



JOHANNES KEPLER
UNIVERSITY LINZ

Research and teaching network

Thomas Würthinger

Visualization of Program Dependence Graphs

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

Master of Science
(Diplom-Ingenieur)

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, August 2007

Abstract

The Java HotSpot™ server compiler of Sun Microsystems uses intermediate graph data structures when compiling Java bytecodes to machine code. The graphs are program dependence graphs, which model both data and control dependencies. For debugging, there are built-in tracing mechanisms that output a textual representation of the graphs to the command line.

This thesis presents a tool which displays the graphs of the server compiler. It records intermediate states of the graph during the compilation of a method. The user can then navigate through the graph and apply rule-based filters that change the appearance of the graph. The tool calculates an approximation of the control flow to cluster the nodes of the graph into blocks.

Using a visual representation of the data structures speeds up debugging and helps understanding the code of the compiler. The thesis describes the code added to the server compiler and the Java application that displays the graph. Additionally, the server compiler and the NetBeans platform are outlined in general.

Kurzfassung

Der Java HotSpot™ Server Compiler von Sun Microsystems benutzt Graphen als temporäre Datenstrukturen beim Kompilieren von Java Bytecodes zu Maschinencode. Die Graphen des Compilers sind Programmabhängigkeitsgraphen, mit denen sowohl der Kontrollfluss als auch die Datenabhängigkeiten modelliert werden. Für die Suche von Fehlern kann eine textuelle Repräsentation der Graphen auf die Kommandozeile ausgegeben werden.

Diese Arbeit beschreibt ein Programm zur Anzeige der Graphen des Server Compilers. Bei der Kompilierung einer Methode werden Zustände des Graphen aufgezeichnet. Der Benutzer kann durch den Graphen navigieren und regelbasierte Filter anwenden, um die graphische Anzeige des Graphen zu verändern. Das Programm berechnet eine Annäherung des Kontrollflusses, um die Knoten in Blöcke zu gruppieren.

Die Verwendung einer graphischen Repräsentation der Datenstrukturen beschleunigt die Fehlersuche und hilft den Quelltext des Compilers zu verstehen. Die Arbeit behandelt den Quelltext, der zum Server Compiler hinzugefügt wurde, und die Java Anwendung, die den Graphen anzeigt. Weiters werden der Server Compiler und die NetBeans Plattform beschrieben.

Contents

1	Introduction	1
1.1	Class Diagram Legend	2
1.2	Related Work	2
2	NetBeans	4
2.1	Why NetBeans?	4
2.2	History	5
2.3	Modular Design	6
2.4	Filesystem	7
2.5	Lookup	8
2.6	Visual Library	9
3	Server Compiler	12
3.1	The Java HotSpot™ VM	12
3.1.1	Client versus Server Compiler	13
3.1.2	Java Execution Model	14
3.2	Architecture of the Server Compiler	15
3.3	Ideal Graph	16
3.3.1	Data Dependence	17
3.3.2	Empty Method	17
3.3.3	Phi and Region Nodes	18
3.3.4	Safepoint Nodes	20
3.4	Optimizations	21
3.4.1	Identity Optimization	21

3.4.2	Constant Folding	22
3.4.3	Global Value Numbering	22
3.4.4	Loop Transformations	23
3.5	MachNode Graph	24
3.6	Register Allocation	26
4	User Guide	28
4.1	Generating Data	29
4.2	Viewing the Graph	30
4.3	Navigating within the Graph	31
4.4	Control Flow Window	32
4.5	Filters	32
4.6	Bytecode Window	36
5	Visualizer Architecture	37
5.1	Module Structure	37
5.2	Graph Models	39
5.2.1	XML File Structure	40
5.2.2	Display Model	42
5.2.3	Layout Model	43
5.3	Properties and Selectors	45
5.4	Filters	46
5.5	Difference Algorithm	48
6	Hierarchical Graph Layout	50
6.1	Why Hierarchical?	50
6.2	Processed Steps	51
6.3	Breaking Cycles	53
6.4	Assign Layers	55
6.5	Insert Dummy Nodes	56
6.6	Assign Y-Coordinates	57
6.7	Crossing Reduction	58

6.8	Assign X-Coordinates	60
6.8.1	DAG Method	60
6.8.2	Rubber Band Method	62
6.9	Cluster Layout	64
6.10	Drawing of Backedges	65
6.11	Optimization for Large Graphs	66
7	Compiler Instrumentation	67
7.1	Overview	67
7.2	Identifying Blocks	68
7.3	Building Dominator Tree	70
7.4	Scheduling	73
7.5	Adding States	74
8	Conclusions	75

Chapter 1

Introduction

When compiling Java methods to machine code, the Java HotSpot™ server compiler of Sun Microsystems uses an intermediate representation that corresponds to a directed graph. Several nodes are added to the graph for every bytecode. Afterwards, transformations are applied to the graph with the goal to increase the execution speed of the method. After all optimizations are applied, the graph is converted to code that can be directly executed on the target machine. The graph is complex for large methods. It is difficult to understand the purpose of a certain node in the graph because of the high number of applied optimizations. Currently, developers use code that prints the graph on the command line when they are debugging the server compiler. This thesis presents a tool that helps the developer understand the graph by giving a visual representation of it.

The user can specify rule-based filters, which change the appearance of the graph. Different filters can be used when different properties of the graph are of interest. Navigation mechanisms are available, such that the user can focus on specific parts of a large graph. An additional feature of the tool is to display the differences between two graphs.

This thesis is divided into eight chapters. Alongside the introduction these chapters are: Chapter 2 describes the NetBeans platform in general. The program that displays the graph is based on the NetBeans platform. Some important concepts of NetBeans and the visual library of NetBeans are explained. Chapter 3 outlines the server compiler. The general architecture and the differences to the client compiler are presented. The focus of the chapter lies on the graph data structure and the optimizations applied by the compiler.

Chapter 4 is a user guide for the visualization tool. It explains how to connect the server compiler to the Java program. The navigation possibilities, the filters, the Control Flow Window, and the Bytecode View Window are described.

Chapter 5 presents the architecture of the Java application that displays the graph. The data models and the class structure are outlined. Chapter 6 is a description of the hierarchical layout algorithm used to find coordinates for the nodes of the graph and interpolation points for the edges.

Chapter 7 presents the C++ code added to the server compiler. This code is responsible for the scheduling and for saving the state of the graph during the compilation of methods. Chapter 8 describes the main difficulties during development of the tool and points out extension possibilities.

1.1 Class Diagram Legend

The class diagrams in this thesis follow the conventions shown in Figure 1.1. Interfaces are orange boxes with an italic name of the interface in it. Green boxes are classes which are part of a previously explained or external API. The connections of the current classes with them are part of the drawing. Generally, classes that are strongly related are grouped using a rounded rectangle with a dashed border.

For inheritance and composition, the standard UML symbols are used. When no cardinality is specified at the start or end of a composition, then the cardinality is 1. The blue arrow means that the source uses the destination of the arrow. A textual attribute classifies the relation further.

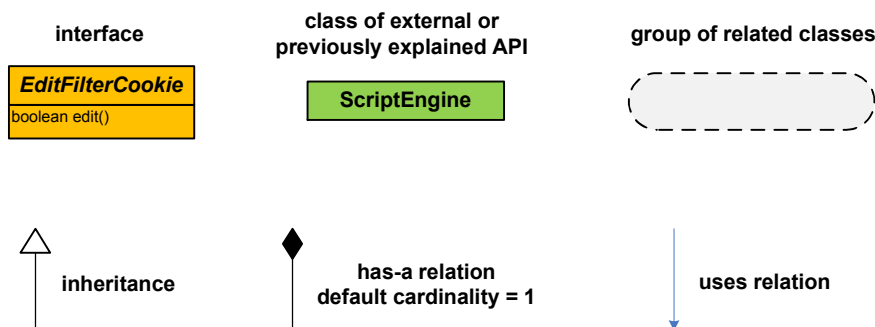


Figure 1.1: Conventions used in the class diagrams.

1.2 Related Work

A debugging tool for the HotSpotTM client compiler [17] visualizes three different data structures: The control flow graph, the data dependence graph, and information about the register allocation. The data is traced by the compiler in a textual format. In contrast to the tool presented in this thesis, a direct communication between the compiler and the application is not possible.

Stefan Loidl presents the data dependence graph visualizer of the tool [20]. It displays the data dependencies of the intermediate representation of the client compiler. In comparison to the graph of the server compiler, the data dependence graph of the client compiler is more sparse.

As most of the nodes have only few incoming edges, the tool does not need to define slots to distinguish between them.

The author's bachelor thesis [32] presents a visualizer for Java control flow graphs, which is also part of the client compiler visualization application. The graph is recorded at several stages during compilation. The control flow graph of the client compiler is simpler than the graph of the server compiler, because it contains only control flow dependencies and no data dependencies. Additionally, there is not a node for every instruction, but for every block of instructions. This significantly reduces the size of the graph. Therefore, some of the advanced navigation and filtering concepts are not necessary for the control flow graph.

Several software products can draw arbitrary graph structures. The development of a specific visualization tool for the server compiler has the advantage that the layout and the navigation is adapted to the needs of the graph of the server compiler. The following list presents three tools that can be used to draw graphs automatically. Features such as filtering or fast navigation within the graph are not available in these tools.

Graph Visualization Software (GraphViz) [13]: GraphViz is a group of open source programs that visualize directed graphs, which are specified in a textual format. The executable `dot.exe` is part of the GraphViz group and converts the textual representation of a directed graph into an image file. The hierarchical layout algorithm presented in this thesis is based on the algorithm used by GraphViz. The main purpose of GraphViz is not to interactively view the graph, but to produce a static image file for the graph. Enhancements to the GraphViz layout algorithm presented in this thesis are the cutting of edges and a second way to assign x-coordinates to the nodes based on the rubber band method. Additionally, backward edges are treated by the visualization tool in a special way.

aiSee Graph Layout Software [14]: aiSee is a commercial graph layout software that is a successor of the free tool Visualization of Compiler Graphs (VCG) [27] developed by Georg Sander. It is not specialized on hierarchical graph layout, but enables the user to choose from different layout algorithms including force directed layout. It supports clustering and folding of the graph. The tool uses a custom input format for the graphs.

uDraw [30]: The uDraw graph visualization software is developed at the University of Bremen and is specialized on hierarchical layout. One of the key features is that the user can, under some restrictions, manually change the layout after the automatic algorithm was applied.

Chapter 2

NetBeans

NetBeans [22] is an *integrated development environment* (IDE) written in Java. It is an open source project highly supported by Sun Microsystems. Although it is mainly designed to support developers in creating Java applications, it can also be used for C/C++ projects. Additionally, there are extensions available for NetBeans that allow to use the IDE also for completely different purposes like UML modeling, scripting in Ruby or Groovy, creating LaTeX documents, and so on. The visualization tool uses the NetBeans core libraries as a platform for building rich client applications with Java.

This chapter explains some important concepts of NetBeans that are used by the visualization tool. It gives a short overview of the NetBeans platform for software developers who are using NetBeans as the basis for their application [2]. If you are looking for a description of NetBeans as a development environment, see [21].

