JOHANNES KEPLER
UNIVERSITY LINZ
Research and teaching network

Thomas Würthinger

Visualization of Java Control Flow Graphs

A thesis submitted in partial satisfaction of
the requirements for the degree of

Bachelor of Science
(Bakkalaureus der technischen Wissenschaften)

Supervised by:
o.Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck
Dipl.-Ing. Christian Wimmer

Institute for System Software
Johannes Kepler University Linz

Linz, October 2006

Abstract

While working on the Java HotSpot client compiler of Sun Microsystems, developers need to analyze the data structures manipulated during compilation. Until now this is done by printing text output to the command line and using the built-in debugger of the programming environment. A graphical visualization of the data can be used to improve the understanding of the compiler and to speed-up debugging.

The *Visualizer Application* presents the internal control flow graph as well as the register allocation and the intermediate representation of the compiler. It is implemented on top of the Eclipse Rich Client Platform and makes use of its plugin system. Additionally the Draw2D library is used for painting graphs. This thesis focuses on the visualization of the control flow graph and presents several different algorithms for positioning nodes and routing edges.

It is difficult to implement an optimal visualization algorithm and to measure the quality of such an algorithm. The presented algorithms give acceptable results for drawing the control flow graphs. It is described in detail how they work and also how they could be extended and further improved.

Kurzfassung

Bei der Arbeit am Java HotSpot Client Compiler von Sun Microsystems müssen Entwickler die während der Kompilierung verarbeiteten Datenstrukturen analysieren. Bisher wurde das durch die Ausgabe von Text auf die Kommandozeile und mit Hilfe des in der Entwicklungsumgebung eingebauten Debugger gemacht. Eine graphische Visualisierung der Daten kann verwendet werden, um das Verständnis des Compilers zu erhöhen und das Debuggen schneller zu machen.

Die *Visualizer Applikation* zeigt sowohl den internen Kontrollflussgraphen als auch die Registerallokation und die Zwischendarstellung an. Sie wurde auf Basis der Eclipse Rich Client Platform entwickelt und verwendet ihr Pluginsystem. Weiters wurde die Draw2D-Bibliothek zum Zeichnen von Graphen verwendet. Diese Arbeit konzentriert sich auf die Visualisierung des Kontrollflussgraphen und präsentiert verschiedene Algorithmen zur Positionierung von Knoten und zum Zeichnen von Kanten.

Es ist schwierig einen optimalen Visualisierungsalgorithmus zu entwickeln und die Qualität eines solchen Algorithmus zu messen. Die vorgestellten Algorithmen erzielen akzeptable Ergebnisse für das Zeichnen von Kontrollflussgraphen. Es ist detailliert beschrieben wie sie arbeiten und auch wie sie erweitert und weiter verbessert werden können.

Contents

Chapter 1

Introduction

Internal data structures of a compiler are difficult to analyze, especially if just a textual representation of the data is available. Displaying the data of the Java HotSpot client compiler of Sun Microsystems is the goal of the *Visualizer Application*. The task of the author of this thesis was to extend the existing visualization application developed by Christian Wimmer and to add a component for displaying control flow graphs. The application should be built on top of the Eclipse Rich Client Platform, which is described in detail in Chapter 3. The visualization should automatically arrange the nodes of a graph and should try to help the programmer to get a better understanding of the structure of compiled Java methods. Edges should be routed avoiding intersections and other properties which make the graph look bad. Additionally the layout of the graph should be manually modifiable: The user should be able to move nodes, hide edges and also combine groups of nodes. Nodes should have an automatic color depending on their properties, but manual coloring should be possible too. Two zoom levels should be implemented. One showing the details and one for the user to get an overview of large methods.

1.1 Existing Solutions

Graph visualization is a well-studied problem, however, it is difficult to measure the quality of a certain solution. The value of a visualization depends heavily on the goal one wants to achieve with it. There are a lot of graph visualization products available. The following list presents some of them:

- *uDraw:* This visualization tool can be used to edit and view directed graphs. The built-in automatic layouter positions the nodes using a variant of the hierarchical layout algorithm presented in Chapter 5.3. It was developed at the University of Bremen in Germany, the current version when writing this thesis was `3.1.1`. Nodes can also be moved manually, however, there are restrictions: All nodes need to be positioned in rows and they must not overlap. The appearance of the nodes is highly customizable. Input files have the extension `.udg`, possible export formats include GIF, TIFF, JPEG, PNG. Binaries of uDraw2006 are freely available from [uDr06], source code is however not provided.

- *aiSee* is a graph visualization product developed by AbsInt. The user can choose one of 15 different layout algorithms which are can be customized using many different parameters. Graphs can not be modified and this tool also has its own textual input file format. Possible export file formats are PNG, BMP, PPM, EPS, PS and SVG. The product is only free for non-commercial use. A 30-day evaluation version of the product is available from [aiS06].

- *Otter* is a tool especially designed for visualizing large networks. It is completely written in Java and even a Java applet running Otter is available. There are several layout algorithms implemented, which are especially useful for networks. This tool uses its own input file format with the extension `.odf`. The source code can be downloaded from [Ott06].

- *Graphviz:* The graph visualization software Graphviz comes with a large number of tools. There exist the two tools `dot` (processing directed graphs) and `neato` (processing undirected graphs) for converting input files to various graphic file formats including GIF, JPG, PS and SVG. The programs `dotty` and `lefty` can be used to view the `.dot` files directly. Some additional programs perform graph operations or measure properties of a graph. Graphs are represented in a special file format with the extension `.dot`. Source code and binaries of all tools are available under the Common Public License Version 1.0 from [Gra06]. There is also a Java graph library called Grappa, which is built on top of Graphviz.

Unfortunately every tool uses its own file format, however, all formats are human readable. Therefore it is easy to generate data and let it get displayed by one of the programs. Performing operations on the graph like moving nodes without restrictions or merging and splitting nodes is in most cases not possible.

Chapter 2

User Guide

This chapter gives a brief description of how to use the Visualizer Application to analyze the data structures of the compiler, which is part of the Java HotSpot Virtual Machine. It gives an overview about what kind of data is visualized and how a developer can take advantage of the tool. The main purpose of the program is assisting Java client compiler developers, but it can be also interesting for Java developers who want to know what is going on behind the scenes. First, some basics about the compiler are presented.

2.1 Java Client Compiler

Java source code is compiled to Java bytecodes by the Java programming language compiler. The advantage of this approach in comparison to compiling straight to machine code is platform-independence. However there is a need for an intermediate layer between Java applications and the underlying operating system: The "Virtual Machine" is responsible for executing the Java bytecodes. There exist several different implementations of Java Virtual Machines, the HotSpot VM is one of them.

2.1.1 Interpretation vs. JIT Compilation

The Virtual Machine has two choices: Either interpreting the bytecodes or compiling them before execution to machine code. Interpreting a method does not need any start-up time, however, it is a lot slower than running the compiled and optimized machine code. The just-in-time (JIT) compilation, however, needs a fixed timeslice before even the first instruction of the method is executed. Once compiled, the method is executed faster. Figure 2.1 shows a diagram comparing the two approaches.

Therefore interpretation is the best choice for methods which are not executed very often and which do not contain any loops. However, it is impossible to figure out in advance how often a method will be called, or how much loop cycles are needed. The Java HotSpot VM first interprets a method. Only when the number of calls plus the number of loop iterations within a method exceeds a certain limit, it starts the just-in-time compiler. In the common case, this gives a good performance, because only the most important methods get compiled.

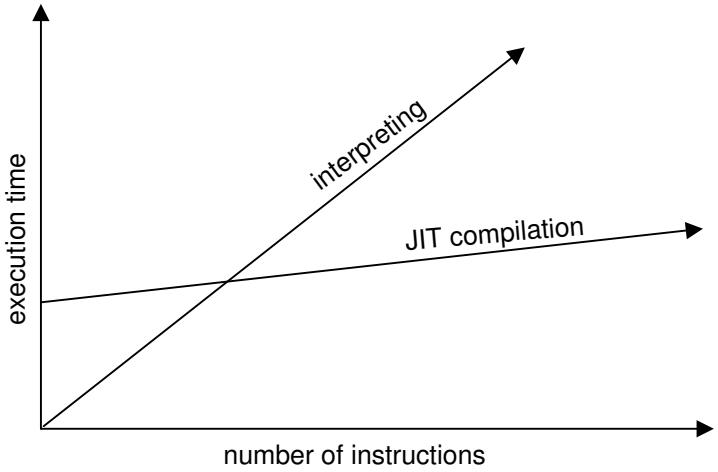

Figure 2.1: Interpretation vs. JIT compilation.

2.1.2 Client vs. Server Compiler

Two compilers are built into the Java HotSpot VM. They both perform the same task: Converting the Java bytecodes of a method to platform-specific machine code. They are however completely different. While the client compiler tries to spend only a short time on compiling a method, the goal of the server compiler is to produce as efficient machine code as possible. This behavior is also reflected in their names: The server compiler is faster for long-running server applications, while the client compiler is the better choice for short-running applications like typical client applications. For a word processor for example it would be annoying to have a long startup time, for a webserver on the other hand it would not really matter. One can think of the client compiler as an intermediate step between the server compiler and the interpreter. The server compiler can be enabled by using the flag `-server` when calling the Java executable. The Visualizer Application only shows the internal data structures of the client compiler.

