classes in detail

## 9.1 Standard classes

### 9.1.1 FFGraph

The class **FFGraph** is a container for the class **FFGraphClass**. As the user has no direct access to **FFGraphClass**, using **FFGraph** is the only way to call methods in **FFGraphClass** objects. This is done by using the -> operator. E.g. to add a node to a graph you would write:

```
FFGraph g;
g->addNode();
```

FFGraph does an automatic garbage collection. This means that if there is no container for a graph left, the graph is freed.
FFGraph defines the following methods:

- **FFGraph()**

  Creates a new graph. The underlying **FFGraphClass** object is not allocated before it is actually used.

- **FFGraph& operator = (const FFGraph& graph)**

  Sets the object to be a container for the same graph as **graph**.

- **int operator == (const FFGraph& graph)**

  Returns whether the object is a container for the same graph as **graph**.

- **int operator != (const FFGraph& graph)**

  Returns whether the object is not a container for the same graph as **graph**.

- **FFGraphClass * operator -> () const**

  Passes the method to the underlying **FFGraphClass** object.

### 9.1.2 FFGraphClass

The class **FFGraphClass** provides standard methods to manipulate graphs. The methods of **FFGraphClass** can not be accessed directly, but must be accessed using the class **FFGraph**.
**FFGraphClass** defines the following methods:

- **BOOL getDirected()**

  Returns whether the graph is directed. This method is intended to be used in the future, as only directed graphs are supported at the moment. May return any value.

- **int getGraphicNodeIndex()**

  If the nodes of the graph are labeled with **FFGraphicNode**, the index of the label is returned. If not, -1 is returned.

[13] See section 6.2 on page 15 for further details

- **int getGraphicEdgeIndex()**

  If the edges of the graph are labeled with **FFGraphicEdge**, the index of the label is returned. If not, -1 is returned.

- **int getGraphWindowIndex()**

  If the graph is labeled with a window label, the index of the label is returned. If not, -1 is returned.

- **void load(const String& buffer, BOOL datei)**

  If **datei == YES** the graph specified in the sgraph compatible file **buffer** is merged into the graph.
  If **datei == NO** the graph that is contained in **buffer** in sgraph format is merged into the graph.

- **void saveToFile(const String& file)**

  The graph is saved to the file **file**.

- **void saveToFile(const String& file, const AVLSet<FFNodeClass *>& set)**

  The subgraph defined by the nodes in **set** and the edges between the nodes in **edge** is saved to file **file**.

- **void saveToString( String& str)**

  The graph is saved to the string **str**.

- **void saveToString( String& str, const AVLSet<FFNodeClass *>& set)**

  The subgraph defined by the nodes in **set** and the edges between the nodes in **edge** is saved to string **str**.

- **FFNode addNode(const FFPoint& p=FFPoint(100,100,0))**

  A new node is added to the graph and returned. If the nodes of the graph are labeled with **FFGraphicNode**, the node is moved to position **p**.

- **FFEdge addEdge(FFNode s, FFNode t)**

  A new edge between source node **s** and target node **t** is added to the graph and returned. **s** and **t** must belong to this graph.

- **void removeNode(FFNode n)**

  Node **n** is removed from the graph. All edges from or to **n** are removed from the graph as well.

- **void removeEdge(FFEdge e)**

  Edge **e** is removed from the graph.

- **void clear()**

  All nodes and edges are removed from the graph.

- **void allNodes(AVLSet<FFNodeClass *>& set)**

  All nodes of the graph are added to set **set**.

- **void allEdges(AVLSet<FFEdgeClass *>& set)**

  All edges of the graph are added to set **set**.

- **const AVLSet<FFNodeClass *>& allNodes() const**

  A set containing all nodes of the graph is returned.

- **const AVLSet<FFEdgeClass *>& allEdges() const**

  A set containing all edges of the graph is returned.

- **int getNodeLabelIndex(const String& type, AVLSet<int> *set=NULL) const**

  If **set == NULL** the index of the node label with type **type** is returned, or -1 if none is found. if **set != NULL** the indexes of all node labels which contain **type** in their type are added to **set**. The index of the first label that is found is returned, or -1, if none is found.

- **int getEdgeLabelIndex(const String&, AVLSet<int> *set=NULL) const**

  If **set == NULL** the index of the edge label with type **type** is returned, or -1 if none is found. if **set != NULL** the indexes of all edge labels which contain **type** in their type are added to **set**. The index of the first label that is found is returned, or -1, if none is found.

- **int getGraphLabelIndex(const String&, AVLSet<int>* set=NULL) const**

  If **set == NULL** the index of the graph label with type **type** is returned, or -1 if none is found. if **set != NULL** the indexes of all graph labels which contain **type** in their type are added to **set**. The index of the first label that is found is returned, or -1, if none is found.

- **FFGraphLabel *getGraphLabel(const String& s, AVLSet<FFGraphLabel *> *set=NULL)**

  If **set == NULL** the graph label with type **type** is returned, or **NULL** if none is found. if **set != NULL** all graph labels which contain **type** in their type are added to **set**. The first label that is found is returned, or **NULL**, if none is found.

- **FFGraphLabel *getGraphLabel(int index)**

  The graph label with index **index** is returned or **NULL** if the graph has no graph label with index **index**.

- **const MPlex<FFGraphLabel *>& getGraphLabels() const**

  An MPlex containing all graph labels is returned.

- **const MPlex<FFNodeLabel *>& getNodeLabels() const**

  An MPlex containing the prototypes for all node labels is returned.

- **const MPlex<FFEdgeLabel *>& getEdgeLabels() const**

  An MPlex containing the prototypes for all edge labels is returned.

- **int addNodeLabel(FFNodeLabel *label)**

  Prototypes of label **label** are added to all nodes in the graph. The index of the label is returned. If the nodes are already labeled with a label **label**, only the index is returned.

- **int addEdgeLabel(FFEdgeLabel *label)**

  Prototypes of label **label** are added to all edges in the graph. The index of the label is returned. If the edges are already labeled with a label **label**, only the index is returned.

- **int addGraphLabel(FFGraphLabel *label)**

  Label **label** is added to the graph. The index of the label is returned. If the graph is already labeled with a label **label**, only the index is returned.

- **void delNodeLabel(int index, BOOL doFree=YES)**

  The node labels with index **index** are removed from all nodes. If **doFree == YES** all labels are freed using the **delete** operator.

- **void delEdgeLabel(int index, BOOL doFree=YES)**

  The edge labels with index **index** are removed from all edges. If **doFree == YES** all labels are freed using the **delete** operator.

- **void delGraphLabel(int index, BOOL doFree=YES)**

  The graph label with index **index** is removed from the graph. If **doFree == YES** the label is freed using the **delete** operator.

- **void delNodeLabel(const String& type, BOOL doFree=YES)**

  Removes the node label with type **type** from all nodes in the graph. If **doFree == YES** all labels are freed using the **delete** operator.

- **void delEdgeLabel(const String& type, BOOL doFree=YES)**

  Removes the edge label with type **type** from all nodes in the graph. If **doFree == YES** all labels are freed using the **delete** operator.

- **void delGraphLabel(const String& type, BOOL doFree=YES)**

  Removes the graph label with type **type** from the graph. If **doFree == YES** the label is freed using the **delete** operator.

- **void assignGraphLabel(const FFGraphLabel *label, int i=-1)**

  Assigns **label** to the graph label with index **i** using the method **void FFLabelClass::assign(FFLabelClass *)**. If **i == -1** the graph checks if it is labeled with a label of the same type as **label** and assigns **label** to it if it finds one.

- **void assignGraphLabel(const MPlex<FFGraphLabel *>& set)**

  For all graph labels in **set**, the graph checks if it is labeled with a label of the same type as the label from **set** and assigns the label from **set** to it if it finds one.

- **void setGraphLabel(FFGraphLabel *label , int index=-1, BOOL doFree=YES)**

  Sets the graph label of index **index** to **label**. If **doFree == YES** the old label is freed using the **delete** operator. If **index == -1** the graph checks if it is labeled with a graph label of the same type as **label** and sets it to **label** if it finds one.

- **void setGraphLabel(const MPlex<FFGraphLabel *>& set, BOOL doFree=YES)**

  For all graph labels in **set**, the graph checks if it is labeled with a label of the same type as the label from **set** and sets it to the label from **set** if it finds one. If **doFree == YES** the old label is freed using the **delete** operator.

- **void setName(const String& name)**

  Sets the name of the graph to **name**.

- **const String& getName()**

  Returns the name of the graph.

- **FFNodeClass *nodeWithValue(int k)**

  Returns the node with the value **k**[14].

- **void rev()**

  Reverts all edges.

- **void eliminateParallelEdges()**

  For all pairs of parallel edges, all but one are removed from the graph. The edges to be removed are chosen randomly.

[14] compare FFNodeClass::setValue(int)

- **BOOL contains(FFNodeClass * n)**
  Returns whether the graph contains node **n**.
- **BOOL contains(FFEdgeClass * e)**
  Returns whether the graph contains edge **e**.
- **BOOL setStartNode(FFNode n)**
  Marks node **n** as start-node. **n** must be a node of the graph. Unmarks the previous start-node.
- **FFNodeClass * getStartNode()**
  Returns the node that has been marked as start-node, or **NULL** if no node has been marked as start-node.
- **BOOL setEndNodes(const AVLSet<FFNodeClass *>& set)**
  Marks all nodes of **set** as end-nodes. All nodes from **set** must be nodes of the graph.
- **BOOL addEndNode(FFNode n)**
  Marks node **n** as end-node. **n** must be a node of the graph.
- **BOOL delEndNode(FFNode n)**
  Marks node **n** as no longer being an end-node. **n** must be a node of the graph.
- **void clearEndNodes()**
  Marks all end-nodes as no longer being end-nodes.
- **const AVLSet<FFNodeClass *>& getEndNodes()**
  Returns a set containing all nodes that are marked as being end-nodes.
- **int getNumNodes() const**
  Returns the number of nodes in the graph.
- **int getNumEdges() const**
  Returns the number of edges in the graph.
- **FFGraph copy(FFGraph& graph)**
  Merges the graph into graph **graph**.
- **FFGraph copy()**
  Returns a copy of the graph.
- **BOOL containsCycle()**
  Returns whether the graph contains cycles.
- **BOOL weakConnected()**
  Returns whether the graph is weak connected.
- **BOOL strongConnected()**
  Returns whether the graph is strong connected.
- **BOOL isTree()**
  Returns whether the graph is a tree.
- **void show()**
  Prints the graph as ASCII to stdout.