2.1 Why NetBeans?

Building upon a platform instead of using only plain Swing speeds up the development of a Java application and prevents developers from reinventing the wheel over and over again. How can an application benefit from using the NetBeans platform as an underlying layer? The following list introduces some useful aspects of the NetBeans library. The most important of them will be explained in detail in upcoming sections.

Module: NetBeans itself can be seen as a collection of Java modules that have well-defined dependencies. It is assured that only the modules currently needed are loaded. This improves memory usage as well as the startup time of an application. Additionally, developers are enforced to develop modular applications, which leads to a better design in general.

Window System: The built-in windowing system allows docking of components and supports tabbing of multiple documents. Additionally, actions that operate on the global selection

can be declared. Only using Swing means that either such functionality does not exist or it must be implemented by hand, highly increasing the total development effort.

Persistence: Configuration and serialization data is organized in virtual filesystems. When the NetBeans application is not running, the data is stored in a filesystem on the hard disk.

Visual Library: The NetBeans platform comes with a high-level drawing library. It is especially useful for the visualization tool as it is designed to draw graphs. It can add a large set of features to a drawing application for "free", at least for just adding a few lines of Java code. Examples for such functionalities are zooming, satellite view, and animation.

Java libraries with the same functionality that are not part of the NetBeans platform could be used, it is however more convenient if the libraries are directly integrated into the platform. This allows the libraries to work together without compatibility conflicts. Additionally, all NetBeans libraries take benefit of the module system, which manages lazy loading of the modules. The drawback of using a large amount of underlying libraries for a project is a higher development time needed at the beginning of a project, because the developer needs to get familiar with the libraries. However, for larger projects this additional cost pays off in the long run. Additionally, this cost needs not be paid when subsequent projects also take benefit of the same libraries. So building the first application on top of NetBeans means at first doing additional work, but the longer one uses the platform, the bigger are the advantages [2].

2.2 History

The first code for the system that evolved over more than a decade to the current version 5.5 of NetBeans was written in 1996. It was a student project, whose intention was to build an integrated development environment by using only Java code. At this time the program was called Xelfi [33]. For producing the screenshot of Xelfi shown in Figure 2.1, installing the old JDK version 1.1 was necessary. The NetBeans of today and Xelfi have only few things in common, but some of the basic design concepts have never changed since the early days. Among them are the modular design and the concept of virtual filesystems. Xelfi soon became a success and therefore a company named after the IDE was founded. During these days the current name of NetBeans was introduced: One of the business ideas was to develop *network-enabled JavaBeans*.

In 1999, Sun Microsystems, the founder of the Java programming language, acquired the company. The company was interested in NetBeans and so the product forms their flagship Java IDE until nowadays. Sun soon realized that the growth of NetBeans can be accelerated by building a development community around it, instead of distributing it as a commercial product. Therefore, they open-sourced the whole IDE in 2000. After this step, people started using NetBeans not only as an IDE, but also as a library to build their own applications. This brought up the idea of a rich client platform.

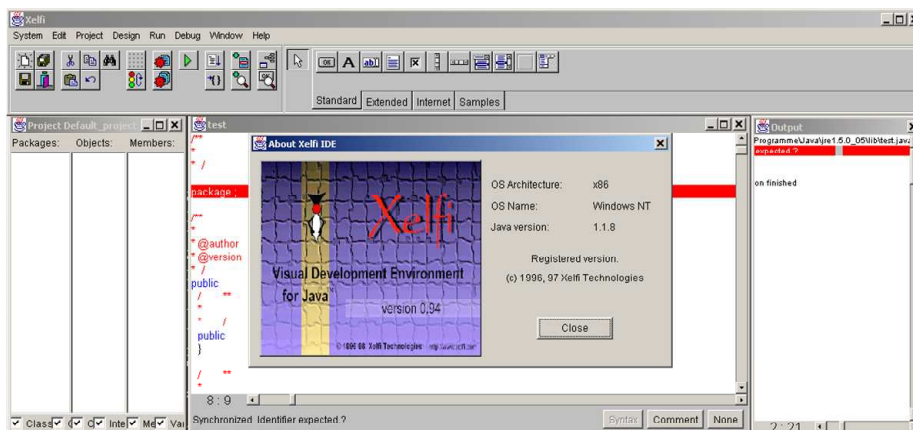


Figure 2.1: Screenshot of Xelfi, the ancestor of NetBeans, running with JDK 1.1.

The number of NetBeans users grows steadily. The current stable version of NetBeans is 5.5, but there is already a pre-release version of NetBeans 6.0 available. The development of the visualization tool started with NetBeans 5.5, but later on it was ported to NetBeans 6.0. [1]

2.3 Modular Design

NetBeans applications consist of separate modules working together to form one big program. The IDE itself is a set of NetBeans modules that support developers at programming in Java. There are some official extensions available like a profiler, special support for mobile application development, and C/C++ programming. Various modules developed by other companies can also enrich the IDE. The NetBeans platform consists of a set of modules that manage the co-operation between the modules and also provide some basic concepts regarding data storage and the user interface.

A NetBeans module is defined by a JAR file with additional information in the manifest. It has a version and specifies on which modules it depends, e.g. which modules need to be available for running this module. Modules can be enabled and disabled while the application is running. Such components are also called *plug-ins* as they resemble a plug. In NetBeans, the term plug-in is reserved for a collection of modules that are deployed as one unit.

Each module has a custom classloader, which searches for classes only in the standard Java classpath and in the modules that are listed as dependencies. The minimum version of a required module is defined when declaring dependencies. Additionally, there must not be any cyclic dependencies. A module must explicitly declare which packages are accessible by other modules. The usage of public classes declared outside of these declared packages is not possible. A lazy loading mechanism for the modules helps reducing memory usage and startup time.

2.4 Filesystem

One of the base concepts of module interaction in a NetBeans application uses virtual filesystems. A module can define an XML layer file to add declarative data to the *system filesystem*, i.e. a virtual filesystem that is shared among all modules. At startup, the filesystems of the individual modules are merged into the system filesystem. Entries in the filesystem can be directories, virtual files or pointers to real files. Virtual files consist of a name and a set of key-value pairs that are defined in the XML layer file.

Listing 2.1 shows an example layer file describing the filesystem of a module. It is a simplified version of one of the layer files used by the visualization tool. An application can use the system filesystem for example to register windows or to add actions to the toolbar and the menu bar.

Actions are registered as files in the filesystem in the folder `Actions`. The module responsible for instantiating the action objects scans through this folder. The name of a file specifies the class that represents the action, in this example the class `ImportAction`. An action can be registered as a menu item by adding an entry to the folder `Menu`. As the import action should appear in the file menu, it is added to the subfolder `File`.

There should only be one instance of class `ImportAction` in the system, so we use a shadowing mechanism that functions similar to link files. The extension `.shadow` specifies that the file points to another file and the attribute `originalFile` specifies the destination of the pointer. There is also a mechanism available for hiding files. To remove the standard open menu item from the file menu, we simply hide the file that defines that menu item by declaring a file with the extension `.instance_hidden`.