2.1.3 Creating Input Files

For creating files which contain the information about the compilation, a debugging version of the Sun JDK 6 or higher is needed. A binary snapshot release can be downloaded from `http://download.java.net/jdk6/binaries/`. Whenever referring to a Java executable, the one in the `fastdebug/bin/` directory is meant.

```
public class Example {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

Listing 2.1: Java example.

In this user guide the simple "Hello world!" Java application shown in Listing 2.1 is used. It consists of a main method with a single statement in it and prints "Hello world!" on the command line. First, the Java class file containing the bytecodes needs to be created by executing `javac Example.java` on the command line which results in the file `Example.class` that contains the Java bytecodes of the class `Example`. Now the example can be started by executing `java Example`.

The command `java -XX:+PrintCFGToFile Example` not only executes the sample program, but also generates a file called `output.cfg`, which can be opened using the Visualizer Application. Figure 2.2 shows the Visualizer Application after opening `output.cfg`. It is maybe a bit surprising that instead of the method `Example.main`, the two methods `String.hashCode` and `String.charAt` are shown in the compilations window. This is because methods are not visualized if they are only interpreted and never compiled. Only four bytecodes are executed for the main method itself, so it will be just interpreted.

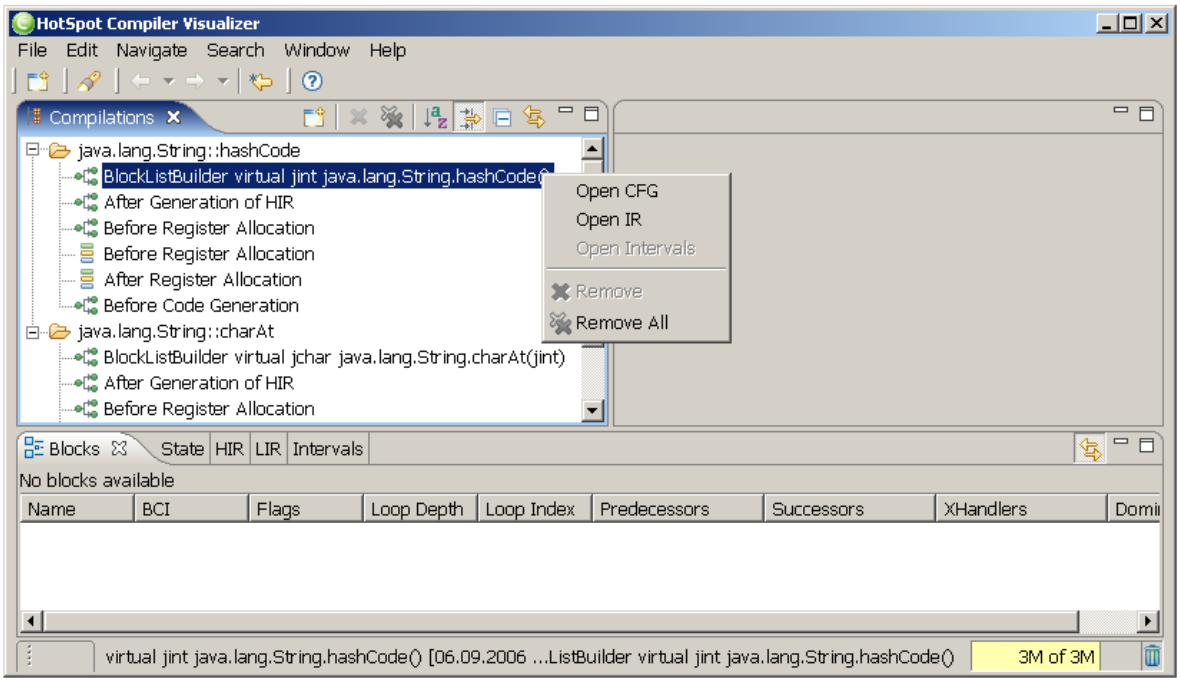

Figure 2.2: Visualizer Application after opening a sample file.

It is however possible to force that all methods get compiled. When the interpreter is disabled with the flag `-Xcomp`, the example runs noticeable slower, because more than 200 methods are compiled. This flag is often combined with a flag that tells the Java VM to compile only a specific method. Adding `-XX:CompileOnly=Example.main` to the command would ensure that all other methods are only interpreted. The flag `-XX:-Inline` can be used to disable inlining of methods.

Figure 2.2 shows that snapshots of the state of the data structures are taken at various points during compilation. Currently the following six states are produced, but this can vary as developers can easily add their own states:

- *BlockListBuilder:* This is the first state and describes the graph after the block list

builder transformed the bytecodes into blocks which are connected giving a control flow graph. Separate graphs are shown for all inlined methods.

- *After Generation of HIR:* During compilation two intermediate representations are used before the final machine code is generated. The first one is called "high level intermediate representation" (HIR) and is close to the Java bytecodes. Normally one HIR instruction is generated for one Java bytecode.
- *Before Register Allocation:* This state is output after several optimizations on the HIR. Also the "low level intermediate representation" (LIR) which is close to machine code is already generated here.
- *Before Register Allocation (Intervals):* A special view on the data structures used during register allocation.
- *After Register Allocation (Intervals):* Displays the register allocation data after physical registers are assigned to the virtual ones. It also shows which values need to be temporarily stored in memory.
- *Before Code Generation:* This last step shows the data structures just before machine code is generated.

When clicking with the right mouse button on one of the states a context menu pops up showing the possible views of this states. There exist three different views which are explained in detail in the following three chapters:

- *IR Visualizer:* Textual view of the currently available intermediate representation.
- *CFG Visualizer:* Visual representation of the control flow graph.
- *Interval Visualizer:* Register life ranges and assigned registers if available.

2.1.4 IR Visualizer

The intermediate representation visualizer displays the HIR and/or the LIR in textual form. It can be viewed as a command line output enhanced with syntax coloring and navigation. Figure 2.3 shows the IR Visualizer on the state "Before Register Allocation" of the method `String.hashCode`. At this state both, HIR and LIR are available.

There is a section for each node of the control flow graph. The first line for each node contains information about the connection to other nodes as well as the flags set for this node. All this information is graphically shown when the CFG Visualizer, which is presented in Chapter 2.1.6, is used. There are up to three subsections showing the state, the LIR and the HIR of a node.

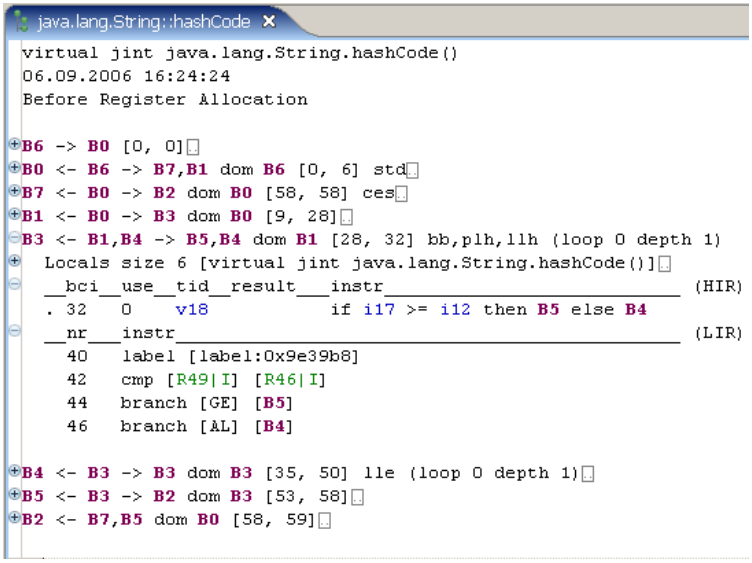

Figure 2.3: Textual view of the intermediate representations.

2.1.5 Interval Visualizer

This view displays the data structures used during register allocation. The client compiler uses the linear scan algorithm. The control flow graph is flattened to a list and the node names are painted on the x-axis. All LIR instructions are numbered serially using the block order and every column corresponds to one LIR instruction. A row with a graphical view of the lifetime interval is painted for every value. Initially virtual registers are assigned to all values. During register allocation, intervals can be split into several shorter ones. A new virtual register is assigned to each newly created interval. Additionally a physical register is assigned to every interval. If there is no register available, the value needs to be stored temporarily in memory.

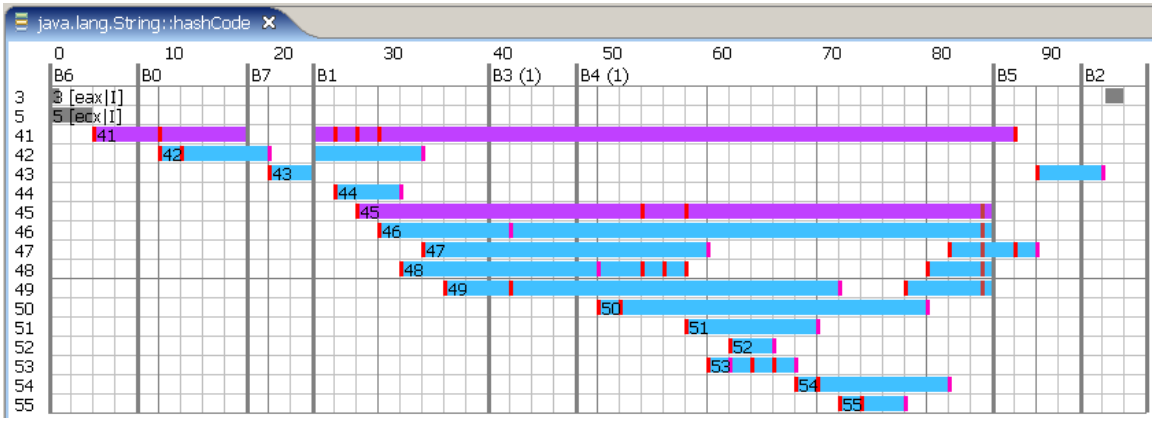

Figure 2.4: Intervals before register allocation.

Figure 2.4 shows the interval view before register allocation. No interval has been split yet, so every line contains one interval. The color of a bar indicates the type of its

associated value, e.g. magenta for objects and blue for integer values.

Figure 2.5 shows the interval view for the same method, but after register allocation. Interval 41 was split into the three parts 41, 57 and 56. Physical registers like `eax` or `ebx` are assigned to the intervals. Orange color is used to indicate that a value needs to be stored in memory for a while, which is also called "spilling". This leads to bad performance and should be avoided whenever possible. In this example, value 41 is not accessed within the loop (which consists of the blocks `B3` and `B4`, see Figure 2.6 for a full control flow graph visualization of the method), therefore it is a good decision to spill this value, as it needs to be loaded from memory only once after the end of the loop and not at every loop iteration.

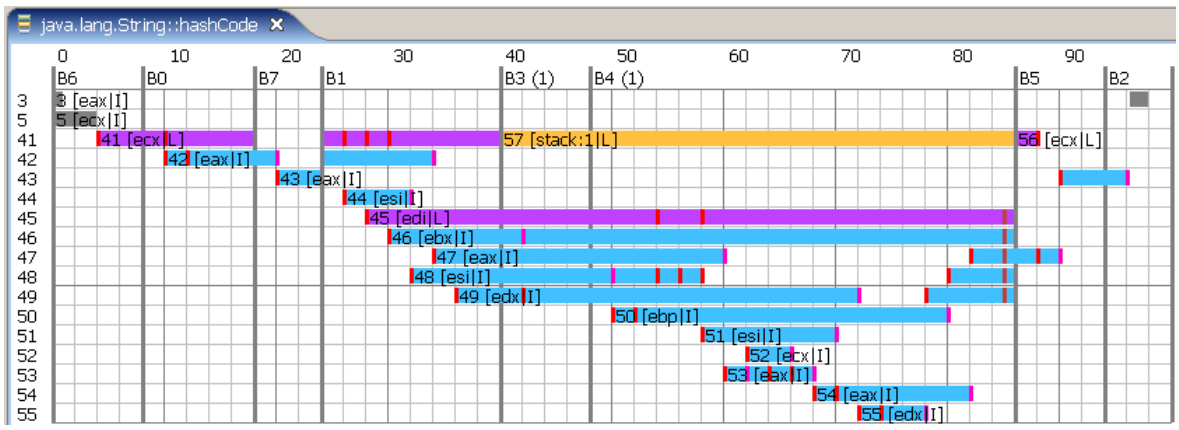

Figure 2.5: Intervals after register allocation.

2.1.6 CFG Visualizer

The third possible view on a Java method concentrates on control flow aspects. The Java HotSpot client compiler reads in Java bytecodes which are not structured, just a sequential order is given. Grouping together the bytecodes which are executed sequentially without any jumps is the task of the block builder. Such blocks are called "basic blocks". Edges between the blocks mark possible control flow. Figure 2.6 shows the control flow graph for the example method `String.hashCode`, which is divided into eight basic blocks. Each of them has a unique number and a corresponding list of HIR and/or LIR instructions which must not contain any jumps. When an edge goes from one block to another, after the last instruction of the edge start block the execution may jump to the first instruction of the edge end block. In this example the last instruction of block `B3` is an `if`-instruction and the execution may either continue at block `B4` or block `B5` depending on the result of the comparison.

Blocks have different colors depending on which internal flags are set. This automatic coloring can be customized using the preference dialog. The following list explains the most important flags:

- *std:* Marks a block as the standard entry block, which means the block where the execution of the method begins. This flag is set for `B0` in Figure 2.6. Note that this

is not necessarily the first block. In the example an artificial block `B6` containing no code corresponding to a Java bytecode was inserted before `B0`.

- *plh, llh:* These flags mark loop headers at the beginning and the end of compilation, Block `B3` is an example.
- *bb:* Blocks like `B4` containing a backedge, an edge back to the loop header, are marked with this flag.
- *osr:* When a method is compiled for on-stack-replacement, which means during interpretation the compiler decides to compile it, a special entry block is created. In this block all necessary values are pushed onto the stack and execution continues at the current position of the interpreter, normally at the start of a loop.
- *ex:* This flag marks a block as the entry of an exception handler. Such blocks can contain a high number of incoming edges.

To make loops easier to detect, the loop depth of a block affects the drawing of a block too. The border of the blocks `B3` and `B4` is a double line, meaning loop depth one. All other blocks in this example have loop depth zero and therefore the border is just a single line. Backedges are edges going from a loop end to the loop header. They have a special color. In the example there is only one backedge going from block `B4` to `B3` and it is painted red.

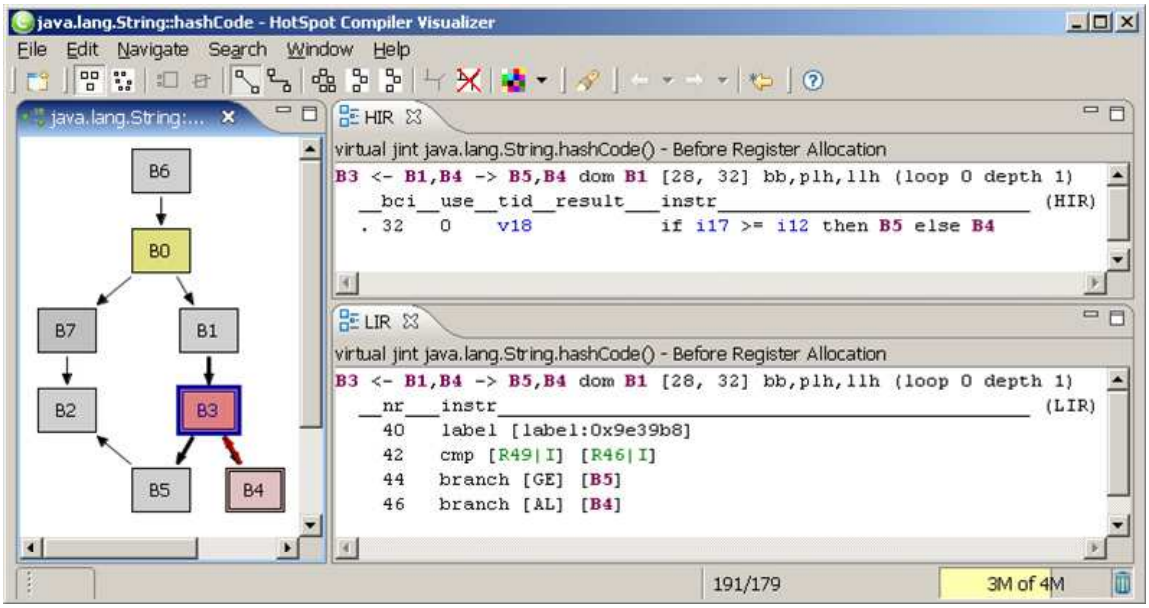

Figure 2.6: CFG Visualizer showing the method `String.hashCode`.

The visualization can be modified manually in several ways to get a better overview. First of all, the blocks can be dragged using the left mouse button. There exist several other operations on blocks which can be applied either using the toolbar buttons or the context menu:

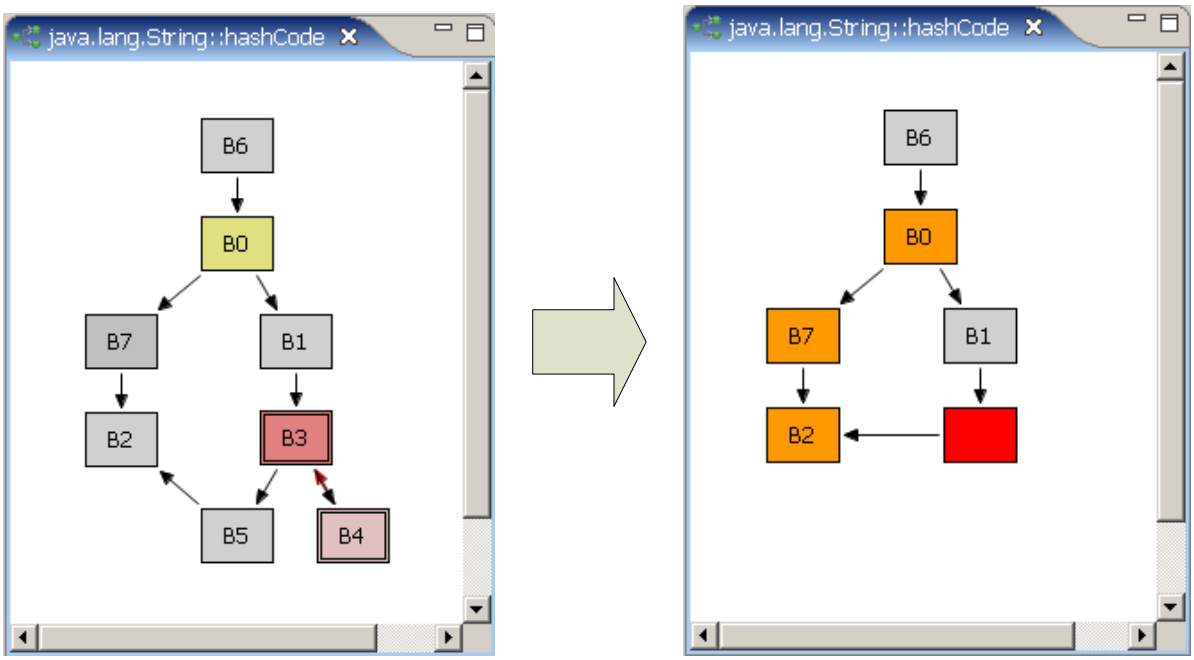

Figure 2.7: Manually changing the graph.