- **void createGraphic(FFGraphWindow *w = NULL, FFGraphicNode *gn=NULL, FFGraphicEdge *ge=NULL)**

  Displays the graph in graphic context **w**. If **gn != NULL** it is used instead of **FFGraphicNode** to store the graphical information of the nodes. If **ge != NULL** it is used instead of **FFGraphicEdge** to store the graphical information of the edges.

- **FFGraphWindow *getWindow() const**

  Returns the graphic context the graph is displayed in, or **NULL** if the graph has no graphic context.

- **void removeGraphic()**

  Removes the graph from the graphic context it is displayed in. The graphic context is freed using the **delete** operator.

- **FFGraphWindow *leaveWindow()**

  Removes the graph from the graphic context it is displayed in. The graphic context is returned.

- **FFGraphWindow *hideGraphic()**

  Removes the graph from the graphic context it is displayed in. All graphic information in the nodes and edges remains unchanged. This means the **FFGraphicNode** and **FFGraphicEdge** classes are not deleted. The graphic context is returned.

### 9.1.3 FFNodeClass

The class **FFNodeClass** provides methods to manipulate nodes. There is no public constructor for the class **FFNodeClass**. If you want a new node, use the method **FFGraphClass::addNode(...)**. **FFNode** is a typedef to **FFNodeClass ***
FFNodeClass defines the following methods:

- **int getValue() const**

  Returns the value of the node. Counterpart to **setValue(...)**.

- **inline void setValue(int i)**

  Sets the value of the node to **i**. Counterpart to **getValue()**. This method may be used by anyone and therefore the value is only very locally valid. **This method must not be used by labels during init, parse or unparse!!!**

- **FFNode copy()**

  Returns a copy of the node in a copy of the graph.

- **inline int indeg() const**

  Returns the number of ingoing edges.

- **inline int outdeg() const**

  Returns the number of outgoing edges.

- **void eliminateParallelEdges()**

  For all parallel edges going into this node all but one are removed.

- **void assignLabel(FFNodeLabel *l, int index=-1)**

  Assigns **label** to the node label with index **i** using the method **void FFLabelClass::assign(FFLabelClass *)**. If **i == -1** the node checks if it is labeled with a label of the same type as **label** and assigns **label** to it if it finds one.

- **void assignLabel(MPlex<FFNodeLabel *>& set)**

  For all node labels in **set** the node checks if it is labeled with a label of the same type as the label from **set** and assigns the label from **set** to it if it finds one.

- **void setLabel(FFNodeLabel *l, int i=-1, BOOL doFree=YES)**

  Sets the node label of index **index** to **label**. If **doFree == YES** the old label is freed using the **delete** operator. If **index == -1** the node checks if it is labeled with a node label of the same type as **label** and sets it to **label** if it finds one.

- **void setLabel(MPlex<FFNodeLabel *>& set, BOOL doFree=YES)**

  For all node labels in **set**, the node checks if it is labeled with a label of the same type as the label from **set** and sets it to the label from **set** if it finds one. If **doFree == YES** the old label is freed using the **delete** operator.

- **int getLabelIndex(const String& type, AVLSet<int> *set=NULL) const**

  If **set == NULL** the index of the node label with type **type** is returned, or -1 if none is found. if **set != NULL** the indexes of all node labels which contain **type** in their type are added to **set**. The index of the first label that is found is returned, or -1, if none is found.

- **FFNodeLabel *getLabel(int index)**

  Returns the node label with index **index**, or **NULL** if the node has no label with index **index**.

- **const MPlex<FFNodeLabel *> *getLabels() const**

  Returns a MPlex containing all labels of the node.

- **FFGraph getGraph()**

  Returns the graph the node is in.

- **void show()**

  Prints the node to stdout.

- **AVLSet<FFNode> succ()**

  Returns a set containing the successors of the node.

- **AVLSet<FFNode> pred()**

  Returns a set containing the predecessor of the node.

- **const AVLSet<FFEdge>& transToSucc() const**

  Returns a set containing the edges to the successors of the node

- **const AVLSet<FFEdge>& transFromPred() const**

  Returns a set containing the edges from the predecessor of the node.

- **void succ(AVLSet<FFNode>& set)**

  Adds the successors of the node to **set**.

- **void pred(AVLSet<FFNode>& set)**

  Adds the predecessor of the node to **set**.

- **void setName(const String& name)**

  Sets the name of the node to **name**.

- **const String& getName() const**

  Returns the name of the node.

- **void hideGraphic()**

  Hides the node from a graphic context.

- **void showGraphic()**

  Shows a node on the graphic context that has been hidden before.

- **FFGraphicNode *getGraphicNode()**
  **FFGraphicNode *operator()()**
  **FFGraphicNode *operator->()**

  Returns the **FFGraphicNode** label, if the node is labeled with **FFGraphicNode**, or **NULL** if it is not.

### 9.1.4 FFEdgeClass

The class **FFEdgeClass** provides methods to manipulate edges. There is no public constructor for the class **FFEdgeClass**. If you want a new Edge use the method **FFGraphClass::addEdge(...)**. **FFEdge** is a typedef to **FFEdgeClass ***. The class **FFEdgeClass** provides the following methods:

- **FFEdge copy()**

  Returns a copy of the edge in a copy of the graph.

- **void setValue(int k)**

  Sets the value of the edge to k. Counterpart to **getValue()**. As anyone can use this method, values are only very locally valid. **Labels must not use this method during init, parse or unparse**.

- **int getValue()**

  Returns the value of the edge. Counterpart to **setValue(...)**.

- **void setWeight (FFCoord x)**

  Sets the weight of the edge to **x**.

- **FFCoord getWeight()**

  Returns the weight of the edge.

- **void setName(const String& name)**

  Sets the name of the edge to **name**.

- **const String& getName()**

  Returns the name of the edge.

- **FFGraph getGraph()**

  Returns the graph the edge is in.

- **FFNode source()**

  Returns the source node of the edge.

- **FFNode target()**

  Returns the target node of the edge.

- **void revert()**

  Reverts the edge.

- **void setTarget(FFNode n)**

  Sets the target of the edge to **n**. **n** must be a node of the same graph as the edge.

- **void setSource(FFNode n)**

  Sets the target of the edge to **n**. **n** must be a node of the same graph as the edge.

- **void assignLabel(FFEdgeLabel *label, int i=-1)**

  Assigns **label** to the edge label with index **i** using the method **void FFLabelClass::assign(FFLabelClass *)**. If **i == -1** the edge checks if it is labeled with a label of the same type as **label** and assigns **label** to it if it finds one.

- **void setLabel(FFEdgeLabel *l, int i=-1, BOOL doFree=YES)**

  Sets the edge label of index **index** to **label**. If **doFree == YES** the old label is freed using the **delete** operator. If **index == -1** the edge checks, if it is labeled with an edge label of the same type as **label** and sets it to **label** if it finds one.

- **void assignLabel(MPlex<FFEdgeLabel *>& set)**

  For all edge labels in **set** the edge checks if it is labeled with a label of the same type as the label from **set** and assigns the label from **set** to it if it finds one.

- **void setLabel(MPlex<FFEdgeLabel *>& set, BOOL doFree=YES)**

  For all edge labels in **set**, the edge checks if it is labeled with a label of the same type as the label from **set** and sets it to the label from **set** if it finds one. If **doFree == YES** the old label is freed using the **delete** operator.

- **int getLabelIndex(const String& type, AVLSet<int> *set=NULL) const**

  If **set == NULL** the index of the edge label with type **type** is returned, or -1 if none is found. if **set != NULL** the indexes of all edge labels which contain **type** in their type are added to **set**. The index of the first label that is found is returned, or -1, if none is found.

- **FFEdgeLabel *getLabel(int index)**

  Returns the edge label with index **index**, or **NULL** if the edge has no label with index **index**.

- **MPlex<FFEdgeLabel *> *getLabels()**

  Returns a MPlex containing all labels of the edge.

- **void show()**

  Prints the edge to stdout.

- **void hideGraphic()**

  Hides the edge from a graphic context.

- **void showGraphic()**

  Shows a previously hidden edge in the graphic context

- **FFGraphicEdge *getGraphicEdge()**
  **FFGraphicEdge *operator()()**
  **FFGraphicEdge *operator−>()**

  Returns the **FFGraphicEdge** label if the edge is labeled with **FFGraphicEdge** or **NULL**, if it is not.

## 9.2 Label classes

### 9.2.1 FFLabelClass

The class **FFLabelClass** is the superclass of all labels and defines some methods that all labels must provide. For details on what the methods should do in a label, see section 5.2 on page 11. The methods **FFLabelClass** defines are:

- **void _setType(const String& s)**

  Appends ":" followed by **s** to the type. Should be called in the constructor of each label.

- **virtual BOOL parse(const String& str)**

  Does nothing. Returns **YES**.

- **virtual String unparse()**

  Does nothing. Returns "".

- **virtual const String& type() const**

  Returns a reference to the type of the label.

- **virtual String baseType() const**

  Returns the part after the last ":" in the type of the label.

- **virtual BOOL isSaveable() const**

  Returns **NO**.

- **virtual BOOL shouldCopy() const**

  Returns **YES**.

- **virtual BOOL isInspectable() const**

  Returns **NO**.

- **virtual BOOL compare(const FFLabelClass*) const**

  Returns **NO**.

- **virtual void event(labelEvent, labelEventStruct *)**

  Does nothing.

- **virtual void assign(const FFLabelClass*)**

  Does nothing.

### 9.2.2 FFGraphLabel

The class **FFGraphLabel** is a subclass of **FFLabelClass**. **FFGraphLabel** is the superclass for all graph labels. It defines the following methods:

- **FFGraphLabel()**

  Appends "FFGraphLabel" to the type.

- **virtual void init(const FFGraph& g)**

  Tells the label that it is bound to graph **g**. The member variable **graph** is set to **g**.

- **virtual FFGraphLabel *proto() const = NULL**

  This method must be implemented by subclasses to reproduce itself.