Listing 2.1 An XML layer file defining an action and hiding a menu item.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE filesystem PUBLIC "-//NetBeans//DTD Filesystem 1.1//EN"
"http://www.netbeans.org/dtds/filesystem-1_1.dtd">
<filesystem>
  <folder name="Actions">
    <folder name="File">
      <file name="at-ssw-ImportAction.instance"/>
    </folder>
  </folder>
  <folder name="Menu">
    <folder name="File">
      <file name="at-ssw-ImportAction.shadow">
        <attr name="originalFile" stringvalue=
          "Actions/Edit/at-ssw-ImportAction.instance"/>
      </file>
      <file name="org-netbeans-modules-openfile-
        OpenFileAction.instance_hidden"/>
    </folder>
  </folder>
</filesystem>
```

The filesystem is also used to save the state of a NetBeans application after shutdown. A subdirectory of the user directory is used to save the data. In this case the filesystem is not represented by an XML file, but by a directory structure that is physically present on the hard disk. At startup, the user-specific filesystem is merged with the filesystems of the modules.

2.5 Lookup

Another mechanism that is specific to NetBeans is the concept of *lookup*. The idea behind lookup is to change the set of interfaces that an object provides during program execution. In Java it is only possible to declare interfaces for classes at compile time. This set cannot be changed later on. It is also impossible that a certain object of a class implements an interface and another one does not.

The interface `Lookup` defines functions that return a collection of objects compatible to the type specified as a parameter. This way it is possible to ask the `Lookup` object if it can provide an implementation of a certain interface. The object can return itself or any other existing or newly created object, the only restriction is that the returned object must implement the interface.

An example for the use of the lookup mechanism is how the save menu item and an editor of a file interact. The save menu item does not know how to save a certain file type. Everything it needs to do is checking whether it is currently possible to save a file and trigger the save process when the menu item is clicked.

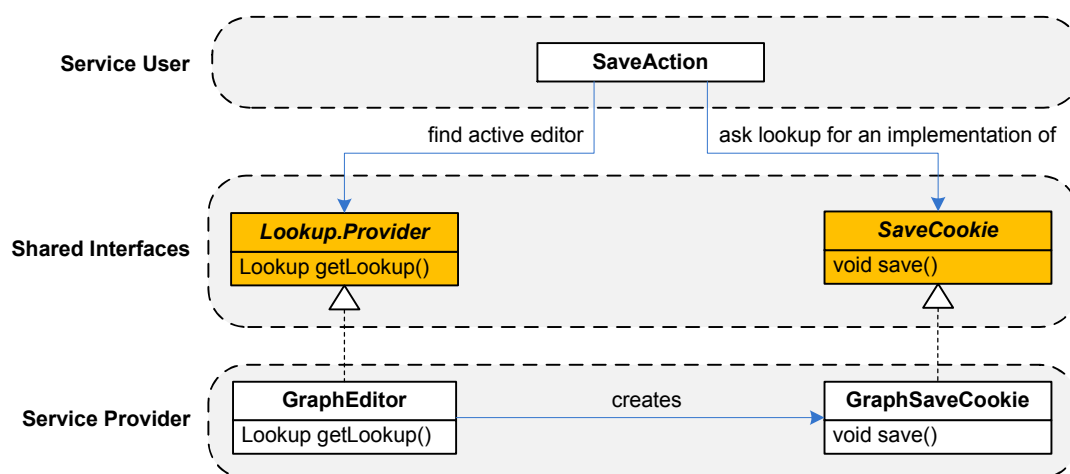


Figure 2.2: Using lookup, there is no dependency between service user and service provider.

Figure 2.2 shows how the classes are related. The interface `Lookup.Provider` is part of the standard NetBeans API and is implemented by objects that provide a lookup. In this case, the service is defined by the interface `SaveCookie` with a method that can be used for saving.

The `SaveAction` object first retrieves the lookup of the active editor and asks for an object of kind `SaveCookie`. When the editor cannot provide the service, it returns `null` and the menu item is disabled. Otherwise it returns an object that implements the interface and that can be used by the menu item in case the user clicks it.

The figure also shows how the classes can be separated in three different modules. One module just contains the declaration of the service. Provider and user of the service only need to depend on this API module and need no dependencies among each other. While the service user and the service provider are able to work together, none of them depends on the other.

There are several additional classes that enrich the lookup functionality. It is possible to monitor the lookup of an object by installing listeners. The class `ProxyLookup` allows to combine the lookups of several objects into a single one. A list component, for example, proxies the lookup of the currently selected nodes. So a pattern similar to the save mechanism can be used for any action that works with the elements of the list. The action declares the interface that an object must provide that the action can work. It will get enabled and disabled depending on the current selection of the list.

The NetBeans platform predefines two global lookup objects. One can be reached by calling `Lookup.getDefault()` and represents the global system lookup. The other one is often used by actions that depend on the current active window. It can be accessed using the method `Utilities.actionsGlobalContext()`. It proxies the lookup of the window that is currently focused. When the user activates another window, this lookup is changed. Reacting on changes can be done by adding listeners to lookup objects. The visualization tool uses this mechanism to always display the properties of the selected objects of the currently active window in the Properties Window.

The `Node` class is closely related to the lookup mechanism. A node can have an unlimited number of child nodes but only one parent node, so they form a tree-like structure. The children of a node are only accessed when the node is expanded by the user. There are several components such as a treeview or a list that are able to use such a tree of nodes as their model. These components provide a proxy lookup that combines the lookups of the currently selected nodes. The visualization tool uses the Node API in the Outline and in the Bytecode Window.

2.6 Visual Library

The NetBeans visual library is a high-level graphical framework built on top of Swing and Java2D. It is designed to support applications that need to display editable graphs such as UML diagrams. The library can also be used by applications that are not built upon the NetBeans platform.

Figure 2.3 shows a class diagram of the most important classes of the visual library and how they interact. Graphical components are called *widgets* and are organized in a tree hierarchy. The topmost widget is always a *scene*. This widget forms the bridge between Swing and the visual library. A scene can create a `JComponent` object that displays the contents of the

scene. In contrast to Swing components, widgets need not have a rectangular shape. There is a predefined mechanism for drawing a connection between two widgets. A connection has a source and a target anchor that are both related to some widget.

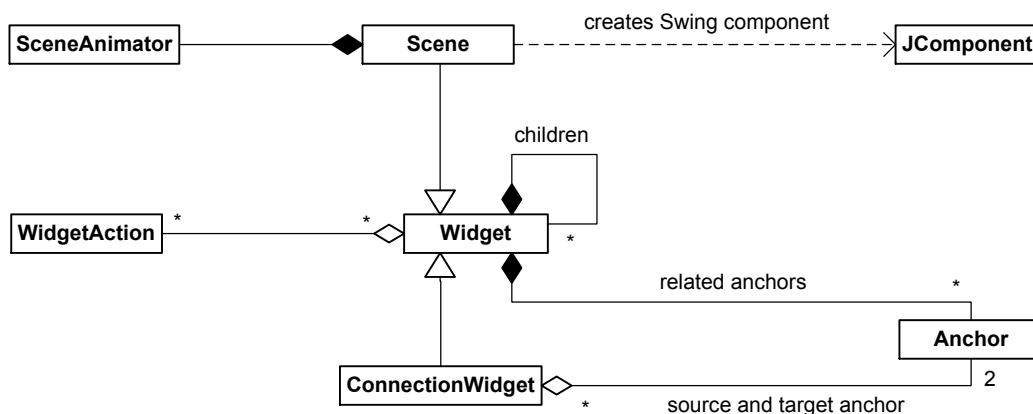


Figure 2.3: Class diagram of the NetBeans visual library.

A widget has an *action map* associated with it, i.e. a list of objects of type `WidgetAction` that can react on GUI events. There are a lot of predefined actions that automatically perform for example moving, selecting or resizing of a widget triggered by user input. The built-in animation mechanisms can be used to move widgets smoothly and to let them fade in or out.

Additionally there are some high-level functions built into the `Scene` class that allow zooming with different levels of details and the automatic construction of satellite views. Listing 2.2 shows the power of the Visual Library in an example. The result is a label that can be zoomed and that changes its background color when it is double clicked. A screenshot of the resulting program is shown in Figure 2.4. The NetBeans libraries that are needed for running this application are `org-openide-util.jar` and `org-netbeans-api-visual.jar`.



Figure 2.4: Screenshot of the visual library example program during execution.

First, the application creates a `Scene` object and adds a `WidgetAction` that allows the user to zoom using the mouse wheel. It constructs a `LabelWidget` and adds it as a child to the scene. Then it constructs a `WidgetAction` that calls an `EditProvider` when a widget is double clicked. This action is added to the label. An action can be added to any number

of widgets. As the scene itself also represents a widget, the scene itself would also change its background on double click when the example action was added to it. Finally a swing `JFrame` component is created, and the view of the scene is added to this window.

Listing 2.2 Java source code of a visual library program with a label widget and an action.

```
public class VisualExample {

    // Create edit action
    WidgetAction editAction = ActionFactory.createEditAction(
        new EditProvider() {
            public void edit(Widget w) {
                w.setBackground(Color.RED);
            }
        }
    );

    public static void main(String[] args) {

        // Create scene object and assign zoom action
        Scene s = new Scene();
        s.getActions().addAction(ActionFactory.createZoomAction());

        // Create label widget
        LabelWidget l = new LabelWidget(s, "Hello world!");
        s.addChild(l);
        l.setOpaque(true);

        // Add action to label
        l.getActions().addAction(editAction);