- *Combine:* An important functionality when analyzing large graphs is combining several blocks to a single node. This way unimportant parts can be reduced to a single node and one can focus on the important parts. In Figure 2.7 the blocks `B3`, `B4` and `B5` are combined to a single red node.
- *Split:* This is the opposite operation to combining nodes. Instead of the combined node, all original nodes are restored.
- *Show/hide edges:* Some blocks, especially exception handler blocks, have many incoming or outgoing edges, which can make the graph look bad. By hiding those edges, the overview can be improved.
- *Color:* The automatic coloring of the nodes depending on flags can be manually overwritten to mark certain edges. In the example, orange was manually assigned to the blocks `B0`, `B2` and `B7`.

For drawing the edges, there are two possible modes available: The first mode draws straight edges whenever possible, otherwise it tries to evade any obstacles and paints the edges as Bezier curves. The other one is called *Manhatten Router* and makes all connections orthogonal. The algorithms for these two approaches are described in detail in Section 6.

There are three different automatic block positioning algorithms. Figure 2.8 shows the example method displayed using all of them. The algorithms are described in detail in Section 5.

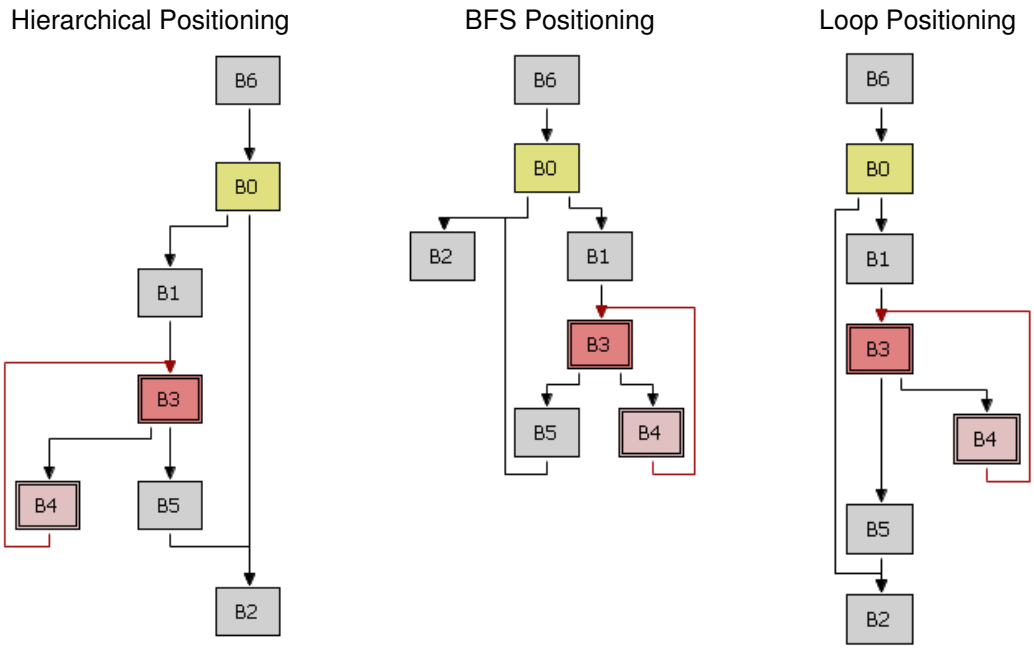

Figure 2.8: Comparison of positioning algorithms.

Especially when analyzing large methods, a quite useful functionality is zooming. Figure 2.9 shows the zoomed out representation of a large method. The user can change between the zoomed out and the normal view using toolbar buttons.

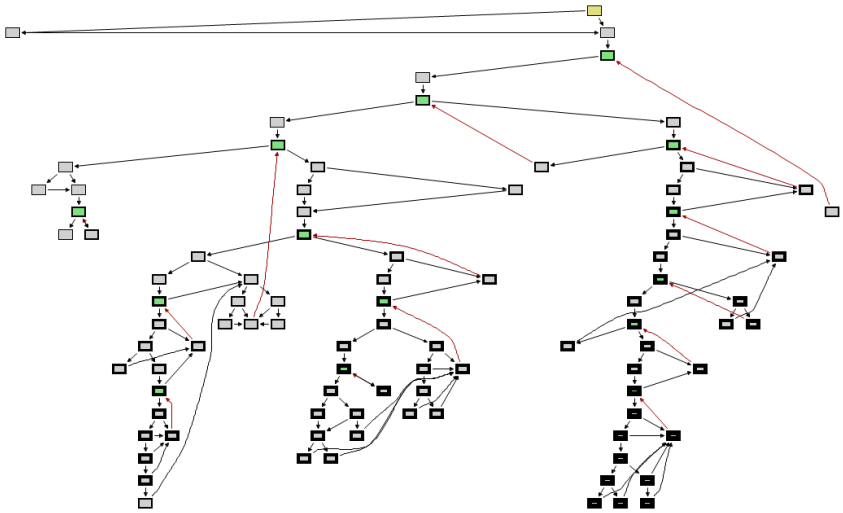

Figure 2.9: An overview of a big method.

Chapter 3

Used Frameworks

Selecting the best framework is an important issue before starting development of an application. A good choice can save a lot of development time, enabling the programmer to concentrate on the problem-specific parts instead of reinventing the wheel over and over again. At least for the basic components of the user interface it is quite common to use a framework.

For Java applications the two most popular graphical user interfaces are Swing and SWT. The differences between them is a hot discussion topic, the search engine Google for example finds more than 1500 websites containing "SWT vs. Swing" or "Swing vs. SWT". In short, SWT relies upon the native components of the operating system, whereas Swing is completely platform-independent. There are two platforms, which are built upon a basic GUI library, enriching it with more high-level functionality such as additional components and a plugin system which allows extensibility. The Eclipse Platform builds upon SWT, whereas the NetBeans Platform uses Swing.

The basic functionality of the Visualizer Application, especially the user interface, is built upon the Eclipse Rich Client Platform. The positioning and routing algorithms presented in the Chapters 5 and 6 are made as independent as possible from any other library to be highly reusable. Only some geometry classes from `org.eclipse.draw2d.geometry` like `Point` or `Rectangle` are used.

3.1 Eclipse

Eclipse development started in 1998, when IBM decided to produce a development tool platform. Many ideas of the proprietary development environment IBM VisualAge were used when Eclipse was designed. The most important property of Eclipse is its extensibility. A powerful plugin system makes it easy to program plugins, extending the functionality of Eclipse. The Eclipse Rich Client Platform is not a development environment by itself, the tools built as plugins on top of the platform make up the development environment. The plugin `org.eclipse.jdt` for example is the implementation of the Java IDE. Normally when the term "Eclipse" is used, the Java programming environment available at `http://www.eclipse.org` is meant. It is however just the Eclipse Rich Client Platform with a couple of selected plugins. [Ecl06]

In 2001 IBM decided to convert Eclipse to an open source project, in 2004 the Eclipse Foundation was created. It is a completely independent organization and its members release Eclipse-based products. Many of the Eclipse plugins are open source and freely available, there are however also some commercial tools. There exist a lot of books about Eclipse, e.g. [AL04] or [GB03].

3.1.1 Basic Architecture

To understand where plugins can extend Eclipse and what kind of concepts are built into the Eclipse Rich Client Platform, Figure 3.1 gives an overview of the basic architecture. The following list gives a short introduction to the terminology of Eclipse.

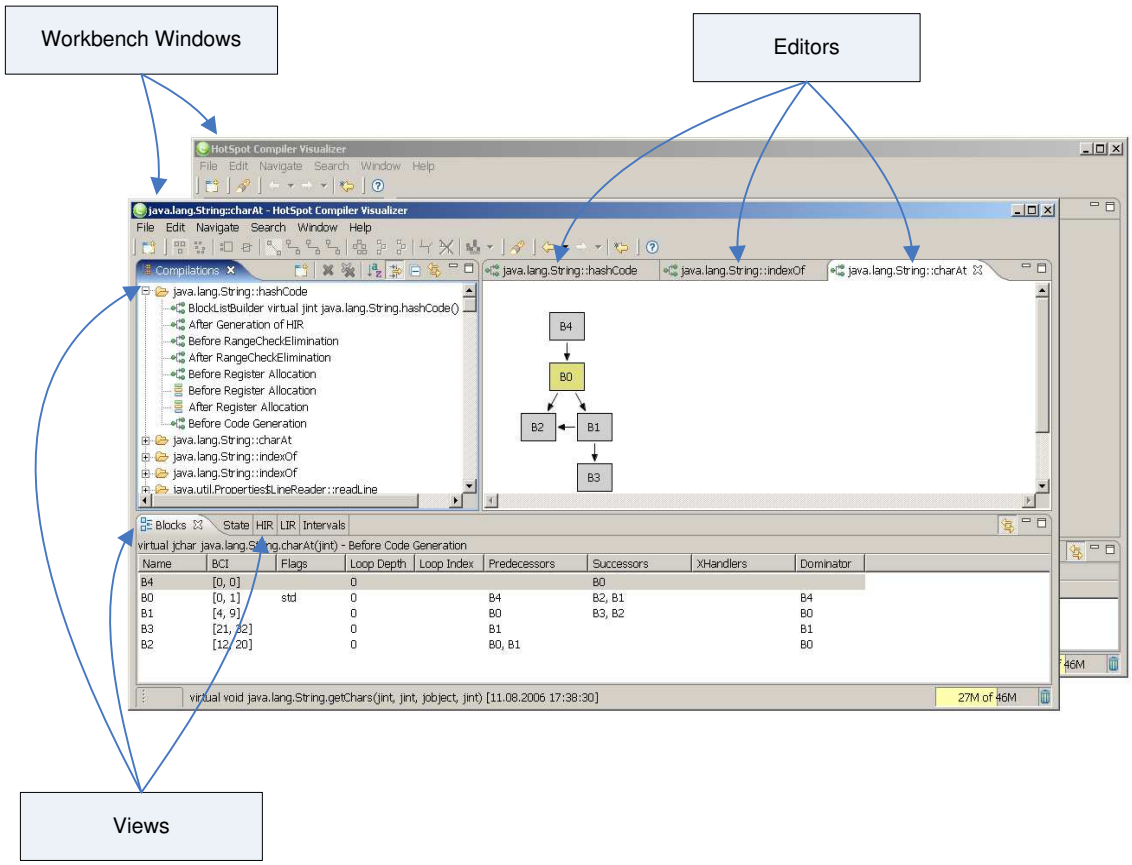

Figure 3.1: Structure of Eclipse.

- *Workbench Window:* The topmost window is called a Workbench Window. There may exist more than one of it, the user can open new Workbench Windows by clicking `Window→New Window`.

- *Editor:* Associated with a specific Workbench Window there are one or more Editors. An editor works on a specific input which can be for example a file and has an "open-edit-save-close" lifecycle. The base class for editors `EditorPart` contains an

abstract method `init` which takes an `IEditorInput`-object as one of its parameters. The methods `doSave` and `doSaveAs` are also abstract. As you can see in Figure 3.1, Editors are shown stacked on top of each other in the main part of the Workbench Window.

- *View:* Views also contribute a part to the Workbench Window, however, they do not work on a specific input. Additionally each kind of View usually exists once per Workbench Window. The interface `IViewPart` has a method `init` as well as a method `saveState`, which is used for example to restore the state of the View after a restart. Views often display additional information about the current active Editor or are responsible for opening Editors. They can be located anywhere around the main window and can also be stack on top of each other.

- *Perspective:* A perspective specifies the arrangement of Views. When the application is shut down, these settings are saved and restored at the next start-up. There is also a menu item `Window`→`Reset Perspective`, which restores the initial perspective.

3.1.2 Eclipse Plugin Development

This section gives a short introduction of how Eclipse plugins are interconnected. The structure of Eclipse is like a puzzle, a lot of separated modules which have dependencies make up the application. The core just loads and manages plugins. So what makes up an Eclipse plugin?

- *Java Class Files:* The functionality of a plugin is encoded in Java class files. Part of Eclipse is a special class loader, which ensures that while the plugin is not needed, none of its classes are in memory.

- *Extensions:* A plugin needs to declare which kind of functionality it wants to contribute. It is only allowed to add to well-defined extension points. Extensions are described in the file `plugin.xml`. A plugin may extend other plugins any number of times.

- *Extension Points:* A plugin can define any number of extension points to be used by other plugins. There can be many plugins extending a single extension point. If this does not make sense, the extension point owner must take care what to do in case of multiple extensions.

Figure 3.2 shows the plugins of the Visualizer Application and how they use extension points of the `org.eclipse.core.runtime` and the `org.eclipse.ui` plugin. Looking at this diagram gives an overview of what kind of functionality the application is adding to Eclipse. There are for example three new editors (CFG Editor, Interval Editor and IR Editor). To be able to save preferences, the plugin `CFG Visualizer` uses the extension point `org.eclipse.core.runtime.preferences`. If those preferences should be configurable in the preference dialog, the extension point `preferencePage` of the plugin

`org.eclipse.ui` must be extended. There is also an `org.eclipse.ui.views` extension point where all the views are registered. Note that when using this extension point it is possible to register multiple views using only a single extension. A plugin can also just contribute its code and define no extensions or extension points such as the plugin `Visualizer Data`.

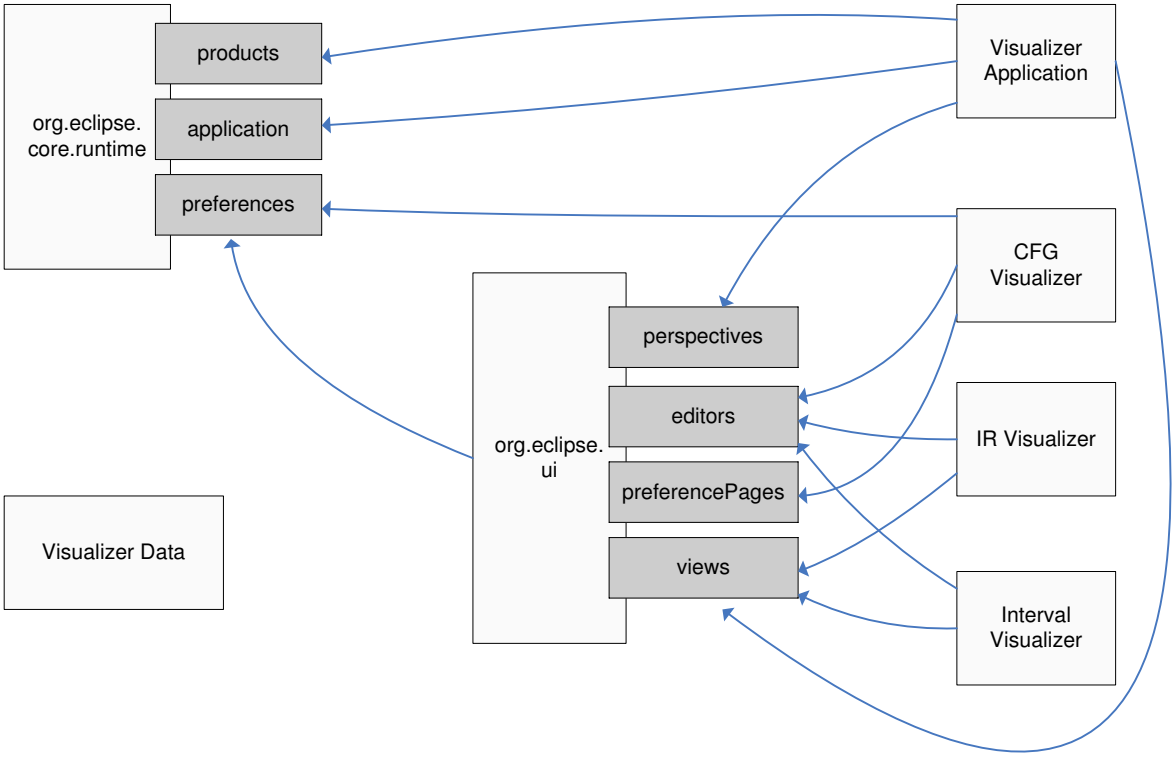

Figure 3.2: Extension point diagram.

3.1.3 The XML Configuration File

For defining extensions or extension points, there exists an XML file called `plugin.xml` for each plugin. It can contain additional information about the plugin such as its name or required plugins. As child elements of the root element <`plugin`>, the elements <`extension`> and <`extension-point`> can be defined. Each extension point is identified by its full qualified name consisting of the id of its plugin and its name. The child elements of <`extension`> are plugin-specific and follow no special rules. However there are some conventions and there are also `.exsd`-files which are "Extension XML Files". These files describe the XML elements allowed inside the <`extension`> element. Listing 3.1 shows the XML configuration file of the CFG Visualizer. There is an extension to `org.eclipse.ui.editors`, which defines a new kind of editor. An extension point is used to add a page to the preference dialog and another one registers an initializer for the preferences.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
  <extension point="org.eclipse.ui.editors">
    <editor
      id="at.ssw.visualizer.cfg.editor.CFGEditor"
      class="at.ssw.visualizer.cfg.editor.CFGEditor"
      contributorClass="at.ssw.visualizer.cfg.editor.CFGEditorActions"
      icon="icons/cfg.gif"
      name="CFG␣Editor">
    </editor>
  </extension>
  <extension point="org.eclipse.ui.preferencePages">
    <page
      class="at.ssw.visualizer.preferences.CFGPreferencePage"
      id="at.ssw.visualizer.preferences.CFGPreferencePage"
      name="CFG␣Editor"/>
  </extension>
  <extension point="org.eclipse.core.runtime.preferences">
    <initializer
      class="at.ssw.visualizer.preferences.PreferenceInitializer"/>
  </extension>
</plugin>
```