- **virtual BOOL hasOwnInspector() const**

  Returns **YES** if the member variable **inspector != NULL**.

- **virtual void showInspector()**

  If the graph label has an inspector, the methods **FFInspector::show()** and **FFInspector::inspect()** are invoked.

- **virtual void updateInspector()**

  If the graph label has an inspector, the method **FFInspector::inspect()** is invoked.

- **virtual FFGraph getGraph()**

  Returns the member variable **graph** which usually contains the graph, the label is bound to.

- **virtual F̃FGraphLabel()**

  If the label has an inspector, the inspector is freed.

### 9.2.3 FFNodeLabel

The class **FFNodeLabel** is a subclass of **FFLabelClass**. **FFNodeLabel** is the superclass for all node labels. It defines the following methods:

- **FFNodeLabel()**

  Appends ":" followed by "FFNodeLabel" to the type.

- **virtual void init( FFNodeClass *n)**

  Sets the member variable **node** to **n**. This method is used by the library to tell a label to which node it is bound.

- **virtual FFNodeLabel *proto() const = NULL**

  Must be implemented by subclasses to reproduce themselves.

- **virtual F̃FNodeLabel()**

  Does nothing.

### 9.2.4 FFEdgeLabel

The class **FFEdgeLabel** is a subclass of **FFLabelClass**. **FFEdgeLabel** is the superclass for all edge labels. It defines the following methods:

- **FFEdgeLabel()**

  Appends ":" followed by "FFEdgeLabel" to the type of the label.

- **virtual void init( FFEdgeClass *e)**

  Sets the member variable **edge** to **e**. This method is used by the library to tell it to which edge it is bound.

- **virtual FFEdgeLabel *proto() const = NULL**

  Must be implemented by subclasses to reproduce themselves.

- **virtual F̃FEdgeLabel()**

  Does nothing.

### 9.2.5 FFSubGraphLabel

The class **FFSubGraphLabel** is a subclass of **FFNodeLabel**. **FFSubGraphLabel** implements nested graphs. This means a **FFSubGraphLabel** contains a graph itself. The classes **FFTclWindow** and **FFTclSubWindow** support the display of nested graphs.

- **virtual BOOL parse(const String& s)**

  **s** should contain the file name of a sgraph file. The graph from the file is load into the graph the label contains.

- **virtual String unparse()**

  Returns the filename of the graph which the label contains.

- **virtual BOOL isInspectable() const**

  Returns YES.

- **FFSubGraphLabel()**

  Adds ":" followed by "FFSubGraphLabel" to the type of the label.

- **virtual BOOL isSaveable() const**

  Returns YES.

### 9.2.6 FFLayoutLabel

The class **FFLayout** is a subclass of **FFGraphLabel** and is the superclass for all labels that do any layout in the distribution. The class does not actually do anything, but is designed to define a common interface for all layout labels. See section 8.2 on page 20 for more information. The class defines the following methods:

- **FFLayoutLabel()**

  Adds ":" followed by "FFLayoutLabel" to the type of the label.

- **virtual void layout(AVLSet <FFNodeClass *>&)**

  Does nothing.

- **virtual void layout()**

  Does nothing.

### 9.2.7 FFSugi

The class **FFSugi** is a subclass of **FFLayoutLabel** and implements a Sugiyama layout algorithm. The class defines the following methods:

- **FFSugi ()**

  Creates a new instance of the **FFSugi** class.

- **virtual void init(const FFGraph& g)**

  Used internally by the library to initialize the label.

- **virtual BOOL isInspectable() const**

  Returns YES.

- **virtual FFGraphLabel* proto() const**

  Returns a new new instance of **FFSugi**.

- **BOOL getUseStartNode ()**
  Returns whether start-node information is used for the layout.
- **void setUseStartNode (BOOL flag)**
  Sets whether start-node information should be used for the layout.
- **BOOL getKeepEdgePoints ()**
  Returns whether edge points are kept in a local layout.
- **void setKeepEdgePoints (BOOL)**
  Sets whether edge points should be kept during a local layout.
- **BOOL getUseGrid ()**
  Returns whether a grid should be used to position nodes and edges.
- **void setUseGrid (BOOL)**
  Sets whether a grid should be used to position nodes and edges.
- **BOOL getLeavesDown ()**
  Returns whether leaves should be put on the lowest level.
- **void setLeavesDown (BOOL)**
  Sets whether leaves should be put on the lowest level.
- **int getCrossingAlg ()**
  Returns what algorithm is used to reduce crossings.
- **BOOL setCrossingAlg (int alg)**
  Specifies that algorithm **alg** should be used to reduce crossings.
- **int getIteration1 ()**
  Returns the number of iterations for the reduce crossings algorithm.
- **BOOL setIteration1 (int n)**
  Sets the number of iterations for the reduce crossings algorithm to **n**.
- **FFCoord getxoff ()**
  Returns the lowest x coordinate the graph will have after a layout.
- **BOOL setxoff (FFCoord x)**
  Sets the lowest x-coordinate the graph may have after a layout to **x**.
- **FFCoord getyoff ()**
  Returns the lowest y coordinate the graph will have after a layout.
- **BOOL setyoff (FFCoord y)**
  Sets the lowest y-coordinate the graph may have after a layout to **y**.
- **FFCoord getzoff ()**
  Returns the lowest z coordinate the graph will have after a layout.
- **BOOL setzoff (FFCoord z)**
  Sets the lowest z-coordinate the graph may have after a layout to **z**.

- **FFCoord getxDistance ()**
  Returns the distance the nodes will have on the x-coordinate after a layout.
- **BOOL setxDistance (FFCoord x)**
  Sets the distance the nodes will have on the x-coordinate after a layout to **x**.
- **FFCoord getyDistance ()**
  Returns the distance the nodes will have on the y-coordinate after a layout.
- **BOOL setyDistance (FFCoord)**
  Sets the distance the nodes will have on the y-coordinate after a layout to **y**.
- **FFCoord getzDistance ()**
  Returns the distance the nodes will have on the z-coordinate after a layout.
- **BOOL setzDistance (FFCoord z)**
  Sets the distance the nodes will have on the z-coordinate after a layout to **z**.
- **FFCoord getDummyOffset ()**
  Returns the percentage of the distance of edge points relative to the distance of nodes.
- **void setDummyOffset (FFCoord p)**
  Sets the percentage of the distance of edge points relative to the distance of nodes to **p**.
- **int getDimension ()**
  Returns the number of dimensions the graph should be layouted in. May return 2 or 3.
- **void setDimension (int i)**
  Sets the number of dimensions the graph should be layouted to **i**. **i** must be 2 or 3.
- **void layout (AVLSet <FFNodeClass *>& set)**
  Does a local layout on the subgraph defined by the nodes in **set** and the edges between the nodes in the set.
- **void layout ()**
  Does a sugiyama layout on the graph.

### 9.2.8 FFSpring

The class **FFSpring** is a subclass of **FFLayoutLabel** and implements a spring-embedder layout algorithm. The class defines the following methods:

- **FFSpring ()**
  Creates a new instance of the **FFSpring** class.
- **virtual void init(const FFGraph& g)**
  Used internally by the library to initialize the label.
- **virtual FFGraphLabel* proto() const**
  Returns a new new instance of **FFSpring**.
- **virtual BOOL isInspectable() const**
  Returns YES.

- **BOOL getAnimation ()**
  Returns whether the layout should be animated.
- **void setAnimation (BOOL)**
  Sets if the layout should be animated.
- **BOOL getUseEdgeWeight ()**
  Returns whether the weight of edges should be used.
- **void setUseEdgeWeight (BOOL)**
  Sets whether the weight of edges should be used.
- **int getMaxIteration ()**
  Returns the maximum number of iterations.
- **BOOL setMaxIteration (int n)**
  Sets the maximum number of iterations to **n**.
- **int getAnimationIntervals ()**
  Returns the number of iterations before displaying a result.
- **BOOL setAnimationIntervals (int n)**
  Sets the number of iterations before displaying a result to **n**.
- **FFCoord getOptDistance ()**
  Returns the optimal distance between two nodes.
- **BOOL setOptDistance (FFCoord x)**
  Sets the optimal distance between two nodes to **x**. Returns **YES**, if x is a valid distance.
- **FFCoord getxoff ()**
  Returns the lowest possible x-coordinate.
- **BOOL setxoff (FFCoord x)**
  Sets the lowest possible x-coordinate to **x**.
- **FFCoord getyoff ()**
  Returns the lowest possible y-coordinate.
- **BOOL setyoff (FFCoord)**
  Sets the lowest possible y-coordinate to **y**.
- **FFCoord getzoff ()**
  Returns the lowest possible z-coordinate.
- **BOOL setzoff (FFCoord)**
  Sets the lowest possible z-coordinate to **z**.
- **BOOL getDimension ()**
  Returns **YES** if 3 dimensions are used and **NO** if 2 dimensions are used.
- **void setDimension (BOOL flag)**
  If **flag == YES** 3 dimensions are used for layout. If **flag == NO**, 2 dimensions are used.

- **void layout (AVLSet<FFNodeClass *>& set)**
  Does a local layout for the subgraph defined by the nodes in **set** and the edges between nodes in **set**.
- **void layout ()**
  Does a layout of the whole graph.

## 9.3 Graphic classes

### 9.3.1 FFWindow

The class FFWindow is a general class that can execute tcl/tk scripts. It defines the following methods:

- **FFWindow()**
  Creates a new instance of **FFWindow**.
- **int mapWindow()**
  Maps the window on the screen.
- **int unmapWindow()**
  Unmaps the window from the screen.
- **int withdraw()**
  Withdraws the window from the screen.
- **int iconify()**
  Iconifies the Window.
- **int deiconify()**
  Deiconifies the Window.
- **virtual void update()**
  Does a graphic update on the window.
- **void setDisplay(const String& disp)**
  Sets the display where the window should appear to **disp**. This method must be called before **initFromFile(...)** to have any effect.
  void structureProc(XEvent);
- **void initFromFile(const String& file, int argc=0, char **argv=NULL)**
  Executes the tcl files **file**. **argc** and **argv** are used as if they were given to the wish shell.
- **virtual int setTitle(const String& title)**
  Sets the title of the window to **title**.