        // Create swing frame
        JFrame f = new JFrame();
        f.setSize(200, 100);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Add scene view
        f.add(s.createView());
        f.setVisible(true);
    }
}
```

Chapter 3

Server Compiler

The visualization tool improves the abilities to analyze internal data structures of the Java HotSpot™ server compiler [23], which is part of the Java HotSpot™ Virtual Machine of Sun Microsystems. A virtual machine (VM) acts as a bridge between a program and the operating system. The primary purpose of VMs is to enable the creation of platform-independent applications. In the beginning of this chapter the Java HotSpot™ VM is described in general, later on the main data structure of the server compiler and some of the most important optimization steps are presented.

3.1 The Java HotSpot™ VM

The Java HotSpot™ VM is a virtual machine developed by Sun Microsystems that implements the Java Virtual Machine Specification [29]. Figure 3.1 shows the main components of this VM. Basically Java methods are executed by the interpreter. When a method is invoked a specific number of times, the just-in-time compiler produces machine code for the method. Later calls of the method jump to the compiled machine code and will therefore run faster. The reason why a method is not immediately compiled to machine code at the first execution is that most Java methods are executed so infrequently that compiling them does not pay off. Depending on whether the virtual machine is started with the flag `-server` or not, the server compiler or the client compiler [15][18] is chosen to do the compilation task.

There are some cases in which the compiled machine code of a method can no longer be used and execution continues in the interpreter. Such cases occur when a compiler makes an optimistic assumption to produce faster code. When the assumption is later invalidated, e.g. because of dynamic class loading, the machine code is no longer usable. Jumping from the interpreter to the compiler and vice versa is not only possible at the invocation of a method, but also during the execution. Reverting back from the compiled machine code to the interpreter can be done at specific points of a method called *safepoints*. This process is called *deoptimization*.

There is also an opposite of deoptimization called *on-stack-replacement*. Imagine a method that is executed only once but consists of a long-running loop. Running the whole loop in the interpreter would heavily decrease execution speed. Therefore, the interpreter does not only count the invocations of a method, but also how often a backward jump occurs. When this counter exceeds a specific threshold, the method is compiled with a special on-stack-replacement entry. At this entry, machine instructions for loading the current values from the interpreter are inserted.

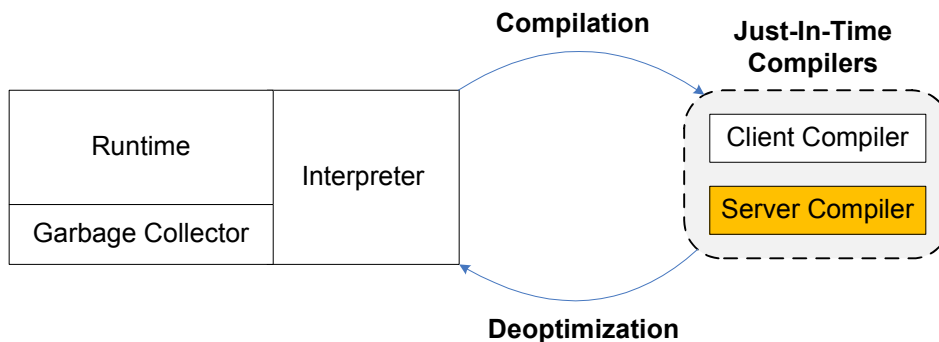


Figure 3.1: Architecture of the Java HotSpot™ Virtual Machine.

3.1.1 Client versus Server Compiler

The difference between client and server compiler is that the client compiler focuses on high compilation speed, while the focus of the server compiler lies on peak performance. The client compiler performs only a limited set of optimizations and is best-suited for short-running client applications. The server compiler needs more time for compilation, but produces more optimized machine code, so the compiled Java methods will execute faster. Therefore it is best for long-running server applications. Currently there are some efforts to allow *tiered compilation*. This means that methods are first compiled using the client compiler and only very important methods of a Java program are later on recompiled using the server compiler.

Internally, the client compiler uses a control flow based representation of the Java code to perform optimizations. The instructions are grouped to blocks where all instructions are executed sequentially if no exception occurs. The server compiler, by contrast, uses a program dependence graph [10], where data dependence and control dependence are both represented by *use-def edges*, i.e. edges pointing from the use of a value to its definition. This allows more sophisticated optimizations spanning over larger regions of a method, but the data structure is also more complex. The visualization tool helps understanding this program dependence graph. At a late stage during compilation, the nodes of the program dependence graphs are scheduled in blocks.

3.1.2 Java Execution Model

The Java HotSpot™ VM follows strictly the Java Virtual Machine Specification [29] when executing a program. Java source code [28] is first compiled to Java bytecodes. The virtual machine reads the bytecodes and executes them according to the specification. In the bytecode language, the state of a method consists of a set of local variables and an operand stack. All operands work on the stack. There are load and store bytecodes to transfer a value from a local variable to the top of the stack and vice versa. The execution model is specified for this stack-like language. Figure 3.2 shows a Java method with a single instruction and the resulting bytecodes of the method when it is compiled using `javac`. The interpreter maintains the current state that consists of the values of the local variables, the operand stack and monitors used for locking. It steps through the bytecodes and updates the state accordingly.

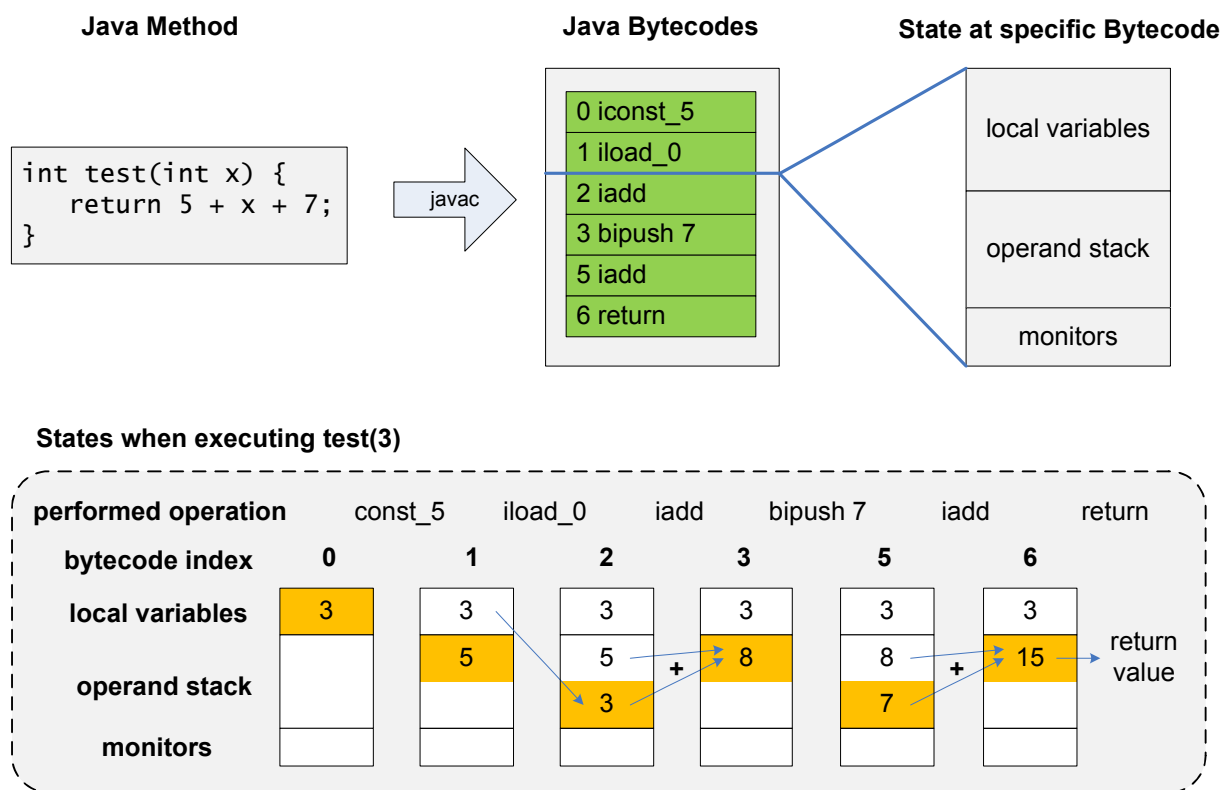


Figure 3.2: States during the execution of an example method.

The lower part of Figure 3.2 shows the states of the interpreter when the method is invoked with the value 3 as the argument. As this method has no synchronization code, there are no monitors in the state. The size of the array of local variables and the maximum stack size are both known for each method before invocation. First, the constant 5 and the parameter value, which is 3, are pushed onto the stack. The add operation pops the two topmost elements and pushes their sum

onto the stack. Afterwards, the constant 7 is pushed and again an add operation is performed. The returned result is the topmost stack element at the end of the execution of the method.

When the interpreter is given a correct state for a bytecode, it can continue the execution in the middle of a method. This property is used by deoptimization. The registers and memory locations from which the current state can be reconstructed are tracked by the compiler. When it wants to deoptimize at a specific location, it inserts the statements that construct the interpreter state and call the interpreter. Internally, the server compiler works with a program dependence graph instead of stack operations. While constructing the graph, the compilers maintain which nodes correspond to the current value of the local variables and the elements on the stack.

3.2 Architecture of the Server Compiler