Listing 3.1: The plugin.xml-file of the CFG Visualizer.

3.1.4 Lazy Loading

One important feature of the Eclipse plugin system is lazy loading. The XML file describing the extensions and the extension points of a plugin is always loaded on start-up. The classes however, are only loaded when the plugin is needed. So for example if a plugin contributes to the menu bar, the xml-file contains everything necessary to display the menu item, e.g. the path to an icon. The plugin itself and all classes of it remain unloaded until the user uses the plugin by clicking the menu item.

In case of the Visualizer Application, the `CFGPlugin` for example is only loaded when the user requests viewing a control flow graph. While the user just looks at interval visualizations, the `CFGPlugin` remains unloaded not wasting any resources.

Figure 3.3 was created using an exploration tool called Spider. It shows the basic structure of the platform. There is the class `Platform` which only consists of static methods and fields. Associated with it, there is an `IPluginRegistry`-object that maintains the plugins as well as their extensions and extension points. Using the method `getPluginDescriptions` and searching the resulting array, the `PluginDescriptor`-object of the `CFG Visualizer` plugin can be found. This is a wrapper object around the actual plugin object, containing also information about the extension points this plugin is providing and the extensions it is making. As long as nobody needs the functionality of the plugin, it remains unloaded and the field `pluginObject` is `null`.

Figure 3.4 shows the same object graph after a control flow graph was opened. Now

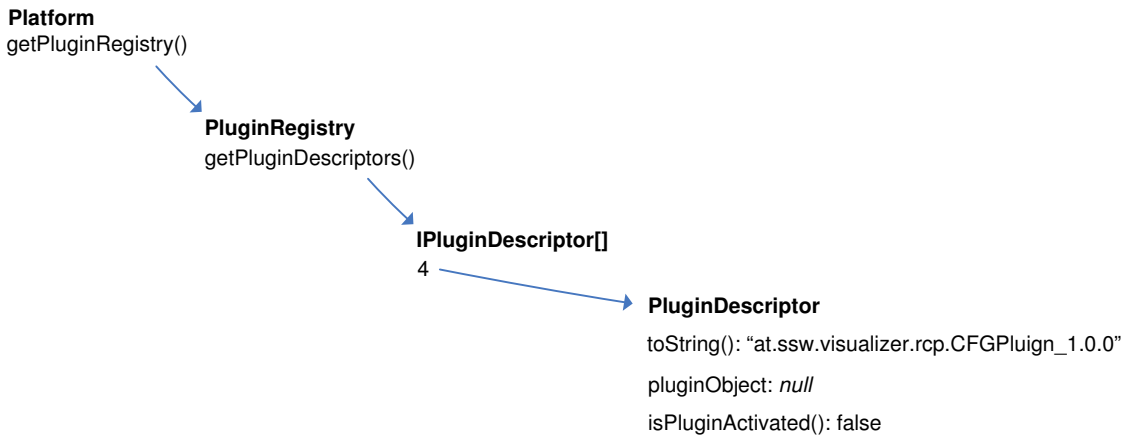

Figure 3.3: Plugin object before first activation.

the `CFGPlugin` is activated and loaded. The plugin object was automatically created. A plugin can also be unloaded when it is not needed any more.

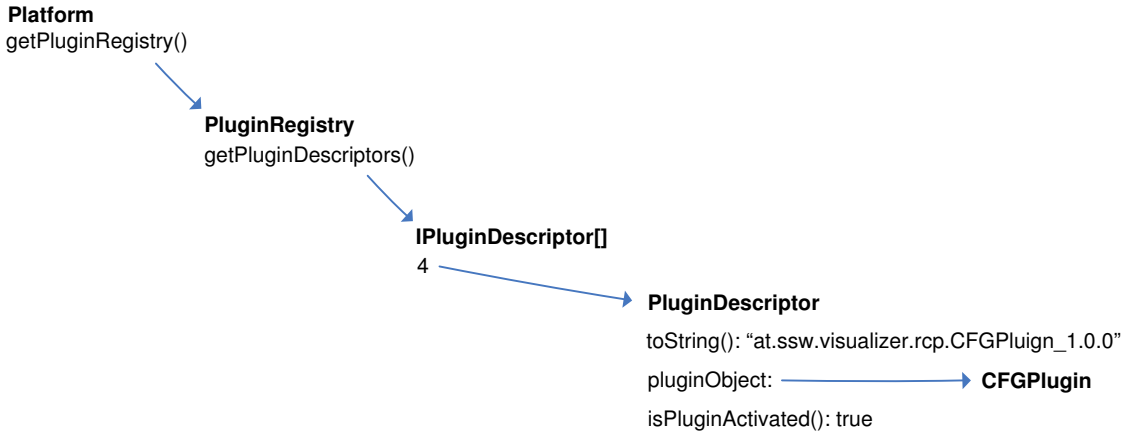

Figure 3.4: Plugin object after first activation.

3.2 Dependencies

To get an overview of what kind of libraries of Eclipse the Visualizer Application is using, the following list shows the most important packages the application depends on. Note that this is a simplified view: These packages are divided into subpackages and there are several other packages necessary to run the application not mentioned here.

- *`org.eclipse.swt`:* Used for the basic widgets such as `Button` or `Canvas` as well as for graphical resources such as `Color` or `Font`.
- *`org.eclipse.jface`:* This is a helper package mainly used for maintaining the preferences. `ColorFieldEditor` and `FontFieldEditor` for example display dialogs for editing color and font settings.

- *org.eclipse.ui:* To be integrated in the Eclipse workspace some classes of the package `org.eclipse.ui` are needed. The class `CFGEditor` for example is a subclass of `org.eclipse.ui.EditorPart`.
- *org.eclipse.draw2d:* This package is needed for the display of the graph, which heavily uses `IFigure` objects. The class `PolylineConnection` for example is used for the connection between nodes, the class `PolygonDecoration` is responsible for painting the arrows.

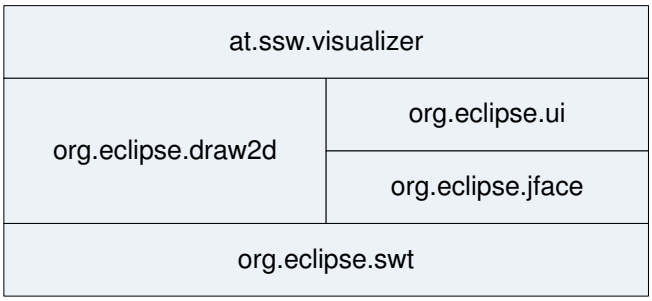

Figure 3.5: Package dependencies.

Figure 3.5 shows how the main packages are built on top of each other. The packages `org.eclipse.draw2d` and `org.eclipse.jface` are built on top of `org.eclipse.swt`. The package org.eclipse.ui depends on `org.eclipse.jface`, while the application itself needs all of the other packages.

3.3 Draw2D and GEF

The Graphical Editing Framework (GEF) is a subproject of the Eclipse tools project. It is split into two plugins:

- *org.eclipse.draw2d:* Responsible for drawing and arranging objects, special support for drawing graphs.
- *org.eclipse.gef:* Enhances Draw2D by providing mechanisms for editing graphical objects and provides a model-view-controller framework.

It is an important decision whether to use only Draw2D or both. Using GEF makes the application more complex however also very powerful when editing is the main purpose. On the one hand CFG Visualizer allows modification operations on the graph like resizing or merging. On the other hand, the results need not be saved and there is also no need for undo and redo capabilities. There is also no need to create or delete nodes. To keep the application simple, it uses only Draw2D.

3.3.1 Draw2D

Draw2D is a lightweight graphical system, where any graphic object is referred to as a figure. Those figures behave very much the way SWT widgets would do, but in fact they are just Java objects and do not require operating system resources. This is why they have the property "lightweight". They need fewer resources than native windows and are also more flexible. This way it is for example possible to create a lot of figures without any performance problems, and additionally those figures are not restricted to have a rectangular shape. A line, for example, is also a figure. It wouldn't be possible to create a line as a native window.

Draw2D tries to hide that the figures are no real windows. It provides functionality that they can have focus and react to keyboard and mouse events. All lightweight figures are painted inside one heavyweight control. The class `LightweightSystem` is the connection between SWT and the figures. It has two helpers:

- *SWTEventDispatcher:* All kind of SWT events first come to the only heavyweight control. The `SWTEventDispatcher` is responsible for converting the SWT event and sending it to the correct figure.
- *UpdateManager:* Performance is an important issue, it would be far too slow if the whole lightweight system would repaint itself whenever a change occurs. Therefore the class `UpdateManager` tries to minimize the region which needs to be repainted and tells only the figures within that region to repaint.

Listing 3.2 gives a small Draw2D-example. In this case Draw2D is used just on top of SWT. The four steps done here are:

- Create a `Shell`-object which represents the main window, this step is SWT-specific and is only done to get an object of type `Canvas`.
- Create the `LightweightSystem`-object as a child of the main window. As the argument of the constructor any `org.eclipse.swt.widgets.Canvas`-object is allowed. This object provides the link between SWT and Draw2D.
- Now any number of figures can be added to the lightweight system. The method `setContents` defines the main figure. All figures can contain other figures.
- The last step is again only needed because it is a standalone application. It ensures that the event loop is executed until the user exits the application.

```
// Create a Shell, which is the top window
Shell shell = new Shell();
shell.open();

// Create the lightweight graphical system
LightweightSystem lws = new LightweightSystem(shell);

// Create a label and add it to the system
```

```
Label label = new Label("Hello world!");
lws.setContents(label);

// Event loop while shell is not exited
Display display = Display.getDefault();
while (!shell.isDisposed()) {
  if (!display.readAndDispatch()) {
    display.sleep();
  }
}
```

Listing 3.2: Simple Draw2D example.

Figure 3.6 shows the resulting application in action. This is a simple example and the screenshot looks just like a screenshot of a normal SWT application. Internally there is however a big difference. The `Label`-object in this example is only virtually a window. It is just a Java object with a method `paintFigure` that is called by the lightweight system and draws the text.

Figure 3.6: Draw2D "Hello world!" application.

3.3.2 GEF

GEF is built on top of Draw2D and it depends not only on SWT, but on several Eclipse-plugins. Therefore it is not possible to use it on its own like Draw2D. It is a good framework for an Eclipse-plugin, which contributes a complex graphical editor with support for manipulation, undo/redo, zooming and printing. There are a lot of useful high-level components built into GEF.

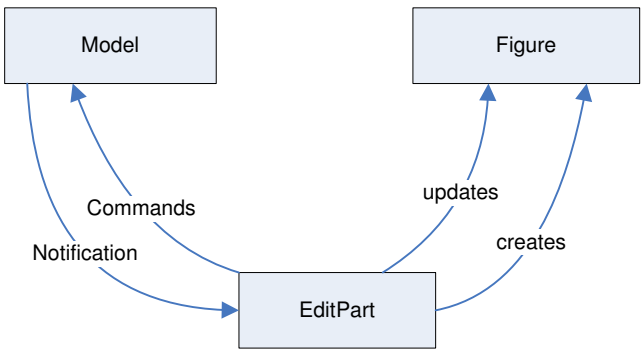

Figure 3.7: Basic GEF architecture.

The biggest disadvantage of GEF is its steep learning curve. It is quite complicated to use because it contains so many pre-built classes and mechanisms. Also some of the concepts are not very generic and only useful if your graphic editor uses the GEF way of thinking.

A real introduction to GEF is beyond the scope of this thesis, the basic structure of the Model-View-Controller framework is however noteworthy. A central role play the `EditPart`-objects which are responsible for coordinating models and figures. They are the controllers, whereas the Draw2D figures fill in the view part. Figure 3.7 shows the relationship between `Model`-objects, `Figure`-objects and `EditPart`-objects.

Chapter 4

Program Architecture

Figure 4.1 shows the five plugins which make up the Visualizer Application. Visualizer Data is responsible for reading the input file and make its content accessible to all other plugins. The plugin named Visualizer Application contains code to display all compiled methods. Depending on user input, this plugin now activates one of the three topmost plugins and tells it to open an editor.

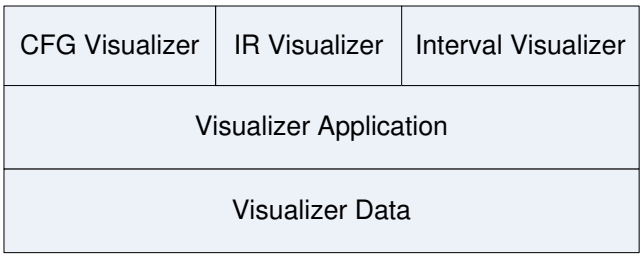

Figure 4.1: Plugin dependencies.