### 9.3.2 FFGraphWindow

The class **FFGraphWindow** is a subclass of **FFGraphLabel** and is the superclass of all window labels. Window labels are used to display graphs. Usually you want to display a graph in a window on a screen. But you can also subclass **FFGraphWindow** to print the graph on a printer or export it as an image or whatever.
**FFGraphWindow** is a virtual class, so no instance of this class can be created. **FFGraphWindow** defines the following methods:

- **FFGraphWindow()**

  Adds ":" followed by "FFGraphWindow" to the type of the label.

- **void scaleXYZ(FFCoord dx, FFCoord dy, FFCoord dz) = NULL**

  Should scale the coordinate system by **dx** in x direction, by **dy** in y direction and by **dz** in z direction.

- **void rotateX(FFCoord dx) = NULL**

  Should rotate the coordinate system by **dx** degrees around the x axis.

- **void rotateY(FFCoord dy) = NULL**

  Should rotate the coordinate system by **dy** degrees around the y axis.

- **void rotateZ(FFCoord dz) = NULL**

  Should rotate the coordinate system by **dz** degrees around the z axis.

- **void translate(FFCoord dx, FFCoord dy, FFCoord dz) = NULL**

  Should translate the coordinate system by **dx** in x direction, by **dy** in y direction and by **dz** in z direction.

- **FFPoint proj(const FFPoint& A) = NULL**

  Should convert the graph coordinate **A** to the output device coordinate system. The third coordinate of the result should be usable by hidden-line algorithms.

- **FFPoint unProj(const FFPoint& A) = NULL**

  Should convert the output device coordinate **A** back to the graph coordinate system.

- **FFPoint adjust(FFGraphicNode *A, FFPoint B) = NULL**

  Should return where an edge to **A** with **B** being the last point of the edge before **A** should end. E.g. if you display nodes as circles you might want the edge to stop at the outskirts instead of going to the center of the circle.

- **void setLabelOffset(int dx, int dy) = NULL**

  This method sets the relative position of the name of a node to the position of the node.

- **void adjustWindowSize() = NULL**

  Should resize the window to display the whole graph.

- **String addNode(FFGraphicNode *n) = NULL**

  Should add a graphic representation of node **n**. The return value must be a unique String by which the graphic representation can be addressed.

- **String addEdge(FFGraphicEdge *e) = NULL**

  Should add a graphic representation of edge **e**. The return value must be a unique String by which the graphic representation can be addressed.

- **String addLabel(FFGraphicNode *n) = NULL**

  Should add a graphic representation for the name of node **n**. The return value must be a unique String by which the graphic representation can be addressed.

- **String addLabel(FFGraphicEdge *e) = NULL**

  Should add a graphic representation for the name of edge **e**. The return value must be a unique String by which the graphic representation can be addressed.

- **void changeColor(FFGraphicNode *n, String& color) = NULL**
  Should change the color of node **n** to **color**.
- **String changeToIcon(FFGraphicNode *n, String& icon) = NULL**
  Should change the representation of node **n** to be displayed as the icon that is stored in file **icon**.
- **String changeToCircle(FFGraphicNode *n) = NULL**
  Should change the representation of node **n** to be displayed as a graphical item like circles, boxes etc.
- **String changeToWindow(FFGraphicNode *n) = NULL**
  This method is used to display nested graphs. The graphic representation of **n** should change to something that can display a graph itself.
- **void changeMask(FFGraphicNode *n, String& mask) = NULL**
  If node **n** is displayed as an icon, the icon should be masked using the bitmap that is stored in file **mask**.
- **void changeColor(FFGraphicEdge *e, String& color) = NULL**
  The color of edge **e** should change to **color**.
- **void changeLabel(FFGraphicNode *n, const String& name) = NULL**
  The name of node **n** should be changed to **name** in the window.
- **void changeLabel(FFGraphicEdge *e, const String& name) = NULL**
  The name of edge **e** should be changed to **name** in the window.
- **FFPoint getRange(const FFGraphicNode *n) = NULL**
  Returns a point containing (width,height,depth) of the bounding box of the node **n**.
- **void setSpline(FFGraphicEdge *e, BOOL spline) = NULL**
  Sets whether edge **e** should be displayed as a spline (**spline == YES**) or not (**spline == NO**).
- **void remove(FFGraphicNode *n) = NULL**
  Should remove the graphic representation of node **n** from the screen.
- **void remove(FFGraphicEdge *e) = NULL**
  Should remove the graphic representation of edge **e** from the screen.
- **void removeLabel(FFGraphicNode *n) = NULL**
  Should remove the graphic representation of the name of node **n** from the screen.
- **void removeLabel(FFGraphicEdge *e) = NULL**
  Should remove the graphic representation of the name of edge **e** from the screen.
- **void updateGraph() = NULL**
  Should redraw the whole graph.
- **void updateGraph(FFGraphicNode *n) = NULL**
  Should redraw node **n**.
- **void updateGraph(FFGraphicEdge *e) = NULL**
  Should redraw edge **e**.

- **void setCurrent(FFGraphicNode *n) = NULL**
  Should unselect all items and mark node **n** as selected.
- **void setCurrent(FFGraphicEdge *e) = NULL**
  Should unselect all items and mark edge **e** as selected.
- **void addToCurrent(FFGraphicNode *n) = NULL**
  Should unselect any selected edges and mark node **n** as selected.
- **void selectIfNot(FFGraphicNode *n) = NULL**
  Should unselect any selected edges. If node **n** is selected unselect it, if it is not selected select it.
- **void unsetCurrent() = NULL**
  Should unselect all selected items.
- **AVLSet<FFGraphicNode *>& selectedNodes() = NULL**
  Should return an AVLSet containing all selected nodes.
- **BOOL isCurrent(FFGraphicNode *n) = NULL**
  Should return whether node **n** is selected or not.
- **void unset(FFGraphicNode *n) = NULL**
  Should unselect node **n**.
- **void unset(FFGraphicEdge *e) = NULL**
  Should unselect edge **e**.
- **void updateInspector() = NULL**
  Should update the inspector, if the window has one.
- **void sizeTo(int x, int y) = NULL**
  Should change the size of the window to (**x**, **y**).
- **void setUpdate(BOOL f) = NULL**
  Sets whether window updates shall be made.
- **BOOL getUpdate() = NULL**
  Returns whether window updates are made.
- **void update() = NULL**
  Forces a window update.
- **int setTitle(const String& s) = NULL**
  Set the title of the window to **s**.
- **void setBusy() = NULL**
  Should in some way indicate that the window is busy, e.g. by changing the mouse cursor.
- **void unsetBusy() = NULL**
  Should somehow indicate that the window is not busy, e.g. by changing the mouse cursor.
- **int mapWindow() = NULL**
  Should map the window to the display.

- **int unmapWindow() = NULL**

  Should unmap the window from the display.

- **int withdraw() = NULL**

  Should withdraw the window form the display.

- **int iconify() = NULL**

  Should iconify the window.

- **int deiconify() = NULL**

  Should deiconify the window.

### 9.3.3 FFGraphicNode

As nodes do not contain graphic information, when **FFGraphClass::createGraphic(...)** is invoked the graph labels itself with the node label **FFGraphicNode**. This label contains all the graphic information that is needed to display a node. **FFGraphicNode** provides the following methods:

- **FFGraphicNode()**

  Creates a new instance of **FFGraphicNode**. You should never have to call this method, as the graph creates this label itself if it is needed.

- **virtual void init(FFNode n)**

  This method is called internally by the graph to initialize the label after it has been bound to a node. The label checks if the graph has a window label and displays itself in the window label if it finds one using the method **FFGraphWindow::addNode(...)**.

- **virtual void event(labelEvent type, labelEventStruct *)**

  Whenever there is a structural change to the graph this method is invoked. This method responds to the following events:

  - **NAME_CHANGE**

    The name on the screen is changed according to the new name of the node

  - **ADD_GRAPHLABEL**

    Checks if the new graph label is a window label and if yes, it displays itself in the new window. As only one window label may be assigned to a graph, it is assured that the node is not already displayed in a window.

  - **DEL_GRAPHLABEL**

    If the graph label that is going to be deleted is the window label the node is displayed in, the node is removed from the window.

- **inline FFCoord getx() const**

  Returns the x coordinate of the node.

- **inline FFCoord gety() const**

  Returns the y coordinate of the node.

- **inline FFCoord getz() const**

  Returns the z coordinate of the node.

- **inline const FFPoint& getCoords() const**

  Returns a reference to the coordinates of the node. To change the coordinates use one of the **moveTo(...)** or **rmoveTo** methods.

- **inline FFCoord getWidth()**

  Returns the width of the label.

- **inline FFCoord getHeight()**

  Returns the height of the label.

- **inline FFCoord getDepth()**

  Returns the depth of the label.

- **inline FFPoint& getRange()**

  Returns the range of the label.

- **void setWidth(FFCoord w)**

  Sets the width of the node to **w**.

- **void setHeight(FFCoord h)**

  Sets the width of the node to **h**.

- **void setDepth(FFCoord d)**

  Sets the width of the node to **d**.

- **inline void setRange(const FFPoint& p)**

  Sets the range of the node to **p**.

- **void changeToCircle()**

  Tells the node to display itself as a graphical item, like a circle or a box etc. . This method calls the method **FFGraphWindow::changeToCircle(...)** to change the node. Which graphical item is used depends on the implementation of the window label.

- **void changeToWindow()**

  Tells the node to display itself as a window that may contain a graph. This method calls the method **FFGraphWindow::changeToWindow(...)**. The actual implementation of how to display the window is left to the window label.

- **const String& getNodeWindow() const**

  Returns the id of the window the node is displayed as, or ”” if the node is not displayed as a window.

- **void changeToIcon(const String& iconFile)**

  Tells the node to display itself as the icon that is defined in **iconFile**. This method calls the method **FFGraphWindow::changeToIcon(...)**. The actual implementation of how to display the icon is left to the window label.

- **String& getIcon()**

  Returns the id of the icon the node is displayed as, or ”” if the node is not displayed as an icon.

- **BOOL isScalable()**

  If the coordinate system of the window label is scaled, this method returns if the graphical item the node is displayed as should be affected by this. The method returns **NO** for icons and window and **YES** for other items.

- **BOOL isCircle()**

  Return whether the node is displayed as a graphical item.