Figure 3.3 shows the steps applied by the server compiler when processing a method. The compiler starts with an empty graph and adds nodes to it while parsing the bytecodes. Whenever a node is added, it performs locally the optimizations identity, global value numbering [4] and constant folding (see Section 3.4). Afterwards it cleans up the graph by properly building the method exits and performing dead code elimination.

The next steps are global optimizations applied to the graph. They are not mandatory and can be skipped by the compiler. After applying an iterative global value numbering algorithm, the ideal loop step is performed at most three times. The ideal loop phase is capable of doing loop peeling, loop unrolling, and iteration splitting (for range check elimination). When major progress is made running the ideal loop phase, it is run again, otherwise the compiler continues with the next step. Conditional constant propagation is an optimization that combines simple constant propagation with the ability to remove `if` statements when the result of their condition is constant. Then iterative global value numbering and several ideal loop phases are performed again.

The ideal graph is then converted to the more machine specific `MachNode` graph (see Section 3.5). Basic blocks are built from the control dependencies. For every node, the latest and earliest possible scheduling is computed satisfying the property that it must be scheduled after all its predecessors and before all its successors. The chosen location of a node should be late to avoid unnecessary computations that are never used, but it should be outside of loops whenever possible. A graph coloring register allocation (see Section 3.6) is performed. After some peephole optimizations, the final machine code is generated from the `MachNode` graph.

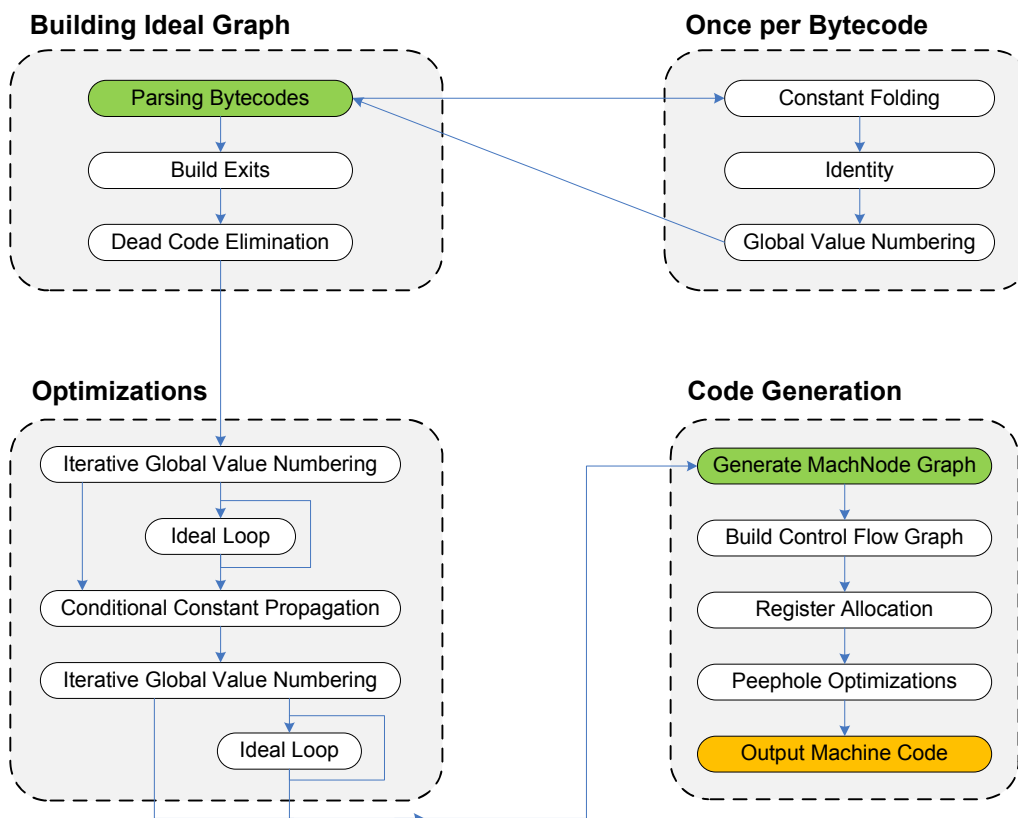


Figure 3.3: Architecture of the Java HotSpot™ server compiler of Sun Microsystems.

3.3 Ideal Graph

The representation of the program in the compiler highly affects the complexity and effectiveness of applied optimizations. A common representation of a program is a *control flow graph*. The source code is a flat sequential structure with an exactly defined order of the instructions. An instruction is defined by an operator and operands that are previously defined instructions. The control flow graph groups instructions that are guaranteed to be executed consecutively into *basic blocks*. At the end of every basic block there is a conditional branch or a jump. A basic block is connected with its predecessors and successors regarding control flow.

The Java HotSpot™ server compiler uses a control flow representation in the later stages of compilation. For most of its optimizations, however, it uses a data structure that combines control flow and data dependencies. This graph data structure is called *ideal graph*. The instructions are not ordered, but form a graph where the edges denote either definition-use data dependencies or control dependencies. By handling control and data dependence more uniform, some of the optimization steps, especially those involving code motion, are less complex. Implementation details of the graph are described in [8].

The program dependence graph in the server compiler is a graph data structure with lightweight edges. An edge in the graph is only represented by a C++ pointer to another node. A node is an instance of a subclass of `Node` and has an array of `Node` pointers that specifies the input edges. The advantage of this representation is that changing an input edge of a node is fast.

3.3.1 Data Dependence

Figure 3.4 shows a part of the program dependence graph generated by the compiler when processing the expression $p * 100 + 1$ where p denotes a parameter of the current method. Nodes are represented by filled rectangles with the type of the node and some additional information in them. Edges are drawn without arrows, but they always start at the bottom side of a node and end at the top side of a node. The layout arranges the nodes so that most edges are going downwards. Every node has a fixed number of input slots that can optionally be used as an end point of an input edge. There are some special nodes that allow an arbitrary number of input edges. These additional edges are always stored after the obligatory input slots.

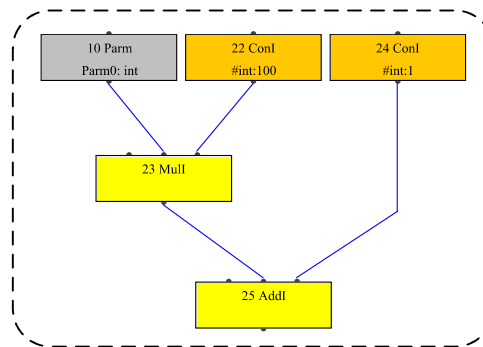


Figure 3.4: Program dependence graph when processing $p * 100 + 1$.

The operations are represented in the graph by nodes that are connected with the operands. The `MulI` and the `AddI` node both take two integer operands. They have three available slots, but the first one is not used in this example. Parameters are accessible via the `Parm` node, the additional information `Parm0: int` indicates that it is the parameter with index 0 and that it is of type `int`. The constants 100 and 1 are also represented as nodes.

3.3.2 Empty Method

Figure 3.5 shows the graph of an empty method. Every graph has a `Root` node and this node is always connected to the `Start` node. To make traversing the graph simpler, nodes at which the method is exited have an outgoing edge to the `Root` node. A node produces exactly one outgoing value, so the outgoing edges have no particular order. Projection nodes like `Parm` are used to model nodes that produce tuples. The `Start` node produces the following values:

Control: The control flow is modeled as edges just like data dependencies. The semantic is however different. The graph formed when all non-control edges are removed can be viewed as a petri net. When the method is executed, the control token passes along the control edges from node to node. An `If` node has two projection nodes as successors. The control token uses one of the two ways.

I_O: This type exists for historical reasons. It is used to serialize certain instructions.

Memory: To serialize memory stores that could interfere with each other, a type to express memory dependencies is used.

Frame Pointer and Return Address: Projection nodes that represent the value of the frame pointer and the return address. They are produced by the `Start` node and are mostly hidden to simplify the graph.

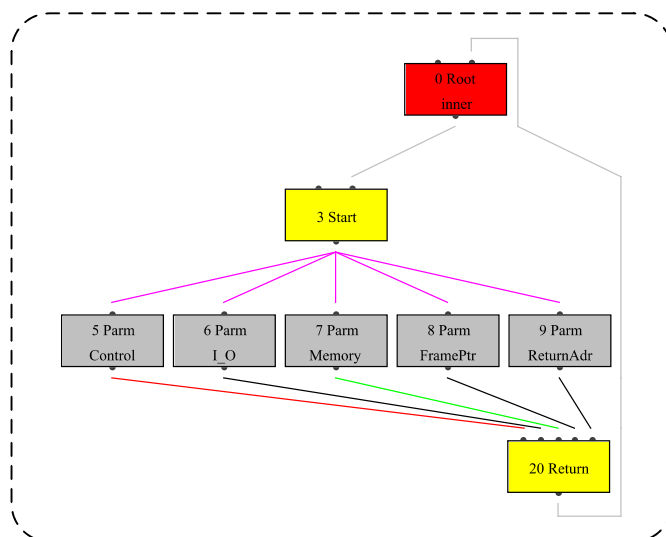
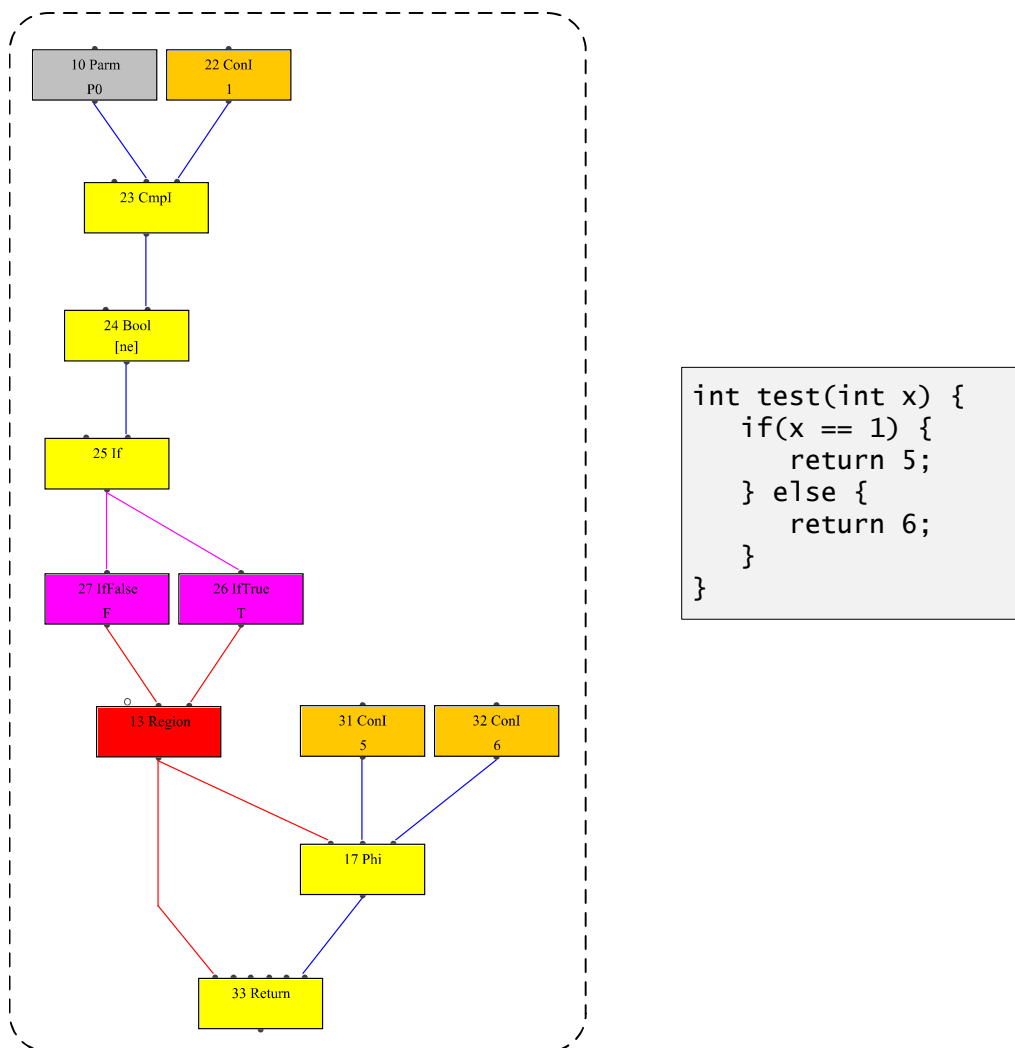


Figure 3.5: Graph when processing an empty method.

3.3.3 Phi and Region Nodes

The ideal graph is in *static single assignment* (SSA) form [9]. This means that a value is assigned only once to a symbol at its definition and is never changed. To model conditional assignment, e.g. if a variable gets assigned different values in different control flow paths, `Phi` nodes are necessary. They merge values from different control flows. In the ideal graph they are always connected to `Region` nodes, which merge the control flow. `Region` nodes are usually inserted at the end of `if` statements or at the loop header. The first input of a `Phi` node is always connected to its corresponding `Region` node. The other inputs specify the values selected for each control flow going into the `Region` node.