The CFG Visualizer plugin is the focus of this thesis. Chapters 5 and 6 contain descriptions of the most important algorithms used by this plugin. The other plugins were developed by Christian Wimmer. CFG Visualizer is built upon Visualizer Application and Visualizer Data.

4.1 Visualizer Data

This package parses the .cfg input file and creates a Java object tree. Figure 4.2 shows the most important classes of this plugin and their relationships. A single `InputData`-object contains all contents of the input file. For every compiled method it has a `Compilation`-object. Such an object contains a `CompilationElement`-object for each state during compilation. A `CompilationElement` can be either a `ControlFlowGraph` (input for CFG Visualizer or IR Visualizer) or an `IntervalList` (input for Interval Visualizer).

A `ControlFlowGraph`-object contains any number of basic blocks. For every basic block its flags (see section 2.1.6), predecessors and successors as well as the intermediate representations are stored in memory.

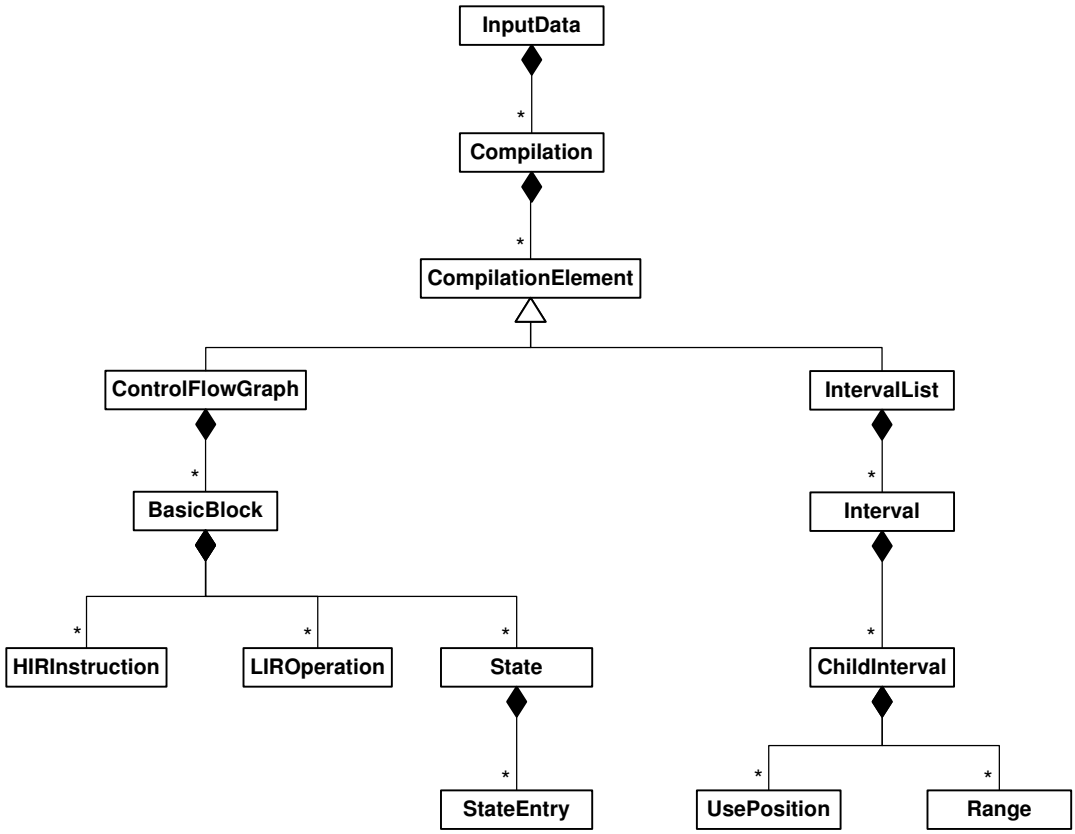

Figure 4.2: Package `at.ssw.visualizer.data`.

The input for the Interval Visualizer contains `Interval`-objects as well as objects of the class `ChildInterval`. Additionally positions and ranges of the intervals are part of the input. To make it possible to see the current intermediate representations and other properties of a block also while displaying an `IntervalList`-object, a reference to the last `ControlFlowGraph`-object is stored.

4.2 CFG Visualizer

The CFG Visualizer plugin is one of the three plugins contributing editors. In this case, an editor for the control flow graph is added. `BasicBlock`-objects are nodes in the graph and the connections between them form the edges. Draw2D (see section 3.3.1) is used for drawing. The next section gives a short overview of the package `at.ssw.visualizer.cfg.editor`, which contains all the code for displaying the graph.

4.2.1 Editor Package

Figure 4.3 shows the class diagram with all classes of the package. A `CFGEditor`-object represents the editor window containing the drawing. It has exactly one instance of the class `GraphPanel`, which is the topmost Draw2D figure, as well as one instance of the

current routing and positioning algorithm. A `GraphPanel`-object contains any number of `Node`-objects. There are two subclasses of class `Node`: `BlockNode` represents a standard node and stands for exactly one `BasicBlock`-object. `MultiNode`-objects are combined nodes. See section 2.1.6 on how to combine nodes. Every `Edge`-object is associated with exactly two nodes.

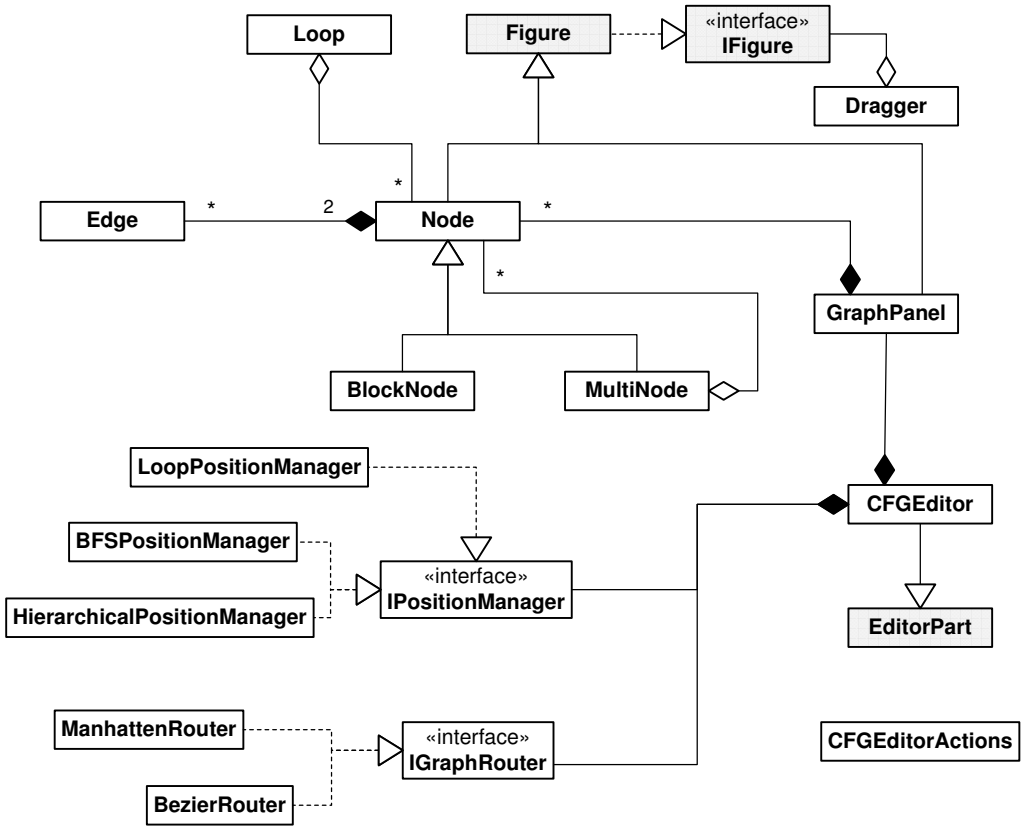

Figure 4.3: Class diagram of package `at.ssw.visualizer.cfg.editor`.

Actions that set for example the current router or position manager are created by the class `CFGEditorActions`. `IPositionManager` is the interface for classes which implement an algorithm for automatically setting coordinates for all nodes. Only the class `CFGEditorActions` has a reference to the specific classes, while in the other parts of the code, always the interface `IPositionManager` is used. This way new position managers can be added easily to the application and only one class needs to be changed.

`IGraphRouter` is the interface every routing algorithm needs to implement. The same design pattern like for the positioning algorithms is applied here, making it easy to extend the application with new routing algorithms.

4.2.2 Model Package

CFG Visualizer heavily depends on the plugin Visualizer Data. To make it easier to react on changes in Visualizer Data and to add the opportunity to display a completely different

control flow graph, an additional abstraction layer was inserted between the two plugins. The class `GraphPanel`, which is responsible for creating the figures needed to display the graph, neither works with a `ControlFlowGraph`-object nor does it use any other classes of `at.ssw.visualizer.data`. It works with `IGraphModel`-objects.

Figure 4.4 shows the class diagram of the data model. An `IGraphModel`-object contains any number of `INodeModel`-objects and also the connections between them represented as `IConnectionModel`-objects.

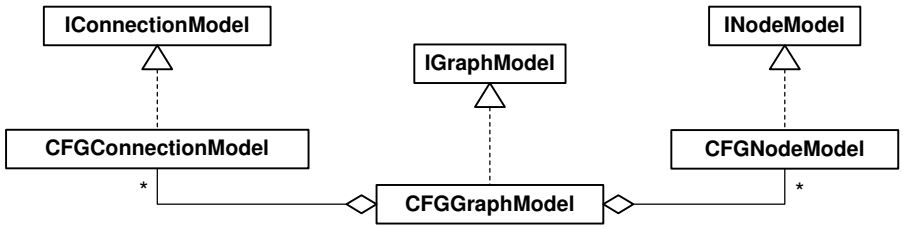

Figure 4.4: Package `at.ssw.visualizer.cfg.model`.

The concrete implementations of the interfaces for the specific control flow input are `CFGGraphModel`, `CFGNodeModel` and `CFGConnectionModel`. Those classes hide the plugin Visualizer Data from class `GraphPanel` and its related classes.

Chapter 5

Positioning Algorithms

Given the graph as the input, a positioning algorithm sets the x- and y-coordinates of all nodes to a specific value. It has to implement the interface `IPositionManager` (see Listing 5.1). New position managers can be added easily, only the class `CFGEditorAction` needs to be changed: A positioning algorithm needs to contribute an action that instantiates the position manager and passes it to the `CFGEditor`-object when activated. The following three sections describe the different implemented positioning algorithms.

```
public interface IPositionManager {
  void reposition(GraphPanel panel);
}
```

Listing 5.1: Interface IPositionManager.

5.1 BFS Positioning Algorithm

Drawing a graph without cycles, also called tree, is much easier than displaying a generic graph. Therefore the breath first search (BFS) positioning algorithm first transforms the input graph into a tree by ignoring certain edges and then tries to find a good visualization for the resulting tree.

5.1.1 Breath First Search

For a better understanding of this positioning algorithm, it is explained how a breath first search works. A detailed description of breath first search can be found in [Sed03]. The algorithm is guaranteed to reach every node of the graph and has a runtime complexity of $O(n)$. Listing 5.2 gives a pseudocode representation of the algorithm. The minimum distance from the root of a node is used by the positioning algorithm to calculate the y-coordinate.

Figure 5.1 shows the breath first search algorithm working on the example method `String.hashcode`. Note the backedges from `B5` to `B2` and from `B4` to `B3`, which are ignored. The graph without those edges is guaranteed to be acyclic.

```
active = all nodes without input edge
dist = 0
while active is not empty do
  mark all active nodes as visited
  dist = dist + 1
  active = all unvisited nodes reachable from current active nodes
  for each node in active
    minimum distance from root of this node = dist
  end for
end while
```

Listing 5.2: Breath first search.

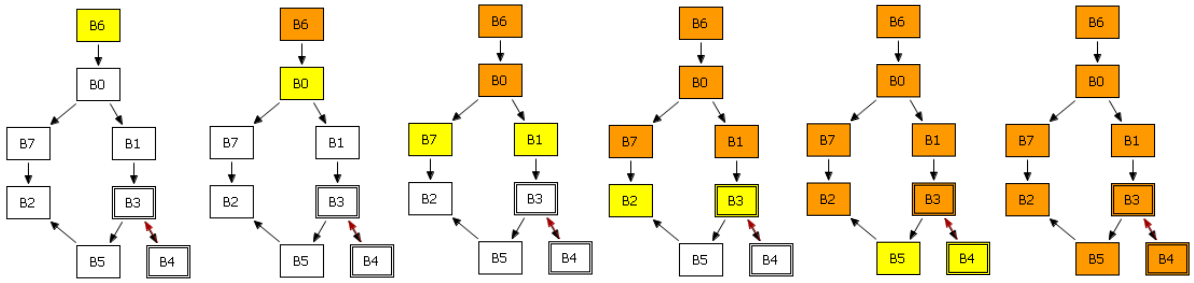

Figure 5.1: Breath first search on the example method.

5.1.2 Tree Drawing

If a graph has only one root and is acyclic, it is called a tree. Normally, the control flow graph of Java methods do not contain multiple nodes with no incoming edge. There are however rare cases where during compilation dead code is not deleted and part of the graph. In this case, the algorithm adds a virtual root with edges to all real roots. This way the graph will always become a tree.

For the calculation of the x-coordinate of a node, first the sum of the widths needed by its children is retrieved. When a node does not have any children, its width is a fixed value. Otherwise, its width is calculated recursively by summing up the width of all children. A node is positioned in the middle of the space occupied by its children.

5.1.3 Results

The algorithm does not use any special properties of control flow graphs and can be applied to any graph. This is a disadvantage, because the user cannot read out much control flow related information from the graph. However, loops are for example usually grouped close together. Additionally, the y-coordinate gives the minimum number of blocks that must be executed before a certain block is reached. An advantage of the BFS algorithm is that the height of the drawing is usually smaller in comparison to the other two algorithms. This makes the graph look compact, especially when displaying graphs

with a lot of nodes. Figure 2.9 in the user guide gives an example for a big method being positioned by this algorithm.

5.2 Loop Positioning Algorithm

In contrast to the previous algorithm, the loop position algorithm is designed for the specific case of displaying control flow graphs. Its main purpose is to make loops more separated and to display the blocks in a similar way to the underlying Java source code.

5.2.1 Grouping Loops

First of all, this algorithm groups loops together to subgraphs. The outermost scope is treated like a loop with the root node as the loop header. From the point of view of an outer loop, an inner loop is nothing else than a simple node with a custom width and height. All inter-loop connections are ignored and are viewed as if the whole subgraph would have the connection.