- **void setMask(const String& maskFile)**

  Tells the node to be masked according to the mask defined in **maskFile**. This method calls the method **FFGraphWindow::setMask(...)**.

- **String& getMask()**

  Returns the id of the mask or ”” if the node is not masked.

- **const String& getNodeWindow() const**

  Returns the id of the window the node is displayed as, or ”” if the node is not displayed as a window.

- **const String& getId() const**

  Returns the id of the graphical representation of the node.

- **const String& getLabelId() const**

  Returns the id of the graphical representation of the name of the node.

- **void setName(const String& s)**

  Sets the graphical representation of the name of the node to **s**.

- **void setColor(const String& c)**

  Sets the color of the node to **c**. **c** must be a color-name your X11 server knows about (see your local rgb.txt), or specified in the format #rrggbb.

- **String getColor()**

  Returns the color of the node.

- **void setWindow(FFGraphWindow *w)**

  Displays the node in the window label . You can use the special window label **NULL** to hide a node.

- **void moveTo(char *x, char *y, char *z=”0”)**

  Tells the node to move to the coordinates (**x**,**y**, **z**).

- **void moveTo(const FFPoint& p)**

  Tells the node to move to the coordinate **p**.

- **void rmoveTo(char *x, char *y, char *z=”0”)**

  Tells the node to add (**x**,**y**,**z**) to its current position.

- **void rmoveTo(const FFPoint& p)**

  Tells the node to add **p** to its current position.

- **void updateGraphic()**

  Tells the node to redisplay itself.

- **inline FFNodeClass *getNode()**
  **inline FFNodeClass *operator()()**
  **inline FFNodeClass *operator->()**

  Returns the node the graphic node is bound to.

### 9.3.4 FFGraphicEdge

As edges do not contain graphic information, when **FFGraphClass::createGraphic(...)** is invoked the graph labels itself with the edge label **FFGraphicEdge**. This label contains all the graphic information that is needed to display an edge. **FFGraphicEdge** provides the following methods:

- **virtual void event(labelEvent type, labelEventStruct *)**

  Whenever there is a structural change to the graph, this method is called. **type** specifies the type of event that occurred. The method responds to the following methods:

  - **NAME_CHANGE**
    Changes the name of the edge according to the new name.
  - **SOURCE_CHANGE**
    Redisplays the edge with the new source node.
  - **TARGET_CHANGE**
    Redisplays the edge with the new target node.
  - **REV_EDGE**
    Redisplays the edge with the new direction.
  - **ADD_GRAPHLABEL**
    Checks whether the new label is a graph window and if it is, the edge displays itself in the new window label. As there can only be one window label bound to a graph it is assured that the edge is not already displayed.
  - **DEL_GRAPHLABEL**
    Checks whether the label that will be removed is a window label and if it is the edge removes itself from the window.

- **FFCoord dist(const FFPoint& p1, const FFPoint& p2)**

  Returns the distance between point **p1** and **p2**.

- **FFCoord dist(const FFPoint& P, const FFPoint& A, const FFPoint& B)**

  Returns the distance between point **P** and the line **A** and **B**.

- **inline FFCoord getxAt(int p)**

  Returns the x coordinate of the edge point with number **p**

- **inline FFCoord getyAt(int p)**

  Returns the y coordinate of the edge point with number **p**

- **inline FFCoord getzAt(int p)**

  Returns the z coordinate of the edge point with number **p**

- **int getPointNear(const FFPoint& p)**

  Returns the edge point that is nearest to point **p**.

- **BOOL addPointAt(int k, const FFPoint& p)**

  Add an edge point as number **k** and position **p**. The number of all following edge points is increased by one.

- **BOOL deletePointAt(int k)**

  Deletes the edge point with number **k**.

- **BOOL deletePointAt(const FFPoint& p)**
  Deletes the point that is nearest to point **p**.
- **int anzPoints()**
  Returns the number of edge points.
- **FFPoint& getPointAt(int k)**
  Returns the point with number **k**.
- **const MPlex<FFPoint>& getEdgePoints()**
  Returns a MPlex containing all edge points.
- **String getId()**
  Returns the id of the graphical representation of the edge.
- **String getLabelId()**
  Returns the id of the graphical representation of the name of the edge.
- **void revert()**
  Reverts the edge in the window.
- **void setName(const String& s)**
  Sets the name of the edge in the window to **s**.
- **virtual void init(FFEdgeClass *)**
  Is called internally by the library to initialize the label. The label Checks whether a window label is bound to the graph and displays itself in the window if it find one.
- **void updateGraphic()**
  Redisplays the edge.
- **void toggleSpline()**
  Toggles whether the edge is displayed as a spline or not.
- **void setColor(const String& c)**
  Sets the color of the node to **c**.
- **String getColor()**
  Returns the color of the edge.
- **void rmoveTo(const FFPoint& p)**
  Adds **p** to the positions of all edge points.
- **void rmovePoint(int k, const FFPoint& p)**
  Adds **p** to the edge point with number **k**.
- **void movePoint(int k, const FFPoint& p)**
  Moves the edge point with number **k** to the position **P**.
- **inline FFEdgeClass *getEdge()**
  **inline FFEdgeClass *operator()()**
  **inline FFEdgeClass *operator−>()**
  Returns the edge the label is bound to.

### 9.3.5 FFInspector

Graph labels may contain inspectors to display their content. The class **FFInspector** defines a general interface for inspectors. The class defines the following methods:

- **FFInspector(FFGraphLabel *g)**

  Creates a new instance of FFInspector and initializes it according to the graph-label **g**.

- **virtual void show()**

  Should display the inspector on the screen.

- **virtual void hide()**

  Should remove the inspector from the screen.

- **virtual void inspect() = NULL**

  Should display the content of the graph-label.

### 9.3.6 FFTclInspector

**FFTclInspector** is a superclass for all inspectors that use the tcl/tk widget set to display themselves. It implements the following methods.

- **virtual void hide()**

  Calls a tk function to remove the inspector from the screen.

- **virtual void show()**

  Calls a tk function to show the inspector on the screen and top it.

### 9.3.7 FFSugiInsp

The class **FFSugiInsp** is an inspector for the FFSugi layout-label. It enables you to change the parameters of the sugyjama layout algorithm.

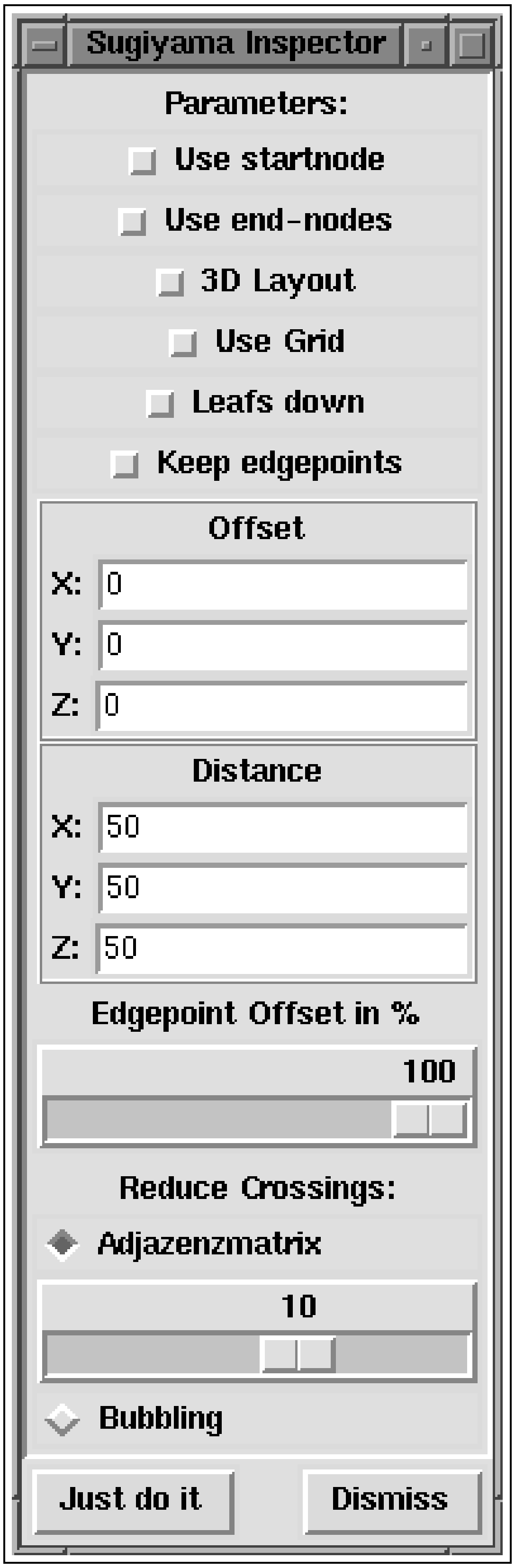


Figure 4: FFSugiInsp

It implements the following methods:

- **FFSugiInsp(FFGraphLabel *l)**
  Creates a new instance of the class **FFSugiInsp** and initializes it with the sugiyama label **l**.
- **int recall(Tcl_Interp *i, int k, char **c)**
  Is called by the tk library whenever the user changes any parameters in the window.
- **void inspect()**
  Displays the contents of the sugiyama layout label in its window.

### 9.3.8 FFSpringInsp

The class **FFSpringInsp** is an inspector for the FFSpring layout-label. It enables you to change the parameters of the spring-embedder layout algorithm.