```

int test(int x) {
    if(x == 1) {
        return 5;
    } else {
        return 6;
    }
}
  
```

Figure 3.6: Graph when processing an if statement.

Figure 3.6 shows the Java source code and the graph representation of a method containing an if statement. The CmpI node compares the parameter and the constant 1. The Bool node is related to the CmpI node and specifies the compare operator, in this case the unequal operator is used. The If node splits control flow into a true and a false path. These two paths are merged by the Region node. The value of the Phi node is in dependence of the taken control flow either the constant 5 or the constant 6. The small circle above the first input of the Region node indicates that this first input is connected to the region node itself. Every Region node is connected to itself, which makes the block finding algorithms easier. The order of the inputs of the Region and Phi nodes is essential. A Phi node gets the value of its nth input when the control path corresponding to the nth input of the Region node is taken.

3.3.4 Safepoint Nodes

At the safepoints of a method execution can jump back to the interpreter as explained in Section 3.1. All elements of the operand stack and the values of local variables must be restored from the registers and the machine stack. In the graph, such points are called `SafePoint` nodes. In addition to the first five values produced by the `Start` node (`Control`, `I_O`, `Memory`, `Frame Pointer`, and `Return Address`), they have an incoming edge for every stack element and for every local variable. A `SafePoint` node also stores the bytecode index (`bci`), where the interpreter can continue execution.

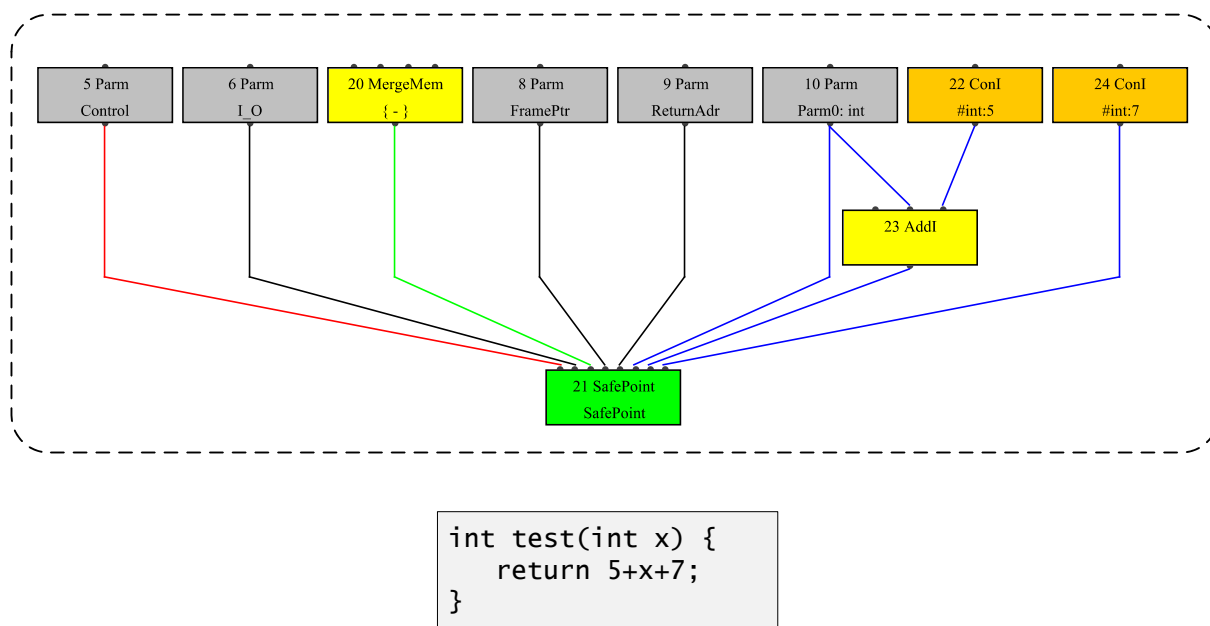


Figure 3.7: `SafePoint` node after parsing `5+x+7`.

Figure 3.7 shows a safepoint node during parsing of the example method `test`, also listed in Figure 3.2. The snapshot of the graph was taken after the `bipush` bytecode. The interpreter could resume with the second `iadd` bytecode, so the `bci` of the safepoint is 5. The first three inputs of the safepoint node specify the control, `I_O`, and memory dependence. The frame pointer and the return address are also needed by the interpreter. After these five standard input slots start the slots for the local variables. The method has exactly one local variable and at bytecode index 5, its value is equal to the value of the parameter of the method. The expression stack is formed by the next two inputs: One edge comes from the `AddI` node, the other from the constant value 7. While processing the bytecodes, the safepoint inputs are used to lookup the values of the local variables and the expression stack. The safepoint inputs are updated according to how a bytecode affects the local variables and the stack.

3.4 Optimizations

While building the graph by processing the bytecodes, local optimizations are applied. After adding a node to the graph the compiler checks whether the newly added node can somehow be replaced by another node that does the same computation in a cheaper way. There are three such optimizations implemented: Identity optimization, constant folding, and global value numbering. The program dependence graph data structure allows some of the optimizations run in parallel benefiting from each other [6][7]. The following three subsections give small examples for each of them. The fourth subsection presents an example of an optimization applied globally after parsing.

3.4.1 Identity Optimization

The identity optimization searches for nodes that compute that same result. In contrast to global value numbering, it searches also for nodes that are different, but produce the same output. Figure 3.8 shows how the expression $x+0$ is processed by the server compiler. First, the full expression including the `AddI` and the `ConI` node are generated (left side). Now the identity optimization finds out that the `Parm` node produces always the same result as the newly created `AddI` node and uses only the `Parm` node further on. After parsing all bytecodes, the compiler performs a dead code elimination: It deletes the `AddI` node and the `ConI` node. The resulting graph is shown on the right side.

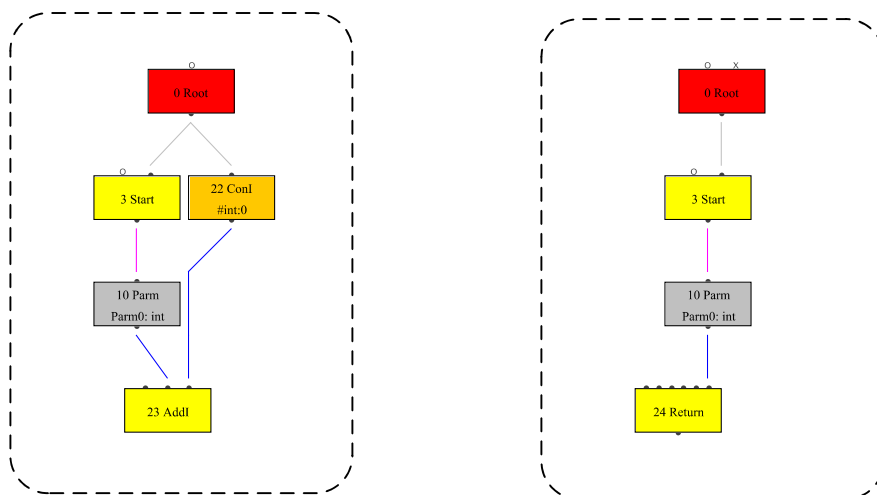


Figure 3.8: Identity optimization: $(x+0)$ is transformed to x .

3.4.2 Constant Folding

Arithmetic operations on constants are performed at compile time and the result is represented by a constant node. In Figure 3.9 the graph for the expression $5+p+7$ is shown. The first add operation $5+p$ is modeled as an `AddI` node, but it is immediately transformed to the expression $p+5$, as the convention that the constant part is always the last input simplifies constant folding. After the compiler has generated the nodes for the second add operation, the constant folding algorithm identifies a simplification possibility and the whole expression is replaced by $p+12$. The algorithm looks for the pattern that the second input of the add operation is a constant and the first input is an `AddI` node, which has a constant input too. Dead code elimination removes the unnecessary two constant nodes and the `AddI` node.

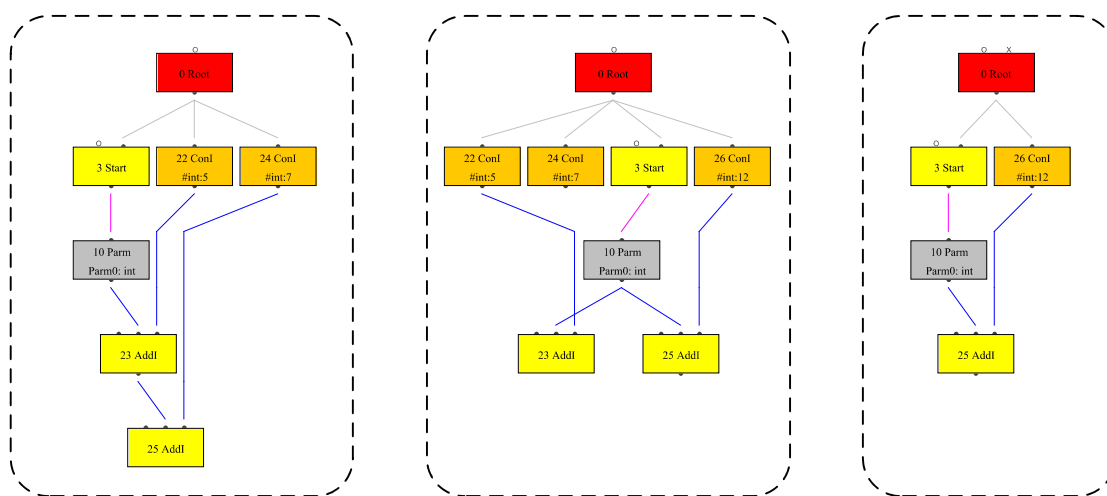