Figure 5.2 shows the loop positioning algorithm applied on the method `rehash` of class `java.util.Hashtable`. The black rectangles and the text are not part of the program output, they just mark the internal grouping of the graph.

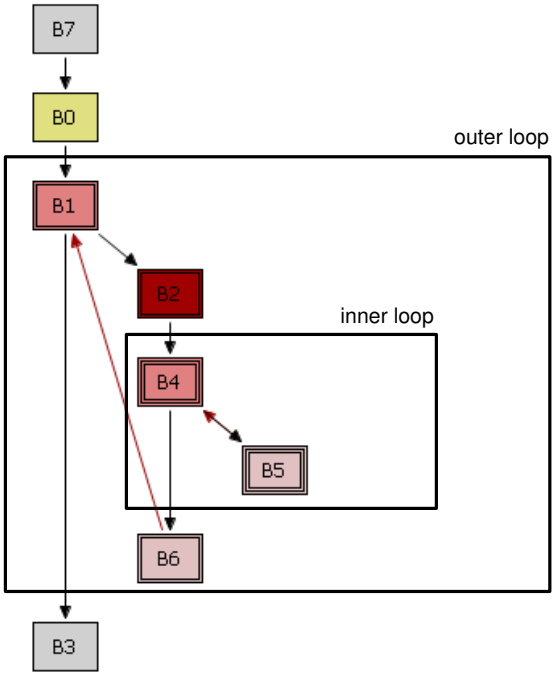

Figure 5.2: Grouping loops together.

Note that for the positioning algorithm, e.g. the connection between `B4` and `B6`, which is an inter-loop connection, does not exist. Instead the virtually created node for the inner loop has a virtual connection to `B6`.

The positioning algorithm is applied from the innermost loops to the outermost loop. Only when the width and height of all inner loops are calculated, an outer loop can be processed.

5.2.2 Modified BFS

Assigning the y-coordinate like the BFS positioning algorithm does not lead to graphs which look similar to the original source code. In the positioning algorithm presented in section 5.1 the y-coordinate of a node depends on its minimum distance to the root node. Source code similarity is a lot higher if the maximum distance determines the y-position.

Fortunately, calculating the maximum distance of a node to the root can be done easily by modifying the BFS algorithm: An edge is only taken as existant if there is no other incoming edge of the destination node from an unvisited node. So instead of taking the first edge leading to a node, the last one is used.

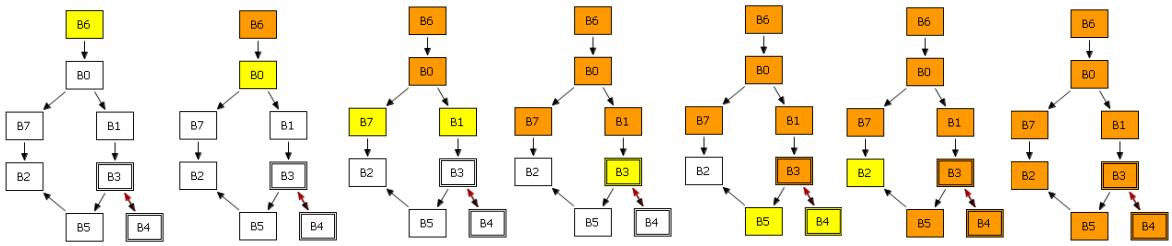

Figure 5.3: Modified breath first search.

Figure 5.3 shows the modified breath first search algorithm applied on the example used in Figure 5.1. The first difference between the modified BFS algorithm and the algorithm presented in Section 5.1 occurs at the fourth step: The modified algorithm does not set `B2` active because there is another way to reach `B2`. As long as `B5` is not visited, `B2` remains in the initial state.

5.2.3 Results

The Java source code responsible for block `B2` is likely to be the last lines of `Hashtable.rehash`. Therefore it reflects the structure of the method, if this block is drawn at the bottom of the graph. Grouping together loops and ignoring jumps out of loops are also good heuristics for making the graph look similar to the code. A loop containing a `break` instruction for example looks better compared to the BFS positioning algorithm.

5.3 Hierarchical Positioning Algorithm

This is a common algorithm often used to calculate positionings for directed graphs, it is described in detail in [BETT99]. There is an implementation of it in the package `org.eclipse.draw2d.graph`, so the Visualizer Application just needs to convert its own

data structures to the data of the algorithm. This section gives an overview of the steps performed by the algorithm.

5.3.1 Algorithm Steps

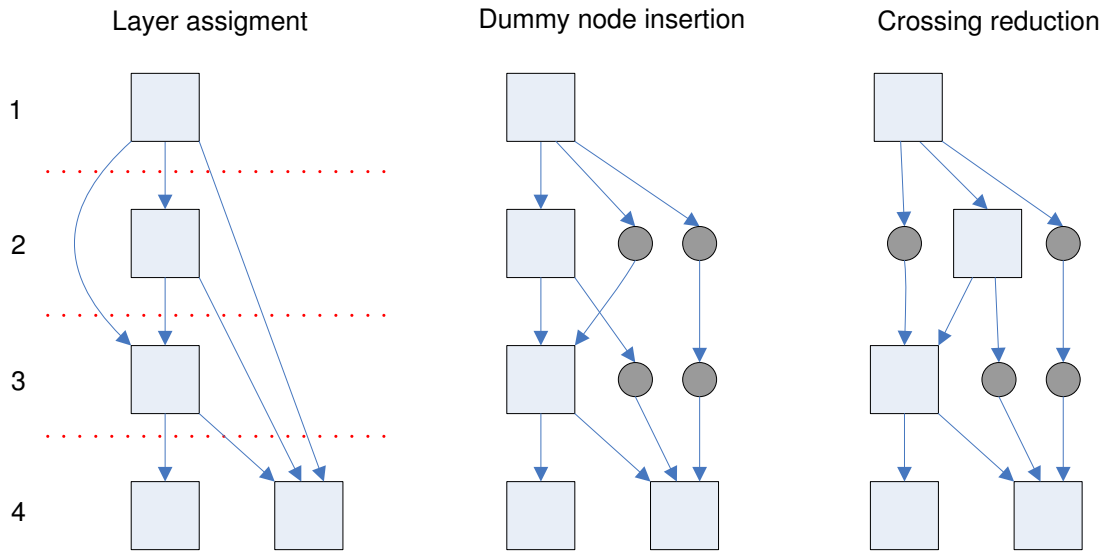

Figure 5.4: Hierarchical positioning algorithm.

- *Break loops:* Just like the other two algorithms, the hierarchical positioning algorithm can only work on directed acyclic graphs. For a better control flow graph related drawing, all edges marked in the input as backedges are removed. This makes the graph acyclic.
- *Layer Assignment:* In the second step, a layer is chosen for every node. This is done just like in the loop position algorithm. However, loops are not treated specially, therefore the resulting layout is not that high. The leftmost part of Figure 5.4 shows a sample graph after layer assignment.
- *Adding Dummy Vertices:* Whenever there is an edge between two nodes that are not assigned to succeeding layers, dummy nodes are inserted. In Figure 5.4 such nodes are drawn as circles. After this step it is guaranteed that no edge spans over more than one layer. This makes the next step a lot easier.
- *Crossing Reduction:* The quality of a drawing mainly depends on the number of edge crossings. Therefore minimizing them has a high priority when trying to paint a good-looking graph. In the crossing reduction step, the algorithm tries to minimize the crossings when looking only at the current and the next layer. Even this simplified crossing reduction is an NP-complete problem, there exists no algorithm with a polynomial running time. So the perfect solution can only be found if the number of nodes is small. However, there exist several heuristics which give quite good results in the common case.

- *Horizontal Coordinate Assignment:* In the last step of the algorithm x-coordinates are assigned to the nodes.

5.3.2 Results

Despite the fact that the algorithm does not use any control flow specific information like loop depth or loop header flags, it gives quite good results. Because of the sophisticated crossing reduction and other heuristic optimization steps, the graph looks nice. A disadvantage in comparison with the two other approaches is that nodes of the same loop do not stick together. So it is harder to see the loop structure of big methods.

5.4 Comparison

In Figure 5.5 the three approaches are compared using a method of medium size. The start block is marked dark gray, loop headers are red, blocks with loop depth greater than zero are orange and blocks without successors are drawn green. All other blocks are painted yellow. This way one can easily see the differences between the three algorithms.

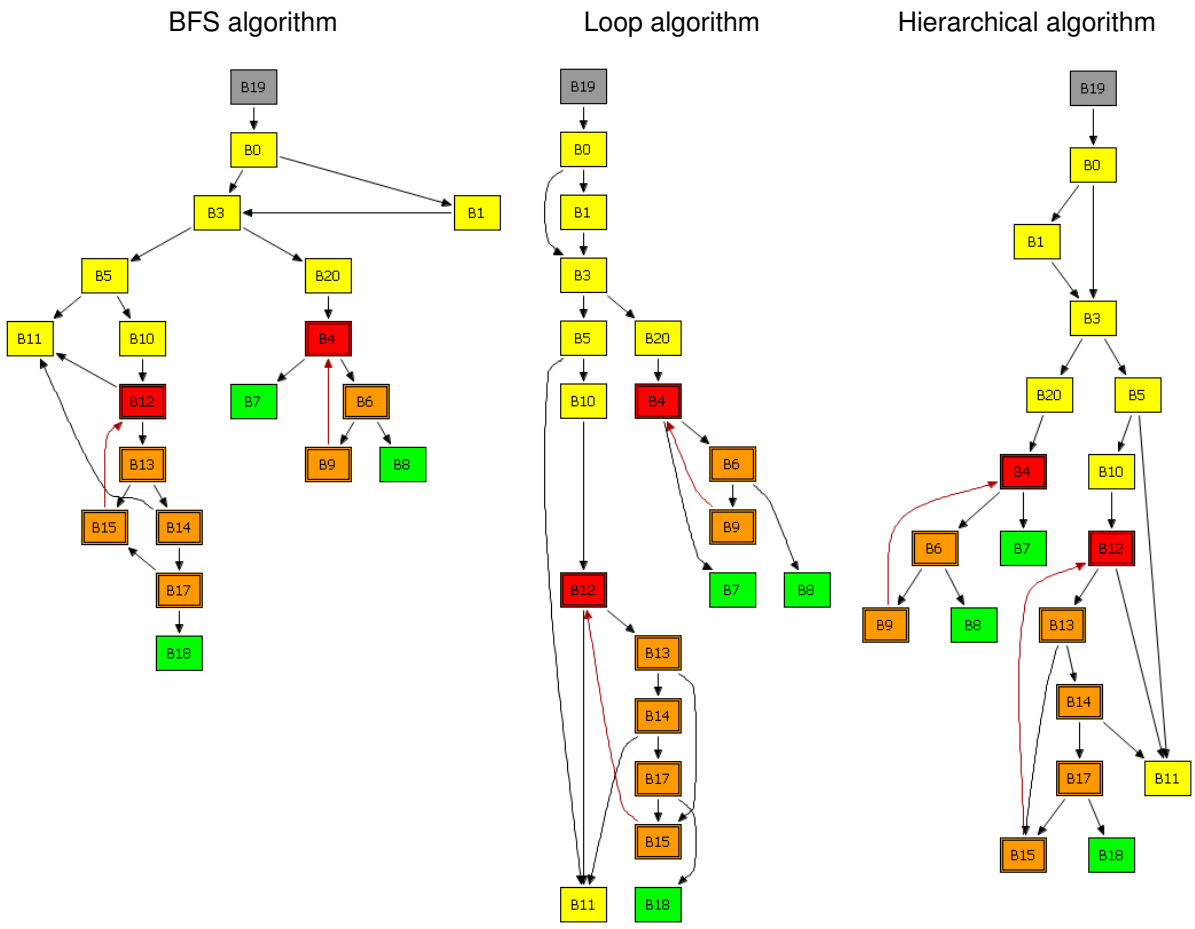

Figure 5.5: Comparison of the three positioning algorithms.

If one looks at the number of edge crossings, the hierarchical algorithm is the winner. The crossing reduction step pays of, in this case all crossings could be eliminated. The other two algorithms both lead to two edge crossings. An enhancement for the first algorithm could be a crossing reduction step, which tries to rearrange the children of a node. Swapping `B14` and `B15` in this example would eliminate the crossings.

The loop positioning algorithm gives of course the best positionings regarding loops. Blocks within a loop are most clearly separated from others. The other two approaches place the green exit nodes and yellow normal nodes closer to loop nodes. As expected, the BFS algorithm gives the shortest drawing, while the loop algorithm results in the longest one.

It depends on the structure of the control flow graph which approach gives the best results. Also important is the focus of the user: The loop positioning algorithm is probably best for searching for the block containing HIR or LIR instructions for a specific source code location. For searching the shortest possible control flow to a certain block or when a small height is preferred, the BFS positioning algorithm is the best choice.

Chapter 6

Routing Algorithms

A class implementing a routing algorithm for the Visualizer Application needs to be a subclass of class `AbstractRouter`. This way it can be set as the router for a Draw2D `Connection`-object. Additionally it needs to implement the interface `IGraphRouter` (see Listing 6.1). This makes them different from generic routers. They know the current `GraphPanel`-object and can therefore use several properties of the specific control flow graph. This is their big advantage compared to the common prebuilt Draw2D routers. The method `init` of `IGraphRouter` is used to tell the router on which `GraphPanel`-object it is working on. `sizeChanged` is called, whenever the size of the panel is modified.

```
public interface IGraphRouter {
  void init(GraphPanel panel);
  void sizeChanged();
}
```

Listing 6.1: Interface IGraphRouter.

The class `AbstractRouter` is an abstract class and is just an incomplete convenience implementation of the interface `ConnectionRouter`. Listing 6.2 shows the two most important methods of `ConnectionRouter`. The method `route` is called with a `Connection`-object as the argument. The router calculates bendpoints and sets them using the `setPoints`-method of the class `Connection`. The second method is called when a connection is no longer valid. This can mean that it has become invisible, or one of its end points moved. Normally, `route` is called immediately after `invalidate`.

```
public interface ConnectionRouter {
  ...
  void route(Connection connection);
  void invalidate(Connection connection);
}
```

Listing 6.2: Interface ConnectionRouter.

6.1 Bezier Routing

The basic idea of this algorithm is to draw a connection just like a person would do it on a piece of paper: Whenever a straight line is possible without any node intersections, just draw it. Otherwise draw a curve with as few direction changes as possible avoiding obstacles. This sounds quite easy to do for a person, for a computer it is however quite a complicated task. The following subsections describe the algorithm step by step.

6.1.1 Intial Routing

The algorithm works on a polyline which starts as a straight line from the start node to the end node and is improved stepwise. Figure 6.1 shows the initial routing. As the example of this chapter, the connection between `B0` and `B1` is illustrated.