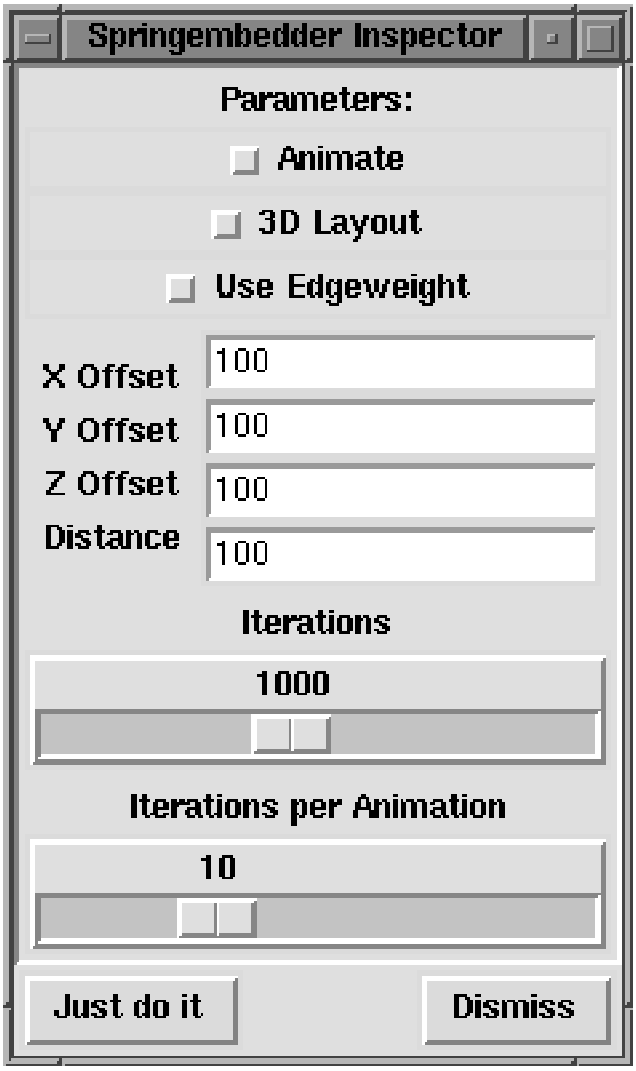


Figure 5: FFSpringInsp

It implements the following methods:

- **FFSpringInsp(FFGraphLabel *l)**

  Creates a new instance of the class **FFSpringInsp** and initializes it with the spring-embedder label **l**.

- **int recall(Tcl_Interp *i, int k, char **c)**

  Is called by the tk library whenever the user changes any parameters in the window.

- **void inspect()**

  Displays the contents of the spring-embedder layout label in its window.

#### 9.3.9 FFTclBasicWindow

The class **FFTclBasicWindow** is a subclass of **FFGraphWindow** and **FFWindow**. It is the superclass for all window-labels that use the tcl/tk widget set to display graphs. The class provides the following methods:

- **virtual void event(labelEvent type, labelEventStruct *)**

  This method is invoked whenever there is a structural change in the graph. **type** specifies which type of event occurred. The class responds to the following methods:

  – **DIRECTEDNESS_CHANGE**
    The graph is redisplayed using arrowheads at edges if the graph is directed.

- **virtual void scaleXYZ(FFCoord x, FFCoord y, FFCoord z)**

  Scales the x-coordinate of the window coordinate system **x** times, the y-coordinate **y** times and the z-coordinate **z** times.

- **virtual void rotateX(FFCoord x)**

  Scales the x-coordinate of the window coordinate system **x** times.

- **virtual void rotateY(FFCoord y)**

  Scales the y-coordinate of the window coordinate system **y** times.

- **virtual void rotateZ(FFCoord z)**

  Scales the z-coordinate of the window coordinate system **z** times.

- **virtual void translate(FFCoord x, FFCoord y, FFCoord z)**

  Translates the window coordinate system by (**x**,**y**,**z**).

- **virtual FFPoint adjust(FFGraphicNode *A, FFPoint B)**

  **B** is the last edge point of an edge before it reaches **A**.

  The result of the method is the position where the edge should meet **A**.

- **virtual BOOL hasOwnInspector() const**

  Return **NO**.

- **virtual void init(const FFGraph& g)**

  Initializes the window label with graph **g**. The layout labels **FFSugi** and **FFSpring** are added to the graph.

- **virtual void adjustWindowSize()**

  Adjusts the size of the canvas the graph is displayed in to be able to display the whole graph.

- **virtual void setLabelOffset(int x, int y)**
  Sets the relative position of the name of a node to the position of the node.
- **virtual String addNode(FFGraphicNode *n)**
  Adds a graphical representation for the node label **n** and returns a unique id.
- **virtual String addEdge(FFGraphicEdge *e)**
  Adds a graphical representation for the edge label **e** and returns a unique id.
- **virtual String addLabel(FFGraphicNode *n)**
  Adds a graphical representation for node **n**.
- **virtual String addLabel(FFGraphicEdge *e)**
  Adds a graphical representation for edge **e**.
- **virtual void changeColor(FFGraphicNode *n, String& color)**
  Changes the color of node **n** to **color**.
- **virtual String changeToIcon(FFGraphicNode *n, String& iconFile)**
  Displays node **n** as the icon defined in **iconFile**. Supported graphic types are **gif**, **ppm**, **xbm** and **xpm**. **iconFile** must be located in the according sub-directory of the icon directory you specified when installing the library. If e.g. **iconFile** is name.xxx there must be two files, ICON_LIBDIR/xxx/nameN.xxx and ICON_LIBDIR/xxx/nameS.xxx where nameN.xxx contains the icon for the node if it is not selected and nameS.xxx contains the icon for the node if it is selected.
- **virtual String changeToCircle(FFGraphicNode *n)**
  Displays node **n** as a circle.
- **virtual String changeToWindow(FFGraphicNode *n)**
  Displays node **n** as a sub-window which may display another graph.
- **virtual void changeMask(FFGraphicNode *n, String& maskFile)**
  If node **n** is displayed as an icon and the graphic type of the icon is **xpm** or **xbm**, the icon is masked according to the bitmap defined in **maskFile**. **maskFile** must be located in the xbm subdirectory of the icon-directory you specified when installing the library.
- **virtual void changeColor(FFGraphicEdge *e, String& color)**
  Changes the color of edge **e** to **color**.
- **virtual void changeLabel(FFGraphicNode *n, const String& name)**
  Changes the graphical representation of the name of node **n** to display the string **name**.
- **virtual void changeLabel(FFGraphicEdge *n, const String& name)**
  Changes the graphical representation of the name of edge **e** to display the string **name**.
- **virtual void remove(FFGraphicNode *n)**
  Removes the node **n** from the window.
- **virtual void remove(FFGraphicEdge *e)**
  Removes the edge **e** from the window.
- **virtual void removeLabel(FFGraphicNode *n)**
  Removes the graphical representation of the name of node **n** from the window.

- **virtual void removeLabel(FFGraphicEdge *e)**
  Removes the graphical representation of the name of edge **e** from the window.
- **virtual void updateGraph()**
  Redisplays the whole graph.
- **virtual void updateGraph(FFGraphicNode *n)**
  Redisplays node **n**.
- **virtual void updateGraph(FFGraphicEdge *e)**
  Redisplays edge **e**.
- **virtual void setSpline(FFGraphicEdge *e, BOOL flag)**
  If **flag == YES** edge **e** is displayed as a spline. If **flag == NO** the edge is displayed as lines.
- **virtual void setCurrent(FFGraphicNode *n)**
  Unselects any selected items and selects node **n**.
- **virtual void setCurrent(FFGraphicEdge *e)**
  Unselects all selected items and selects edge **e**.
- **virtual void addToCurrent(FFGraphicNode *n)**
  Unselects all selected edges and selects node **n**.
- **virtual void selectIfNot(FFGraphicNode *n)**
  Unselects node **n** if it is selected or call **addToCurrent(n)** if is not selected.
- **virtual void unsetCurrent()**
  Unselects any selected items.
- **virtual BOOL isCurrent(FFGraphicNode *n)**
  Returns whether node **n** is selected.
- **virtual AVLSet<FFGraphicNode *>& selectedNodes()**
  Returns a reference to the set containing all selected nodes.
- **virtual void unset(FFGraphicNode *n)**
  If node **n** is selected, it is unselected.
- **virtual void unset(FFGraphicEdge *e)**
  If edge **e** is selected, it is unselected.
- **virtual int recall(Tcl_Interp *i, int k, char **c)**
  This method is called by the tk library whenever the user does any input in the window.
- **const Tcl_Interp *getInterp()**
  Returns the tk interpreter the class uses.
- **virtual void sizeTo(int w, int h)**
  Resizes the window to width **w** and height **h**.
- **FFPoint getRange(const FFGraphicNode *n)**
  Returns the range of the bounding box of **n**.

- **virtual void setUpdate(BOOL f)**

  Sets whether graphic updates should be made in the window.

- **virtual BOOL getUpdate()**

  Returns whether graphic updates are made in the window.

- **virtual void update()**

  Forces a graphic update in the window.

- **virtual int setTitle(const String& s)**

  Sets the title of the window to **s**.

- **virtual void setBusy()**

  Changes the mouse cursor to display a watch.

- **virtual void unsetBusy()**

  Changes the mouse cursor to display an arrow,

- **virtual int mapWindow()**

  Maps the window to the screen.

- **virtual int unmapWindow()**

  Unmaps the window from the screen.

- **virtual int withdraw()**

  Withdraws the window from the screen.

- **virtual int iconify()**

  Iconifies the window.

- **virtual int deiconify()**

  Deiconifies the window.

- **String fsBox(String message= String("Select File:"), String type= String("*"), String name=String(""))**

  Displays a file-selector box. **message** contain the title of the box, **type** contains an extension filter and **name** may contain a directory or file that should be preselected.

- **FFTclBasicWindow()**

  Creates a new instance of class **FFTclBasicWindow**. You usually do not use this class directly, but a subclass of this class.

#### 9.3.10 FFTclWindow

The class **FFTclWindow** is a subclass of **FFTclBasicWindow** and is able to display a graph in a window on the screen using the tcl/tk widget library. The class contains an inspector of the class **FFTclGraphInsp** to display the content of the labels bound to the graph. **FFTclWindow** provides the following methods:

- **FFTclWindow()**

  Creates a new instance of the class **FFTclWindow**. This instance may be bound to a graph using the **FFGraphClass::createGraphic(FFGraphWIndow *)** method to display the graph on the screen.

- **virtual void init(const FFGraph& g)**

  Initializes the window with graph **g** and puts a window displaying **g** on the screen.

- **virtual void event(labelEvent type, labelEventStruct *)**

  This method is invoked from the library if there is any structural change to the graph. **type** specifies the type of change. The method responds to the following event types:

  - **NAME_CHANGE**
    Sets the title of the window according to the new name of the graph.
  - **ADD_NODELABEL**
    **DEL_NODELABEL**
    **ADD_EDGELABEL**
    **DEL_EDGELABEL**
    Updates the inspector.