Figure 3.9: Constant folding: $(5+p+7)$ is transformed to $(p+12)$.

3.4.3 Global Value Numbering

Global value numbering is an optimization similar to the identity optimization. It searches for nodes that are equal to the currently inserted nodes. Equality means that the nodes themselves and also all of their inputs are equal. In this case, only one of them is needed and the other one gets deleted by dead code elimination. A node has a hash value for fast equality testing. It is based on its properties and the C++ memory addresses of its inputs.

Figure 3.10 shows the graph produced when compiling the statement $(x+1) * (x+1)$. The left graph is a snapshot taken after the processing of $(x+1)$. In the middle the second $(x+1)$ is represented by the `AddI` and the `ConI` node. As the hashcode of the two `AddI` nodes is the same, the old `AddI` node is connected a second time to the safepoint node instead of the newly created `AddI` node. So the following `MultI` node gets a connection to the first `AddI` node in both slots. After dead code elimination the compiler deletes the second `AddI` node. The resulting graph is shown on the right side.

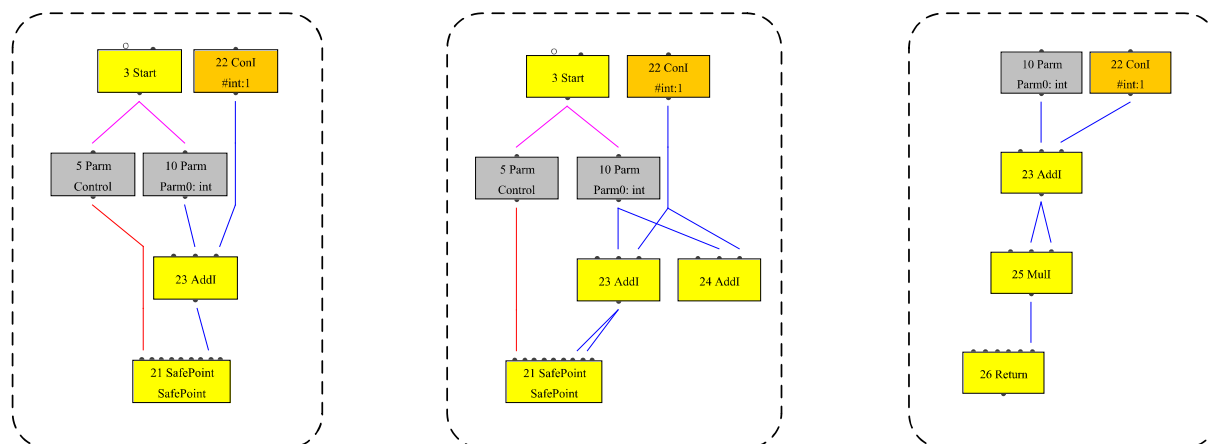


Figure 3.10: Global value numbering: $(x+1) * (x+1)$ is transformed to $(x+1)^2$.

3.4.4 Loop Transformations

The server compiler performs a large number of global optimizations after parsing. They can be divided into three main categories: Iterative global value numbering, conditional constant propagation, and loop transformations. As presenting all of them would go beyond the scope of this thesis, only the step to identify counted loops is described in this section.

The identification of loops brings advantages for array bounds check elimination and is necessary for loop unrolling and loop peeling. After parsing the bytecodes, a loop is represented by a control flow cycle involving `Region` nodes. The first task is to find regular loops within the graph and identify `Region` nodes that represent *loop headers*, i.e. nodes that the control token must always pass when entering the loop. A loop header is represented by a `Loop` node, which replaces the `Region` node. When the input bytecodes were created by compiling Java code, then there exist only loops with one entry. There are however no such restrictions on the bytecodes. As loops with more than one entry are a rare case and handling them would be complicated, the server compiler does not optimize such loops.

A common loop pattern is represented by a loop variable starting at a specific value and going constant steps up to an upper bound. After reaching the bound, the loop is exited. Some languages like FORTRAN have language constructs for this kind of loops, but in Java this must be coded using a local variable and manually inserted increments and conditions. There are special optimizations for such loops, so the server compiler identifies the loop pattern and converts it to a construct with a `CountedLoop` node and a `CountedLoopEnd (CLE)` node. Figure 3.11 shows the nodes that define a counted loop. They specify the beginning and end of the loop, as well as the loop variable and the increment per loop iteration.

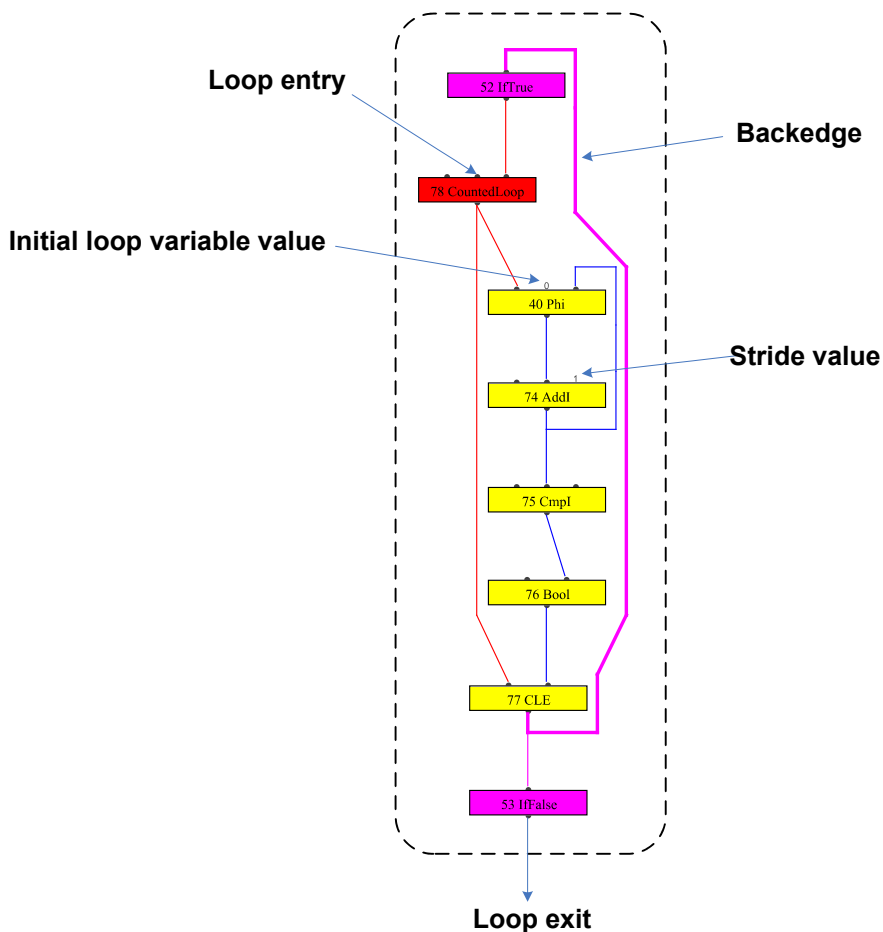


Figure 3.11: Nodes that define a counted loop.

3.5 MachNode Graph

After all global optimizations are applied, the ideal graph is still in a machine-independent form. The next step is converting the graph to a more machine-near form. The resulting nodes are later scheduled and directly converted to machine code. Bottom-up rewrite systems [24] can be used for the optimal selection of machine instructions when producing machine code from expression trees. The server compiler selects subtrees out of the ideal graph and converts them one by one. It selects specific nodes as root nodes and transforms their related tree using tree selection rules. Some nodes like Phi nodes are marked as `dontcare` and have no corresponding nodes in the new graph. Other instructions are marked as `shared`, which means that they must not be shared among subtrees. Code for instructions that are part of more than one subtree is duplicated. The result of the root node of a tree is always placed in a register so it can be reused without recomputation.

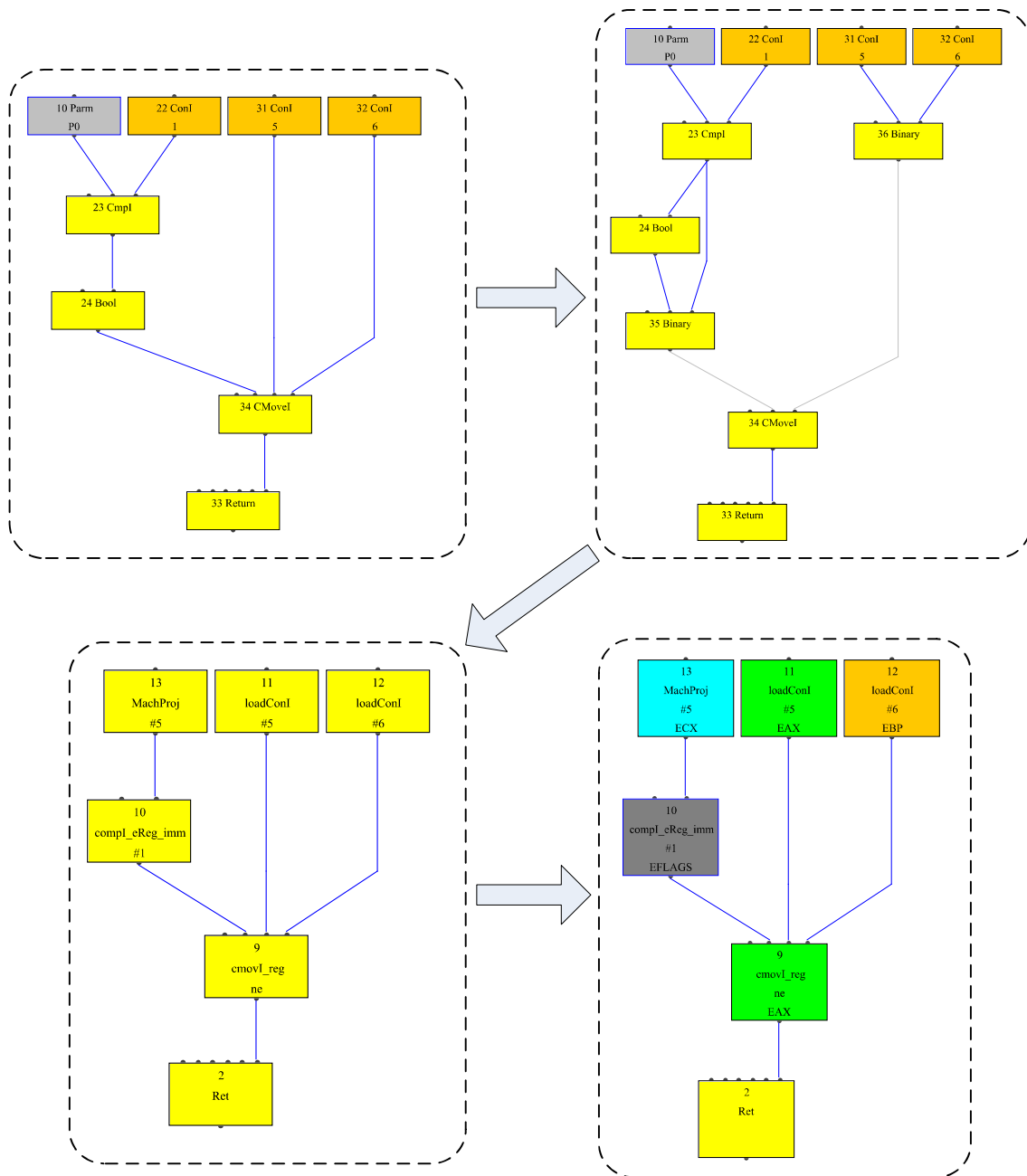


Figure 3.12: Matching and register allocation example.

Each tree is converted using a deterministic finite automata. There exist architecture description files for i486, AMD64, and Sparc that describe the available instructions and their costs. Listing 3.1 shows schematically an extract of the i486 architecture description file. The line starting with `match` specifies the tree pattern that can be converted by this rule. A rule has an associated estimated cost. The compiler matches a tree such that the total cost of applied rules

is minimal. The file also contains additional properties of the rules including for example the resulting machine code. The first rule converts a `CmpI` node with a register and an immediate operand to a `compI_eReg_imm` node. The second rule can convert a `CMoveI` node with two `Binary` nodes as predecessors to a single `cmovI_reg` node.

Listing 3.1 Architecture description file extract.