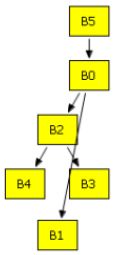

Figure 6.1: Initial routing.

If there are no intersections between nodes and the polyline, the algorithm ends here and sets the straight line as the routing.

6.1.2 Evade Obstacles

Now the algorithm iteratively tries to avoid intersections between the polyline and nodes. Currently there is an intersection with `B2` and `B3`. An additional bendpoint is inserted to avoid the intersection with `B2`, the resulting polyline is shown in Figure 6.2.

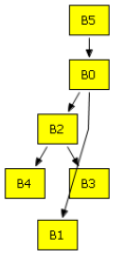

Figure 6.2: After first iteration step.

In the example, after evading block `B2`, there is still an intersection between the polyline and `B3`. Therefore the algorithm will insert additional intermediate points and the resulting line will avoid intersecting `B3` too (see Figure 6.3).

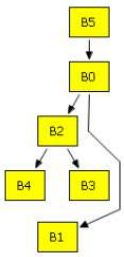

Figure 6.3: After second iteration step.

Points are added to the polyline until there are no more intersections or a fixed number of iterations is exceeded. The second case was introduced to avoid endless routing calculcations and to improve the performance. In most cases, ten intersections will not be exceeded. But how does the algorithm choose the additional points for avoiding intersections? Figure 6.4 shows all possibilities tried by this specific implementation. The rectangle in the middle stands for the obstacle that must be avoided. First, nine single points are sampled (left part of the figure), additionally four possibilities which add two points (right part) are tested.

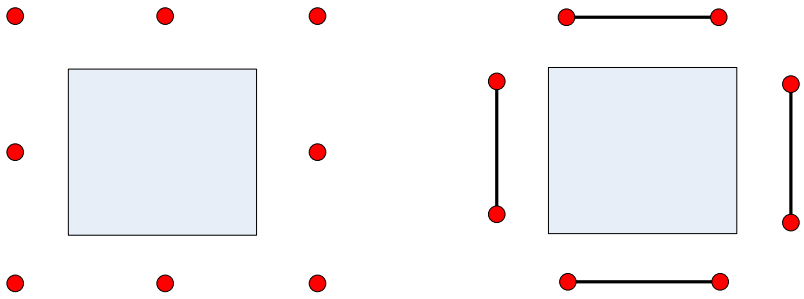

Figure 6.4: All possible evasion points.

For every possible evasion the costs are calculated. In this implementation, this is equal to the number of intersections between the polyline and any other line or node. As an additional criteria, the angles between the introduced points and their neighbors are taken into account. Note that customizing this cost function greatly affects the quality of the result. In Figure 6.3 for example the cost function is responsible that two points to the *right* of `B3` are inserted. Inserting two points to the *left* of `B3` would lead to an intersection with the edge going from `B2` to `B3`.

When counting the intersections with other edges, only the intersections with their approximation, a straight line between start and end node, are counted. This is done for performance reasons and because certain connections may be still unrouted while processing the current edge.

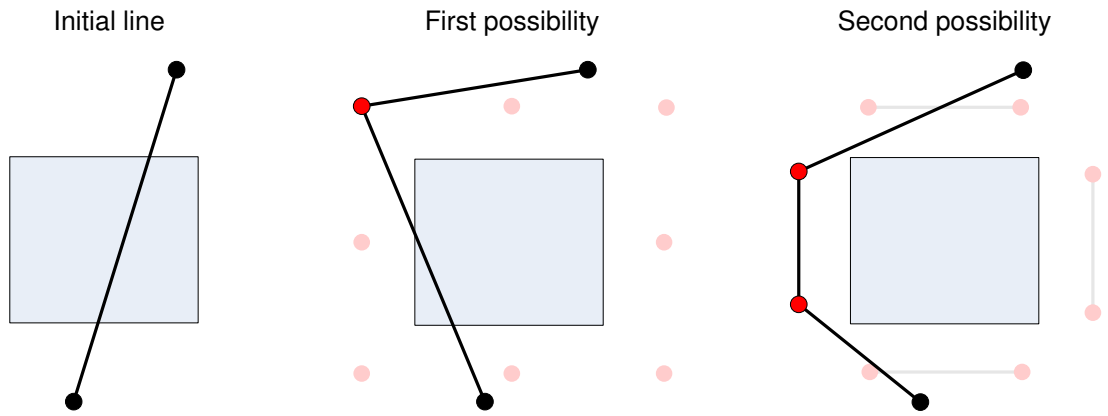

Figure 6.5: Two examples for adding intermediate points.

Figure 6.5 shows an example line and two possibilities to add extra points. In this case the algorithm selects the second possibility because in contrary to the first one the resulting polyline does not intersect the node.

6.1.3 Simplify Polygon

After evading obstacles many times, the resulting line can be more complex than necessary. Even in our simple example, after two iterations, the polyline contains one needless point. This is why there is an additional simplification step.

In this part of the algorithm, every possible pair of non-succeeding points is tested. Whenever there are no intersections between those points, all intermediate points can be deleted safely without degrading the result.

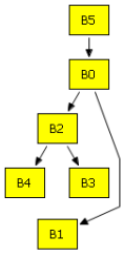

Figure 6.6: After simplification step.

Figure 6.6 shows the polyline after the simplification step. The point next to `B2` was removed. Now the line is simpler and does not have a bend to the left followed by a bend to the right.

6.1.4 Create Curves

Finally many intermediate points are added to make the polyline look like a smooth curve. Lots of graph visualization programs use b-splines, which approximate polylines. For this

algorithm it is however important that the curve goes through all points, otherwise many additional intersections could be introduced in this step. For this reason cubic Bezier curves are the better choice in this case. The equation for the approximation points of the cubic Bezier curve is as follows:

$$C(t) = P_0(1-t)^3 + 3P_1t(1-t)^2 + 3P_2t^2(1-t) + P_3t^3$$

In addition to the two endpoints P_0 and P_3, two interpolation points P_1 and P_2 are needed. Figure 6.7 shows how the application calculates those two points. For the whole curve to look smooth, it is important that each point and its two neighboring interpolation points lie exactly on a straight line.

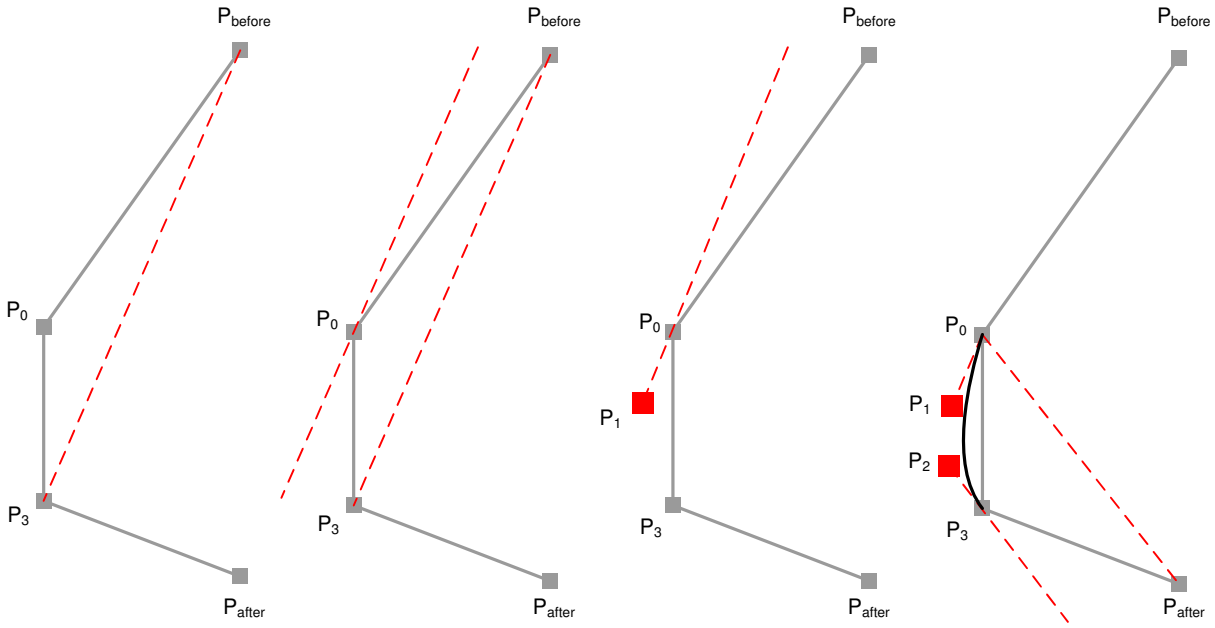

Figure 6.7: Calculating the two interpolation points.

This is ensured by putting the interpolation point of P_0 for example on a line which is parallel to the line going through the previous and the next point in the polyline (P_{before} and P_3). The line itself has to go through P_0, see the second image of Figure 6.7 for the result. The offset between P_0 and its interpolation point P_1 is calculated by multiplying the offset between P_0 and P_3 with a constant factor. A value of 0.3 for the factor seems to be appropriate. Figure 6.8 shows the example control flow graph after the Bezier algorithm made the polyline between `B0` and `B1` look smooth.

6.1.5 Results

This algorithm leads to good results in the common case, especially for small to medium sized methods. Very important is a good heuristic for the cost function to choose the right points when evading an obstacle.

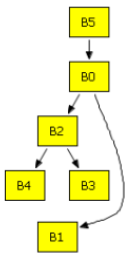

Figure 6.8: After Bezier creation.

6.2 Manhatten Routing

The term "Manhatten Routing" refers to a routing strategy where only orthogonal lines are allowed. An additional constraint for developing this algorithm was to ensure that incoming edges will always end at the top of a node and outgoing edges will always start at the bottom of a node. To avoid ambiguity, only incoming edges may share the same end point. Multiple outgoing edges must start at different locations. It is however rare that a block has more than three successors. Only in case of `switch` statements or a lot of exception handler blocks the number of outgoing edges can be high.

6.2.1 First Heuristic Approach

The first attempt to solve the Manhatten routing problem was similar to the way a person would search a specific location in an unknown city. At every coordinate the direction is greedily chosen taking into account the current and the wanted position. Additionally, points occupied by obstacles must not be entered. A matrix with an entry for each x- and y-coordinate specifying whether it is occupied is constructed to be able to check quickly for obstacles. It contains entries for each node as well as for already routed edges. Unfortunately, the results of this approach were not very good. The number of bends of a connection was ignored when finding a good solution, but it affects the result a lot. Additionally, in special cases there were also problems with endless loops. It is not guaranteed that greedy decisions do not lead to dead ends or circles.

6.2.2 Shortest Path Approach

A complete contrast to the heuristic approach is the algorithm which looks at the coordinate system as a graph. Finding the best routing between two nodes is reduced to finding the shortest path between them. There is a corresponding node in the coordinate graph for every point on the drawing area. The length of an edge between two neighbored coordinates is basically one, penalties for intersections with other edges or nodes are added. This leads to better results, also endless loops cannot occur. However the performance is bad. The drawing area can be quite large and even if edges are only allowed for example to be on even coordinates, there exist millions of virtual nodes making finding the shortest past quite slow.

Additionally the result is not "optimal", because the algorithm computes the best path for one edge completely separated from the other edges. So the global view of the graph is not very nice.

6.2.3 Final Algorithm

The first two algorithms are both dissatisfying, so a completely different way of solving the problem is needed. The key to a successful algorithm is to realize that edges with many bends look bad and are not needed in the normal case. In fact, if the nodes are placed by one of the three automatic positioning algorithms, an edge never needs more than four bends to connect its two nodes without intersecting any other node.

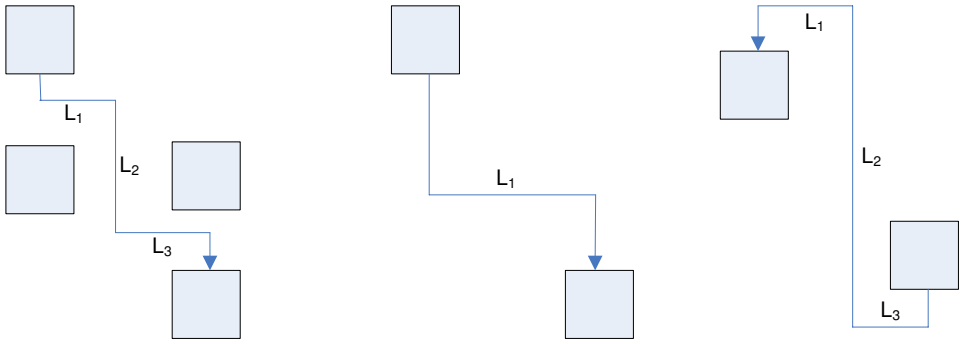

Figure 6.9: Possible positions of two connected nodes.

So limiting the amount of bends of a connection is an approximation step that does not affect the quality of the solution a lot. In fact the final algorithm is a bit similar to the Bezier routing algorithm. First a limited amount of routing candidates are chosen and their cost is calculated. Then the one with the lowest cost is chosen. Figure 6.9 shows how connections are simplified. The x-coordinates of L_1 and L_3 as well as the y-coordinate of L_2 are variable. When the destination node is above the source node, then the rightmost option is taken. Otherwise all candidates of the middle and the leftmost alternative are evaluated.

One important step of the algorithm is the selection of "good" candidates. It is obvious that not *all* integer coordinates can be tested, because in the left and the right case, this would even mean for small graphs (with a width and a height of approximate 1000) to try about 1.000.000.000 possibilities. This is unrealistic, but fortunately there is a good heuristic for choosing candidates.

The algorithm uses the fact that the layout of the graph will be placed automatically in most cases or at least only a few manual changes will be made. Therefore nodes will be positioned in rows making it less important to try a lot of possibilities for the two horizontal lines L_1 and L_3. Testing three possible y-coordinates for them should be enough for most graphs, because there are few nodes with more than three outgoing or incoming nodes. When choosing the x-coordinate for the vertical line L_2, basically one wants to take the middle between the start and the end position. If there are any intersections, possible obstacles should be avoided. So it is reasonable to test only two

x-coordinates to the left and the right of nodes lying between the start and the end node. Figure 6.10 graphically shows this strategy.