- **virtual int recall(Tcl_Interp *i, int k, char **c)**

  This method is invoked from the tcl/tk library whenever the user makes any input in the window.

- **virtual int exportSelection(int , char*, int)**

  This method is invoked by the tcl/tk library. It exports the graph as a string containing the graph in sgraph format.

- **virtual int getSelection(char* str)**

  This method is invoked when the user does a copy or paste. It builds a string containing the current graph in sgraph format.

- **virtual void updateInspector()**

  Updates the inspector.

- **virtual void showInspector()**

  Brings the inspector on the screen.

#### 9.3.11 FFTclSubWindow

This class can display a graph in a node which is displayed as a window in the **FFTclWindow** window label. The class provides the following methods:

- **virtual FFGraphLabel *proto() const**

  Returns a new instance of **FFTclSubWindow**. This method is used internally by the library.

- **virtual int recall(Tcl_Interp *i, int k, char **c)**

  This method is called by the tcl/tk library whenever the user triggers an event in the sub-window.

- **virtual void init(const FFGraph& g)**

  Initializes the class with graph **g**.

- **FFTclSubWindow(FFNode dad= NULL, Tcl_Interp *i=NULL, Tk_Window win=0, String& w=String(""), String& id=String(""))**

  Creates a new instance of **FFTclSubWindow**. **dad** is the node the subgraph of this window is bound to. **i** and **win** are the tcl interpreter and the window where the subgraph should be displayed in. q**w** is the name of the canvas widget where the subgraph should be displayed in and **id** is the tk item id of the sub-graph window.

#### 9.3.12 FFPoint

The class **FFPoint** implements 3 dimensional vectors to store coordinates. The class actually contains 4 dimensions where the fourth dimension is necessary to be able to do translations with linear operators. The class defines the usual operations expected for numeric data-types. It also defines **FF_EPSILON** to 0.000000001. If the distance between two **FFPoints** is smaller than **FF_EPSILON** they are considered to be equal. You can access a single dimension by reading the member variable `v[]`. Methods without description do what you would expect them to do.

- **FFPoint()**

  Create a new instance and initializes it with (0,0,0).

- **FFPoint(const FFPoint& p)**

- **FFPoint(FFCoord nx, FFCoord ny, FFCoord nz=0.0)**

- **BOOL operator == (const FFPoint& p) const**

  Returns **YES** if the distance between the points is smaller than **FF_EPSILON**.

- **FFPoint operator-() const**

- **FFPoint operator + (const FFPoint& p) const**

- **FFCoord operator * (const FFPoint& p) const**

- **FFPoint operator * (FFCoord i) const**

- **FFPoint operator / (FFCoord i) const**

- **void operator -= (const FFPoint& p)**

- **void operator += (const FFPoint& p)**

- **void operator *= (FFCoord i)**

- **void operator /= (FFCoord i)**

- **FFPoint& operator = (const FFPoint& v)**

- **char * operator()() const**

  Returns an ASCII representation of the point.

- **FFCoord abs() const**

  Returns the absolute value of the point.

- **FFCoord abs2D() const**

  Returns the absolute value of the first two dimensions of the point.

#### 9.3.13 FFMatrix

The class **FFMatrix** implements a 3x3 Matrix. It is actually a 4x4 Matrix, but the fourth dimension is used to be able to do translations with linear operators. The class implements the methods you expect from a numeric data-type. You can access a row by reading the member variable `row[]`. Methods without description do what you would expect them to do.

- **FFMatrix()**

  Creates a 0-Matrix.

- **FFMatrix(const FFPoint&, const FFPoint&, const FFPoint&, const FFPoint&)**
- **BOOL operator == (const FFMatrix&) const**
- **FFMatrix operator-() const**
- **friend FFMatrix operator - (const FFMatrix&, const FFMatrix&)**
- **FFMatrix operator + (const FFMatrix&) const**
- **FFMatrix operator* (const FFMatrix&) const**
- **FFMatrix operator* (FFCoord) const**
- **FFPoint operator* (const FFPoint&) const**
- **FFMatrix operator / (FFCoord) const**
- **void operator -= (const FFMatrix&)**
- **void operator += (const FFMatrix&)**
- **void operator *= (FFCoord)**
- **void operator *= (const FFMatrix&)**
- **void operator /= (FFCoord)**
- **char * operator()() const**
  Returns an ASCII representation of the Matrix.

# 10 The ffgraph graph editor

A graph editor called **ffgraph** is included in the distribution to give you an example of how to use the library. You can pass a list of graphs to **ffgraph** which will be loaded and displayed.

# A The extended sgraph file format

The file format used in this library is a slightly extended version of the sgraph[Him93] file format. This means that you should be able to load all sgraph graphs, but not necessarily vice versa.
The extensions are:

- The file may contain 8-Bit data.
- The file may contain quoted " and newlines.
- The file may contain Strings which are longer than one line or 1k.
- Attributes can be in arbitrary order.

The grammar of the sgraph file format used in the library is:

⟨graph⟩ → **GRAPH IDENTIFIER =** ⟨directedness⟩ ⟨graph_internals⟩ ⟨list_of_nodes⟩ **END**
⟨directedness⟩ →
| **DIRECTED**
| **UNDIRECTED**
⟨graph_internals⟩ →
| {**$** ⟨ffGraphArgList⟩ **$**}
⟨ffGraphArgList⟩ →
| ⟨ffGraphArg⟩ ⟨ffGraphArgList⟩
⟨ffGraphArg⟩ → **FF_L IDENTIFIER IDENTIFIER**
| **FF_LO NUMBER NUMBER**
| **NP NUMBER NUMBER**
| ⟨numbers_and_strings⟩ **;**
| **UNKNOWN_INTERNAL** ⟨unknown_internal⟩
| **,**
⟨list_of_nodes⟩ →
| ⟨node⟩ ⟨list_of_nodes⟩
⟨node⟩ → ⟨node_nr⟩ ⟨node_internals⟩ ⟨nodelabel⟩ ⟨list_of_edges⟩
⟨node_nr⟩ → **NUMBER**
⟨nodelabel⟩ → **IDENTIFIER**
⟨node_internals⟩ →
| {**$** ⟨ffNodeArgList⟩ **$**}
⟨ffNodeArgList⟩ →
| ⟨ffNodeArgList⟩ ⟨ffNodeArg⟩
| ⟨ffNodeArg⟩
⟨ffNodeArg⟩ → **NUMBER NUMBER**
| **NP NUMBER NUMBER**
| **SN**
| **EN**
| **FF_3D NUMBER NUMBER NUMBER**
| **FF_C IDENTIFIER**
| **FF_L IDENTIFIER IDENTIFIER**
| **FF_I IDENTIFIER IDENTIFIER**
| **FF_I IDENTIFIER**
| **UNKNOWN_INTERNAL** ⟨unknown_internal⟩
⟨list_of_edges⟩ → **;**
| ⟨edge⟩ ⟨list_of_edges⟩
⟨edge⟩ → ⟨targetnode_nr⟩ ⟨edge_internals⟩ ⟨edgelabel⟩
⟨targetnode_nr⟩ → **NUMBER**
⟨edgelabel⟩ → **IDENTIFIER**
⟨edge_internals⟩ →
| {**$** ⟨ffEdgeArgList⟩ **$**}
⟨ffEdgeArgList⟩ →
| ⟨ffEdgeArgList⟩ ⟨ffEdgeArg⟩
| ⟨ffEdgeArg⟩
⟨ffEdgeArg⟩ → **EL** ⟨edgenumberlist⟩
| **FF_3D** ⟨ffEdgeNumberList⟩
| **FF_C IDENTIFIER**
| **FF_L IDENTIFIER IDENTIFIER**
| **UNKNOWN_INTERNAL** ⟨unknown_internal⟩
⟨edgenumberlist⟩ →
| **NUMBER NUMBER** ⟨edgenumberlist⟩
⟨ffEdgeNumberList⟩ →

| **NUMBER NUMBER NUMBER** ⟨ffEdgeNumberList⟩
⟨unknown_internal⟩ → ⟨numbers_and_strings⟩
⟨numbers_and_strings⟩ →
| ⟨number_or_string⟩ ⟨numbers_and_strings⟩
⟨number_or_string⟩ → **NUMBER**
| **IDENTIFIER**

The terminal symbols are:

| | |
|---|---|
| **NUMBER** | ::= $[0-9]^+ \mid -[0-9]^+$ |
| **IDENTIFIER** | ::= $\underline{"}$ *char* * $\underline{"}$ |
| **UNKNOWN_INTERNAL** | ::= $[A-Z]^+$ |

*char* may be any character, but " and \, which have to be escaped using \. All other terminal symbols are themselves. The graph, node and edge arguments expect the following parameters:

- **FF_LO X Y**

  Sets the relative offset of a name of a node to the position of a node to (**X**, **Y**)
- **NP X Y**

  Sets the position of a node to (**X**, **Y**).
- **SN**

  Marks the node as being a startnode.
- **EN**

  Marks the node as being an endnode.
- **FF_3D X Y Z**

  Sets the position of a node in 3 dimensions to (**X**, **Y**, **Z**).
- **FF_C COLOR**

  Sets the color of the node or edge to **COLOR**
- **FF_L TYPE CONTENT**

  Invokes the the method **BOOL FFLabelClass::parse(String)** in the label with type **TYPE**.
- **FF_I ICON** $[MASK]$

  Tells the node to display according to icon file **ICON** and mask file **MASK**.
- **EL** $[XY]^+$

  Sets the positions of the edgepoints of an edeg to **(X,Y)**$^+$
- **FF_3D** $[XYZ]^+$

  Sets the positions of the edgepoints of an edge to **(X,Y,Z)**$^+$

# B The class hierarchy

This figure shows the class hierachy of the most important classes in the library.

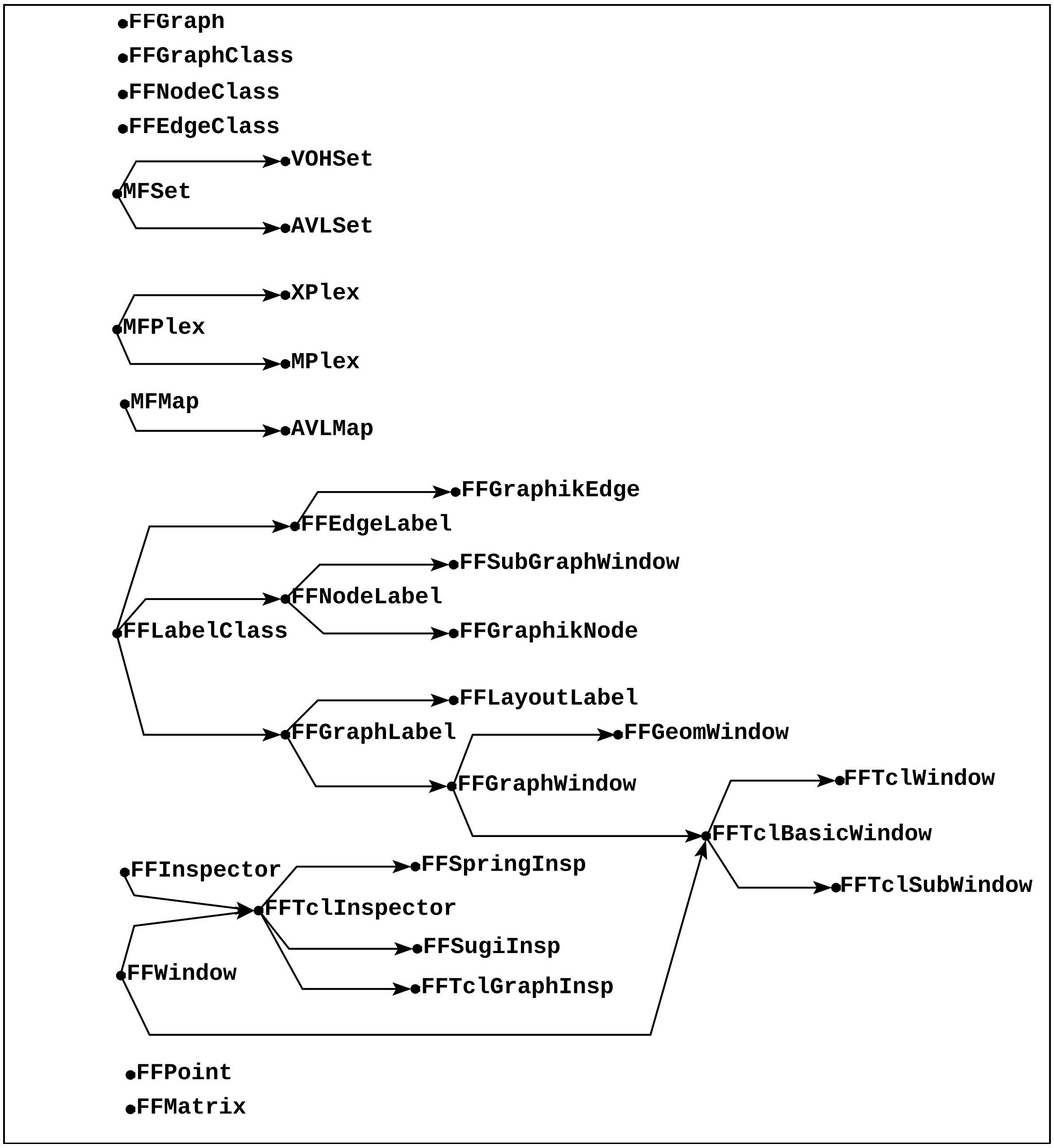


Figure 6: The class hierarchy

## Acknowledgements

I am grateful to Prof. Bernhard Steffen, Andreas Claßen, Dr. Ruth Wishart, Karlheinz Agsteiner and all of the META-Frame team for their constructive cooperation in the design and implementation of the ffgraph library.

# References

[Him90] Michael Himsolt. *GraphEd User Manual*. Universität Passau, 1990.

[Him93] Michael Himsolt. *Sgraph Programmer's Manual*. Universität Passau, 1993.

[Näh94] S. Näher. LEDA user manual (version 3.0). Technical report, Max-Plank-Institut für Informatik, Saarbrücken, 1994.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, April 1994.

[SACTM95] B. Steffen A. Claßen T. Margaria. The META-Frame: An environment for flexible tool management. In *6th Int. Conf. on Theory and Practice of Software Development (TAPSOFT'95)*, LNCS N.915, pages 791–792, Aarhus (DK), May 1995.

[San94] Georg Sander. Graph layout through the VCG tool. Technical Report A03/94, Universität des Saarlandes, Saarbrücken, Germany, October 1994.

**Technische Berichte**
**der Fakultät für Mathematik und Informatik**
**Universität Passau ab 1993**

MIP-9301 B.S. Chlebus, M. Kaufmann, J.F. Sibeyn: Deterministic Permutation Routing on Meshes.
MIP-9302 M. M"uller: Bondgraph Execution as a New Algorithm for Circuit Simulation.
MIP-9303 C. Lengauer: Loop Parallelization in the Polytope Model.
MIP-9304 M. Griebl, C. Lengauer: On the Space-Time Mapping of WHILE-Loops.
MIP-9305 V. Weispfenning: A New Approach to Quantifier Elimination for Real Algebra.
MIP-9306 A. Kemper, G. Moerkotte: Physical Object Management.
MIP-9307 M. Steinbrunn, G. Moerkotte, A. Kemper: Optimizing Join Orders.
MIP-9308 A. Kemper, G. Moerkotte, K. Peithner, M. Steinbrunn: Optimizing Disjunctive Queries in Object Bases.
MIP-9309 A. Kemper, D. Kossmann: Dual-Buffering Strategies in Object Bases.
MIP-9310 J. Knoop, O. R"uthing, B. Steffen: Optimal Code Motion: Theory and Practice.
MIP-9311 B. Freitag, T. Margaria, B. Steffen: A Pragmatic Approach to Software Synthesis.
MIP-9312 O. Burkart, B. Steffen: Pushdown Processes: Parallel Composition and Model Checking.
MIP-9313 B. Steffen, T. Margaria, B. Freitag: Module Configuration by Minimal Model Construction.
MIP-9314 A. Geser: A Solution to Zantema's Problem.
MIP-9315 S. Gorlatch, C. Lengauer: Parallelization of Divide-and-Conquer in the Bird-Meertens Formalism.
MIP-9401 U. Fraus: Inductive Theorem Proving for Algebraic Specifications TIP System User's Manual.
MIP-9402 O. Burkart, B. Steffen: Composition, Decomposition and Model Checking of Pushdown Processes.
MIP-9403 A. Geser, J. Knoop, G. L"uttgen, O. R"uthing, B. Steffen: Chaotic Fixed Point Iterations.
MIP-9404 V. Weispfenning: Parametric Linear and Quadratic Optimization by Elimination.
MIP-9405 W. Bachl, F.J. Brandenburg, T. Hickl: Hierarchical Graph Design Using HiGraD.
MIP-9406 M. Griebl, C. Lengauer: A Communication Scheme for the Distributed Execution of while Loops.
MIP-9407 A. Geser: An Improved General Path Order.
MIP-9408 S. Graf, H. Luschgy: The quantization of the Cantor measure.
MIP-9409 J. Knoop, B. Steffen, J. Vollmer: Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs.
MIP-9410 U. Lechner, C. Lengauer, M. Wirsing: An Object-Oriented Airport.
MIP-9411 T. Margaria, M. Griva, R. Tesio: Semantic Extraction for the Automatic Verification of VHDL Descriptions.
MIP-9412 M. Steinbrunn, K. Peithner, G. Moerkotte, A. Kemper: Bypassing Joins in Disjunctive Queries.
MIP-9413 J. Knoop, O. R"uthing, B. Steffen: A Tool Kit for Constructing Optimal Interprocedural Data Flow Analyses.

MIP-9414 C. Lengauer, M. Griebl: On the Parallelization of Loop Nests Containing while Loops.
MIP-9415 G. Ritter: Jackson's Theorems and the Number of Hidden Units in Neural Networks for Uniform Approximation.
MIP-9501 B. Steffen: Finite Model Checking and Beyond.
MIP-9502 K.G. Larsen, B. Steffen, C. Weise: A Constraint Oriented Proof Methodology Based on Modal Transition Systems.
MIP-9503 V. Weispfenning, R. Xue: Parametric Mixed Integer Programming by Elimination.
MIP-9504 V. Weispfenning: Solving Parametric Polynomial Equations and Inequalities by Symbolic Algorithms.
MIP-9505 S. Graf, B. Steffen, G. L"uttgen: Compositional Minimization of Finite State Systems Using Interface Specifications.
MIP-9506 F.J. Brandenburg, M. Himsolt, H.J. R"oder, K. Skodinis: Designing Data Flow Graphs by Graph Grammars.
MIP-9507 A. Eickler, C.A. Gerlhof, D. Kossmann: A Performance Evaluation of OID Mapping Techniques.
MIP-9508 S. Gorlatch: From Transformations to Methodology in Parallel Program Development: A Case Study.
MIP-9509 B. Steffen, T. Margaria, A. Claßen: Heterogeneous Analysis and Verification for Distributed Systems.
MIP-9510 B. Freitag: Method Resolution and Virtual Classes in a Deductive Object-Oriented Database.
MIP-9511 J. Knoop, B. Steffen, J. Vollmer: "Optimal Code Motion for Parallel Programs".
MIP-9512 S. Gorlatch: Constructing List Homomorphisms.
MIP-9513 K. Donner: Ersatzarithmetiken f"ur die fehlerstabile Implementation numerischer Algorithmen am Beispiel der Polynominterpolation.
MIP-9514 K. Donner: Skalierung von Matrizen und numerische Stabilit"at der Gauß-Elimination
MIP-9515 K. Donner: Orthogonale Polynome f"ur zeit"aquidistante Abtastung von Signalen
MIP-9516 B. Steffen, T. Margaria, A. Claßen, V. Braun: Incremental Formalization: a Key to Industrial Success
MIP-9517 A. Dolzmann, Th. Sturm: Simplification of Quantifier-free Formulas over Ordered Fields.
MIP-9518 M. Mendler: A Timing Refinement of Intuitionistic Proofs and its Application to the Timing Analysis of Combinational Circuits
MIP-9519 T. Margaria (Hrsg.): Programmiersprachen und Grundlagen der Programmierung