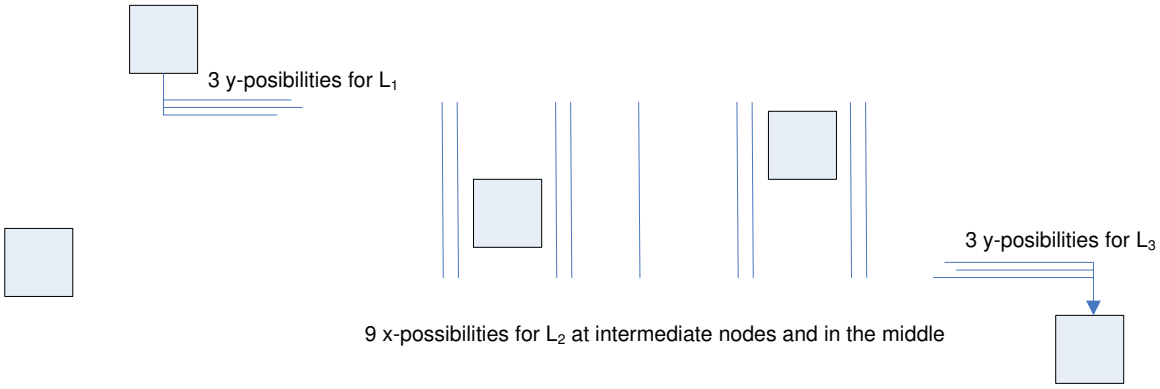

Figure 6.10: Choosing good candidates.

Now the number of candidate connections is reduced significantly even for large graphs to a few thousand. In the example shown in Figure 6.10 only 81 $(3 * 9 * 3)$ possibilities need to be evaluated.

There must be balance between the quality of the cost function and the number of candidates. On the one hand it is senseless to have a good cost function, but it cannot figure out a good solution because there are only bad candidates. On the other hand, having a huge amount of candidates, but a very simple cost function is not better. The first version of this algorithm tried every single x-coordinate for L_2, but had an elementary cost function for performance reasons. Additionally for L_1 and L_3 it only used the first row which is completely free of other horizontal lines. Unfortunately, this resulted in a rather bad result. Figure 6.11 shows the final algorithm on the left side and the former one on the right side. The example shows the method `String.indexOf` after using the BFS positioning manager. The left one is the better one, because of the following reasons:

- *Intersections:* There is only one intersection on the left side compared to nearly ten on the right side. The number of intersection is one of the most important criteria for evaluating graph drawings.

- *Bends:* The number of bends should also be low. For instance take the connection between `B21` and `B23`. The left algorithm simply draws a straight line which looks a lot better.

- *Close Lines:* There are a lot of sites in the right drawing, where two lines do not directly intersect, but are drawn close together. For example, the right algorithm draws the connection between `B6` and `B7` two times next to another line. On the left side, there is no such mistake.

- *Occupied Rows:* The left algorithm only draws a horizontal line, if the row is not occupied by any other line. This way all possible intersections between horizontal lines are prevented. However it does not look good if lots of different rows contain

lines. The connections between `B0` and `B1` and between `B0` and `B2` are not drawn using the same row on the right side. This makes them look worse than on the left side.

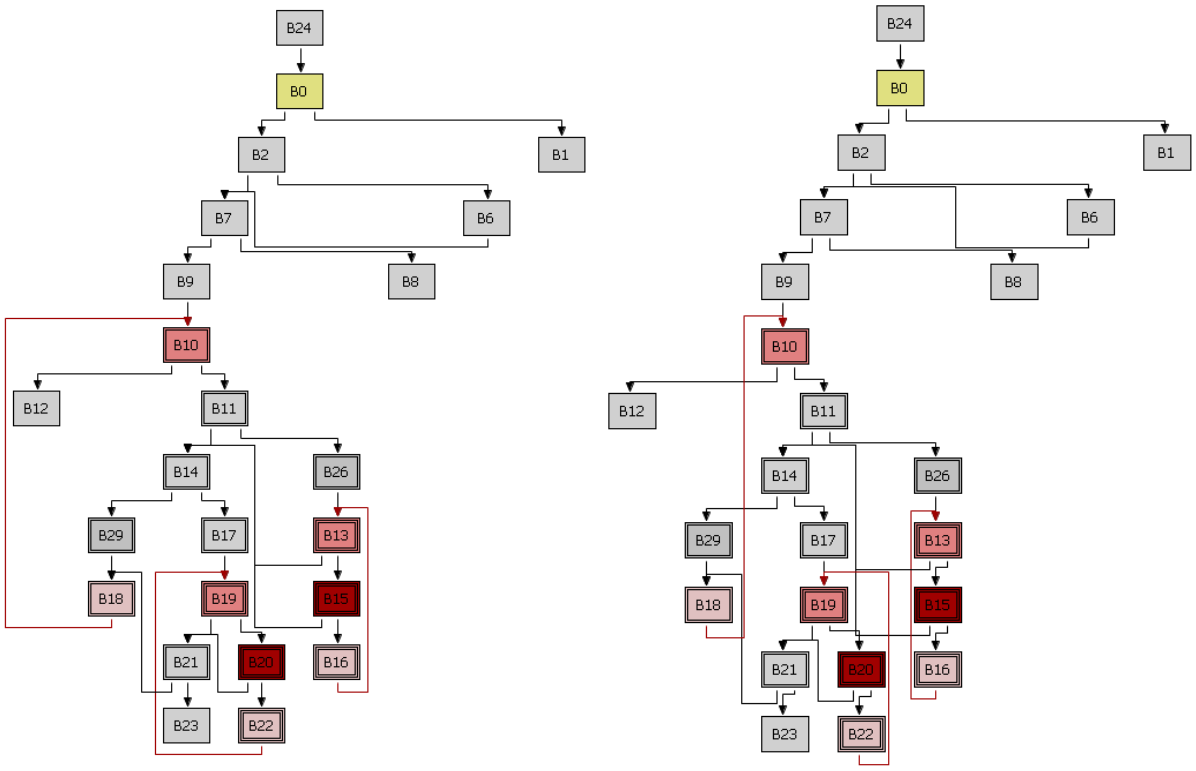

Figure 6.11: Comparing two different variants of the algorithm.

The cost function tries to take all possibly bad looking things into account and calculates a floating point value. Important is especially the relation between two costs. For example, is it better to have two close lines or an additional intersection? Fine tuning of the configuration is needed in order to get good results. The cost function of the algorithm takes the following things into account:

- Length of the line
- Number of bends
- Intersection area between the line and nodes
- Intersection area with other already routed connections
- Touch with already routed connections
- A bonus is granted when the line intersects lines with the same destination node.

In order to improve the result, after all connections are routed for the first time, they are routed again. This ensures that all other connections are already routed and can be used to calculate intersections and touches. It is possible that after the second run, a third run would give completely different results. However this is rather unlikely and normally two runs lead to quite good results.

Choosing the correct start point for a connection can have a high impact on the quality of the drawing. When a node has only one outgoing edge, exactly the middle of the node is chosen. The second common case is a node having two outgoing edges. Here the algorithm tries both possibilities: Either the first edge starts left and the second one right, or inverse. The possibility with the minimal sum of the routing costs for both edges is chosen. So the algorithm does not only look at a single edge, but at all outgoing edges of a specific node. For more than two connections, the strategy to try all possibilities would be too expensive. Instead the order of the outgoing edges is chosen heuristically using the x-coordinates of their destination nodes.

An additional optimization was introduced to make for example the straight connection between `B21` and `B23` possible. The algorithm checks whether the two end nodes of a connection have exactly the same x-coordinate. If this holds and there are no other nodes between them, a straight line is drawn without any further computations.

As the algorithm tries a huge amount of candidate connections and needs to check them for intersection with lines or nodes, performance is a big issue here. Constructing a matrix with an entry for every position on the drawing area, like the shortest path approach does, is however an order of magnitude too slow. Just checking all nodes and edges for intersection is not much better. The main problem is the big drawing area, but in fact when checking for intersection only a specific part of it is of importance. Therefore the algorithm divides the area into quadratic cells of a specific size. A value between 50 and 300 showed to be adequate. A cell has a list of elements which are currently in touch with it. This can either be nodes or segments of connections. When checking for intersection, only objects in cells touched by the current line need to be tested.

6.2.4 Results

After completely changing the strategy of the algorithms three times, the result is finally at least practical. A comparison with the Bezier algorithm is shown in Figure 6.12. The Bezier routing gives a better result than the orthogonal router, because the human eye likes curves rather than lots of right angles. Additionally the problems which make a drawing look bad listed in the previous subsection are far easier to avoid using round curves.

There are only few examples where the Manhatten router is the better choice. Some small and symmetric graphs can look better using orthogonal lines. Users of the Visualizer Application can easily switch between the two routing possibilities and choose the one they think is currently appropriate.

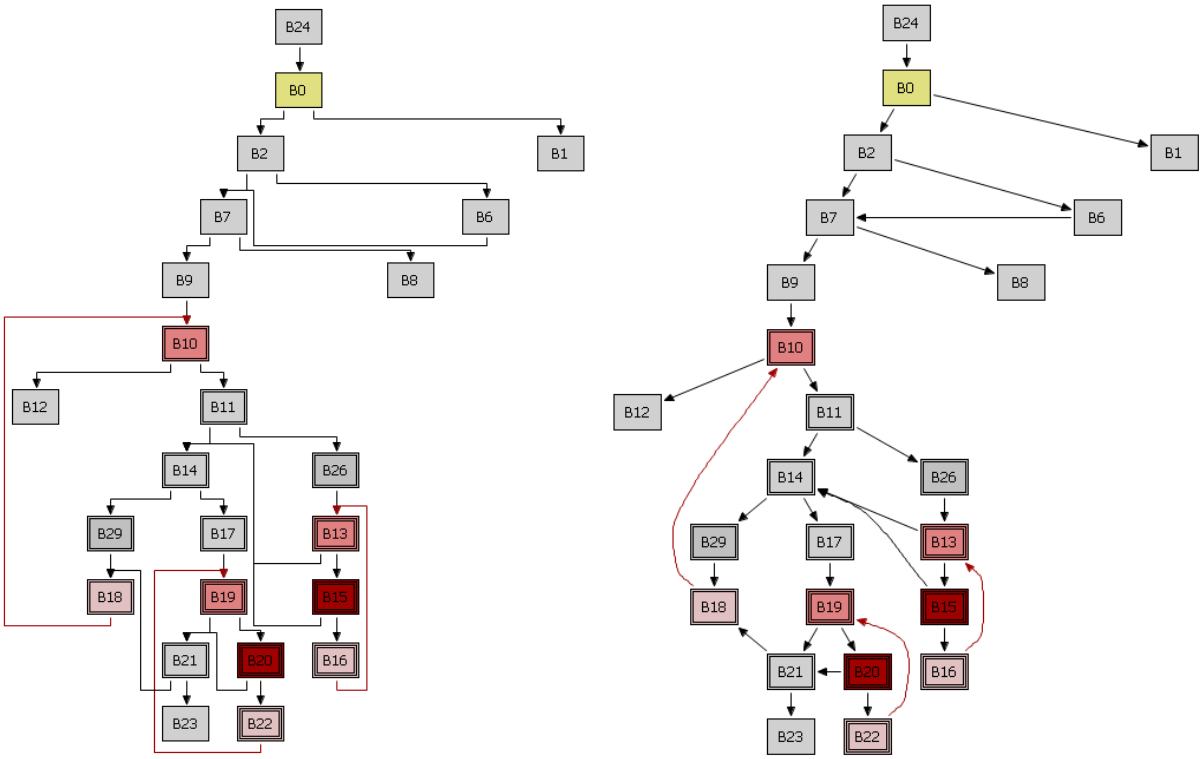

Figure 6.12: Comparison between orthogonal routing and Bezier routing.

Chapter 7

Conclusions

Working with the Eclipse Rich Client Platform simplifies a lot of tasks, especially because of the powerful drawing framework Draw2D. Also the management of preferences and the handling of user actions is quite intuitive. At the beginning of a project a lot of time must be spent to get familiar with these complex frameworks. However, especially for large or long-living applications this pays off heavily in the long run.

The most difficult algorithmic part of this project was the task to find a "good" positioning and a "good" routing algorithm. Most of the existing graph visualization solutions presented in section 1.1 simply draw a static view of the graph. The resulting graph of the Visualizer Application should not only look nice, but the positions of the nodes should also be modifiable by the user. This property makes the routing a lot more difficult because it cannot assume properties implied by certain positioning algorithms. In this application the routing is completely independent from a specific positioning. This is also a disadvantage, maybe the graph visualization could be improved if restrictions on the positioning of nodes were introduced.

When developing routing or positioning algorithms, it is often quite discouraging that the resulting algorithm does not give the expected results. It is difficult to appraise the expected performance of such an algorithm. While pre-estimating runtime speed or memory usage is a manageable task for a software engineer, the quality of the outcoming drawing is a rather unknown property. For the Manhatten routing algorithm, several versions were developed and thrown away until the final version was implemented. In the end, the results of the algorithm are acceptable, but still in most cases worse than the results of Bezier routing.

7.1 Possible Extensions

After the application was finished on top of the Eclipse Rich Client Platform, it was ported to the NetBeans Platform, which is also an open-source project but originally developed by Sun Microsystems. While the part of the application depending heavily on graphical user interfaces needed a lot of changes, the routing and positioning algorithms nearly stayed unmodified. There are plans to further extend the Visualizer Application by adding better views for the intermediate representations LIR and HIR. Currently the

IR Visualizer presented in Section 2.1.4 acts just like an advanced command line output. Because of the plugin architecture available in Eclipse and NetBeans, adding additional views or editors can be done easily without changing existing code.

New position managers and routers can be added easily to the CFG Visualizer: The correct interfaces must be implemented by the new class and an action needs to be created. Additionally, the model of the application is completely based on interfaces (see Section 4.2.2). Therefore the CFG Visualizer could be used to view other graphs, for example a data dependency graph.

List of Figures

Listings

Bibliography

[aiS06] *aiSee*. URL http://www.absint.com/aicall/. 2006

[AL04] Arthorne, John ; Laffra, Chris: *Official eclipse 3.0 FAQs*. Addison-Wesley, June 2004 (the eclipse series). – 386 S

[BETT99] di Battista, Giuseppe ; Eades, Peter ; Tamassia, Roberto ; Tollis, Ioannis G.: *Graph Drawing: Algorithms for the Visualization of Graphs*. Alen Apt, 1999. – 397 S

[Ecl06] *Eclipse.org home*. URL http://www.eclipse.org. 2006

[GB03] Gamma, Erich ; Beck, Kent: *Contributing to eclipse: Principles, Patterns, and Plug-Ins*. Addison-Wesley, October 2003 (the eclipse series). – 395 S

[Gra06] *Graphviz*. URL http://www.graphviz.org. 2006

[Ott06] *Otter*. URL http://www.caida.org/tools/visualization/otter/. 2006

[Sed03] Sedgewick, Robert: *Algorithms in Java*. Addison-Wesley Professional, 2003. – 768 S

[uDr06] *uDraw*. URL http://www.informatik.uni-bremen.de/uDrawGraph/. 2006