```
// Signed compare instruction
instruct compI_eReg_imm: flags cr, register op1, immediate op2
  match:    Set cr (CmpI op1 op2)
  opcode:   0x81,0x07

// Conditional move
instruct cmovI_reg: register dst, register src, flags cr, operator cop
  match:    Set dst (CMoveI (Binary cop cr) (Binary dst src))
  ins_cost: 200
  opcode:   0x0F,0x40
```

Figure 3.12 shows how the graph for the example method presented in Figure 3.6 is converted to a machine-specific form. During global optimizations, the compiler replaces the `If` and the `Phi` node with a `CMoveI` node (see top-left). For the rules to match correctly some constructs must be changed. In this case, the two `Binary` nodes are inserted and form new inputs of the `CMoveI` node (see top-right). The matcher identifies that the two rules defined in

Listing 3.1 can be applied. The two `ConI` nodes representing the values 5 and 6 are converted to `loadConI` nodes. The node for constant 1 is no longer necessary, because the information that `compI_eReg_imm` should compare the input with 1 is modeled as a parameter. The node `cmovI_reg` is created according to the second rule. The bottom-left graph shows the result of the matching process.

After the construction of the `MachNode` graph, the compiler builds the control flow graph consisting of basic blocks and schedules the nodes. Then the register allocator selects machine registers for the nodes (see bottom-right graph).

3.6 Register Allocation

Register allocation selects machine registers to hold values that must be stored between calculations. If there are not enough registers available to hold all values, they must be temporarily stored in the main memory, which is an expensive operation called *spilling*. The goal is to have as less spillings as possible when executing a method. The *life range* of a value is the range between its production and its last usage. Two values can be stored in the same register if their life ranges do not intersect. If the life ranges intersect, then at some time both of them must be stored, so it is impossible to store both values in the same register.

The server compiler uses a graph coloring register allocator [5][3]. First it builds an *interference graph*, which is a graph with the values as nodes and an undirected connection between two nodes if their life ranges intersect. A coloring of a graph is an assignment of a color to each node of the graph with the restriction that two directly connected nodes must not have the same color. When the available registers are viewed as the colors, then a valid coloring the graph is a valid register allocation. Two values that have intersecting life ranges are directly connected in the interference graph and get therefore different registers assigned. When it is not possible to color the graph, then spilling a value is unavoidable. The life range of the value is splitted: one life range between its production and the storage to memory, another life range between its loading from memory and its last usage. Now the interference graph has a better chance of being successfully colored as most likely some of the connections of the original life range do not exist in one of the two shorter new life range intervals..

First, the algorithm to color a graph with n colors selects nodes with less than n neighbors. Obtaining a correct color for such a node is trivial when the rest of the graph is successfully colored. The node gets the color that is not used by any of its neighbors and as there are maximal $n-1$ neighbors, such a color always exists. Such easily colorable nodes are consecutively removed from the graph. If at some point there is no such node, then the graph is not colorable. The server compiler iteratively inserts spilling code until a valid coloring is found.

The register allocation by graph coloring is expensive for large methods. The client compiler uses a linear scan register allocator [31] instead of a graph coloring algorithm.

Chapter 4

User Guide

This user guide introduces the most important functionalities of the Java HotSpot™ server compiler visualization tool. The tool consists of a Java application, which is used to display and analyze the graphs, and an instrumentation of the Java HotSpot™ server compiler that generates the data. Figure 4.1 shows the global architecture. There are two ways to transfer the data from the server compiler to the Java application, either via intermediate XML files or directly via a network stream. The Java application consists of several window components that are explained in this user guide.

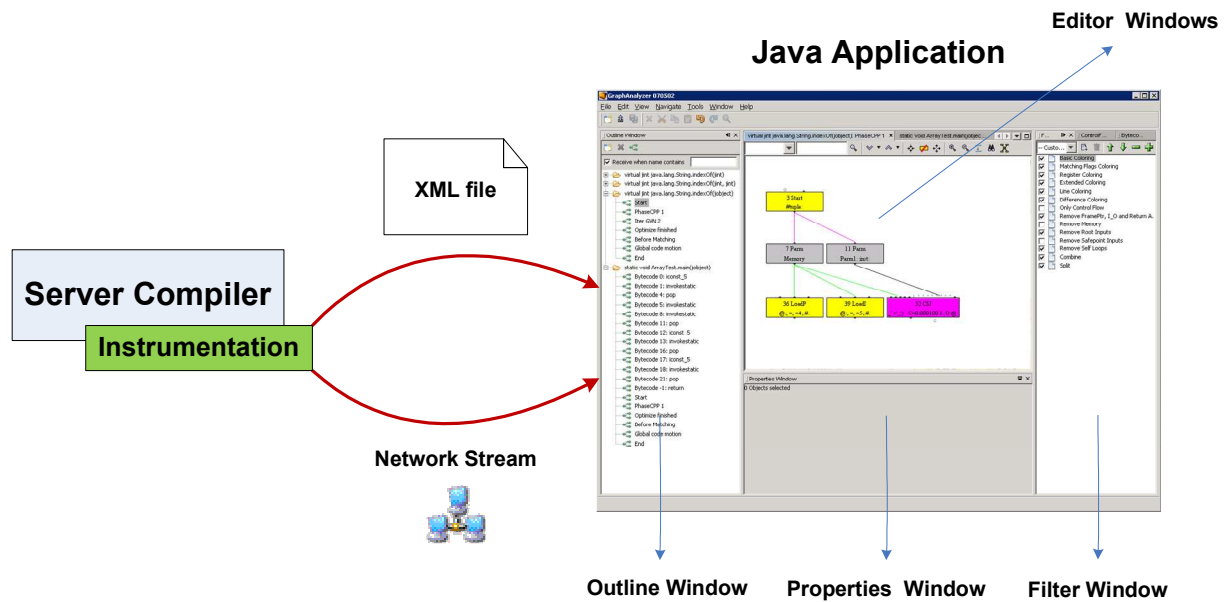


Figure 4.1: Architectural overview.

4.1 Generating Data

A special debug version of the Java HotSpot™ server compiler is needed for generating data. It has an additional command line option `-XX:PrintIdealGraphLevel=1` that specifies how detailed the compiled methods should be recorded, i.e. how many snapshots of the graph should be taken during compilation. There are four different levels:

Level 0: This is the default value and stands for disabling tracing at all.

Level 1: At this level only three states per method of the graph are traced: one state immediately after parsing, one state after the global optimizations have been applied, and one state at the end of compilation before machine code is generated.

Level 2: This level includes intermediate steps for the global optimizations: iterative global value numbering, loop transformations, and conditional constant propagation. The number of graphs depends on the number of applied optimization cycles. Additionally, the state of the graph is traced before it is converted to a `MachNode` graph and once before register allocation.



Level 3: The third level is detailed: After each parsed bytecode, the compiler traces a graph state, and the loop transformations are dumped with more intermediate steps.

With increasing level, the necessary storage space and compile time overhead increases too, so the lowest needed level should be used.

At startup, the server compiler tries to open a network connection to the Java visualization application. The two options `-XX:PrintIdealGraphAddress=ip` specifies the network address and `-XX:PrintIdealGraphPort=p` the port. The default values are "127.0.0.1", i.e. the local computer, and port 4444. If opening of the connection succeeds, the data is immediately sent to the tool. Otherwise, the data is saved to a file called `output_1.xml`. With multiple compiler threads the second thread saves its data to `output_2.xml` and so on.

All compiled methods are recorded. By default the Java HotSpot™ VM decides based on the invocation count and the number of loop iterations when it schedules a method for compilation. The flag `-Xcomp` completely disables the interpreter, so all methods get compiled before their first invocation. Using this flag however means that a large number of methods get compiled. The option `-XX:CompileOnly=name` can be used to restrict compilation to a certain class or method.

The currently loaded methods of the application are available in the Outline Window. XML data files can be loaded using the `File->Open` menu item. When the network communication is in use, the transferred methods appear automatically. In the top section of the Outline Window, listening on a port for data can be enabled and disabled with a checkbox. Additionally, a filter can be specified to reduce the number of methods that should be traced. The server compiler sends only methods to the Java application if their name contains the string specified in the textbox next to the checkbox.

The methods appear with a folder icon  and the available snapshots for a method are child elements . Double clicking on a snapshot opens a new editor window in the center and displays the graph.

4.2 Viewing the Graph

The viewed graph consists of nodes with input and output slots and edges that connect two slots. The input slots are always drawn at the top of a node, the output slots at the bottom. When a node is selected using the left mouse button, its key-value pairs are shown in the Properties Window. This functionality is also available for all items in the Outline Window. The text that appears inside the nodes is an extract of their property values and can be customized in the preferences dialog. Figure 4.2 shows the editor window of an example graph and the corresponding Properties and ControlFlow Window.

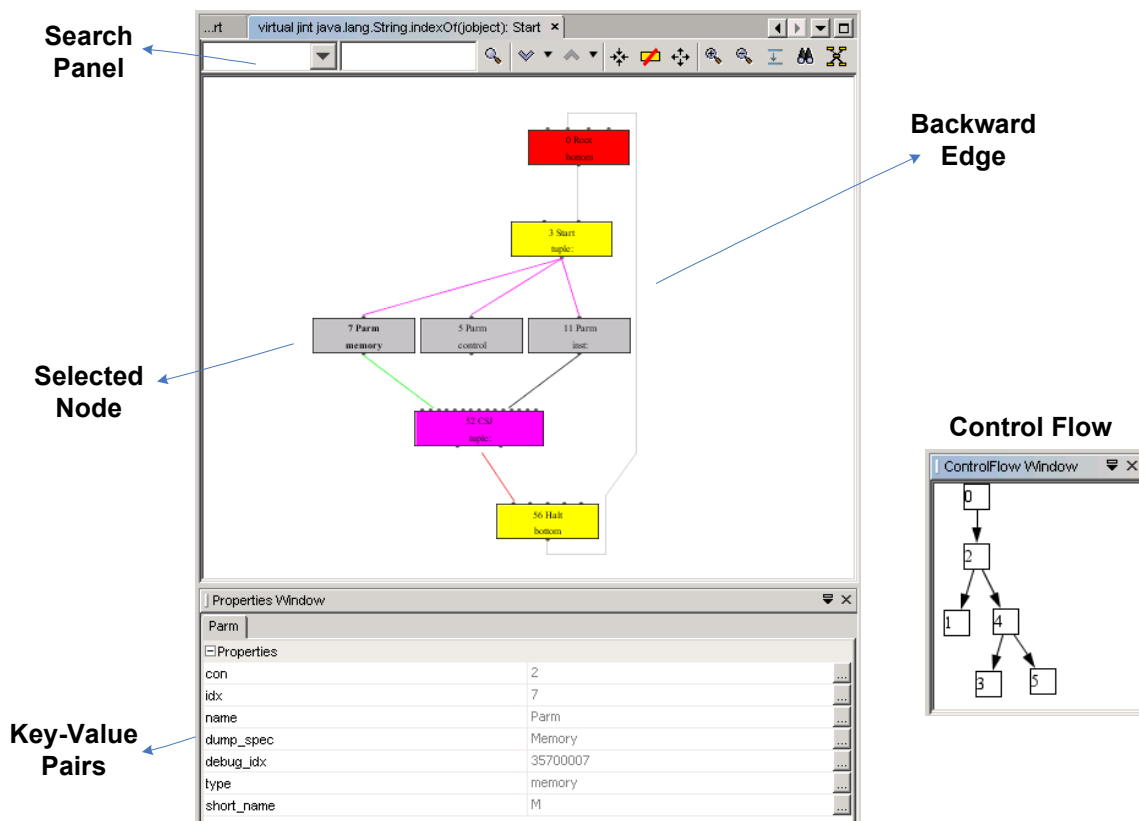





Figure 4.2: Viewing a graph using the Java application.



Right-clicking on an edge shows a context menu with its source and destination nodes. This is especially useful for edges that are only partially visible. When an edge would be so long that it

disturbs the drawing, it is cut and only its beginning and ending is drawn. By default, the nodes are drawn grouped into clusters. This can be turned off and on using a toolbar button .


Rolling the mouse wheel zooms in and out, there are also toolbar buttons available for this purpose  . The currently shown extract of the graph is changed by holding the middle mouse button pressed and dragging around. A detailed description of how to navigate through the graph is given in the next section. The current graph can be exported to an SVG file using the `File->Export` menu item.


When a graph is currently opened, the difference to a second graph can be calculated: Right-clicking on another graph and selecting the option `Difference to current graph` opens a new window with an approximation of a difference between the two graphs.

4.3 Navigating within the Graph

As in most cases only a particular part of a graph is of interest, navigation possibilities are mandatory. When a graph is opened for the first time, all nodes are visible and the root node is shown horizontally centered on the screen. Selected nodes can be hidden from the view using the context menu or a toolbar button . There is a button to show again all nodes . In the context menu of a node, two submenus allow to navigate to one of its immediate predecessors or successors.

Nodes that are not marked as fully visible can be either semi-transparent or invisible: Not fully visible nodes are semi-transparent when they are connected to a node that is fully visible, otherwise they are invisible. When such semi-transparent nodes are double clicked, they become fully visible. On the other hand, fully visible nodes become semi-transparent or invisible when they are double clicked. This allows fast expanding and shrinking of the current set of visible nodes without using the context menu. There is one exception of the double click semantics: When all nodes are fully visible, then double clicking on a node does not hide this node, but hides all other nodes of the graph.

So the standard use when analyzing a specific node is to first search for the node in the full graph. Then show only this node by double clicking it and afterwards expand the predecessors and successors of the node as needed. When the selection of the target node set is done, the semi-transparent nodes are more disturbing than helpful in most cases. Therefore they can be completely hidden using the toolbar button .

Another way of navigating through a large graph and nevertheless keeping overview is to use the satellite view. It can be enabled by holding the key 's' pressed or using the toolbar button . It displays a zoomed out view of the whole graph that exactly fits into the editor window and draws a rectangle that indicates the part that was currently shown. This viewport can be moved around using the mouse. When the satellite view is exited either by releasing the key or deselecting the toolbar button, the new extract of the graph is shown. Figure 4.3 shows the satellite window of an example graph and the corresponding extract of the graph that is shown in the editor window.

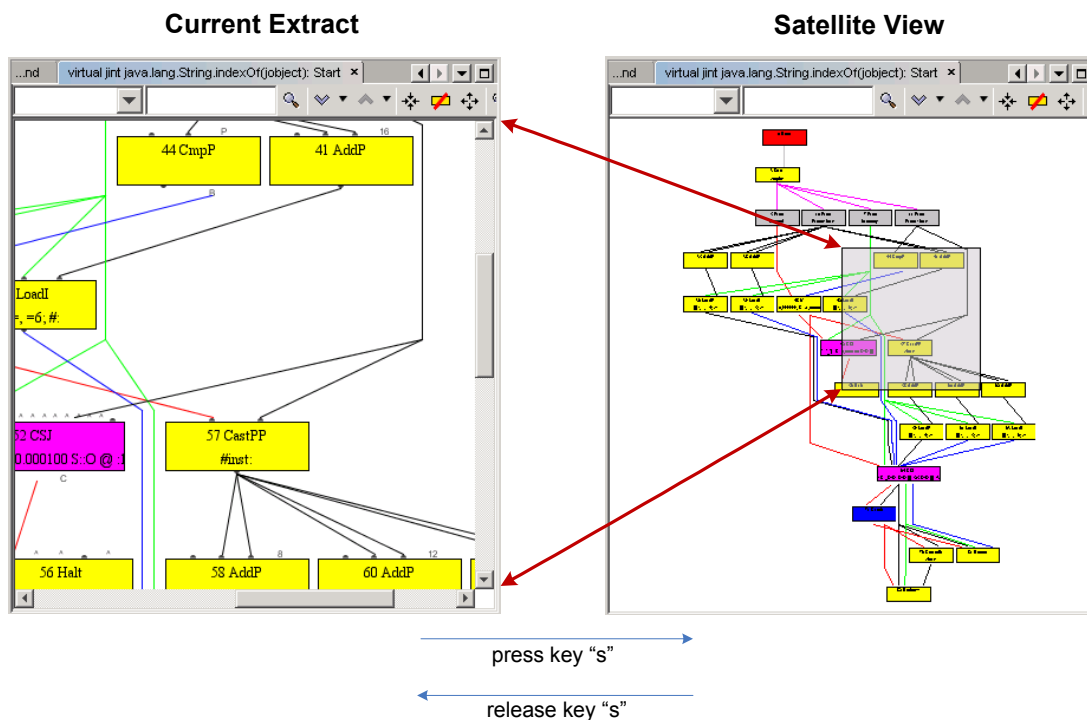


Figure 4.3: The satellite view gives an overview of a graph.






4.4 Control Flow Window

To increase the overview in large methods, an approximation of the control flow graph is available. Every node is assigned a *basic block*, i.e. a set of instructions that are executed consecutively without any branches. The Control Flow Window shows a graph with a node for every block that is connected to the block’s predecessors and successors. Note that this is only an approximation as the real control flow information is only available in a late stage during compilation. Every node is put in the latest possible block fulfilling the condition that it must be evaluated before all its successors. Selecting a block selects all nodes that are assigned to this block in the full graph and centers them in the view. Invisible nodes of the block get automatically visible.

4.5 Filters

The graph coming from the server compiler does not contain any display information. The only additional information available beside the graph description are key-value pairs for the nodes of the graph. Filters change the representation of the graph based on node properties. There

are filters for changing the color of nodes and edges, for removing nodes and also two special filters for combining and splitting nodes.

In the Filter Window, all currently available filters are listed in their processing order. Filters can be activated and deactivated using the checkbox left to their name. The toolbar buttons on the right allow adding , removing , and moving filters  . The current set of selected filters can be saved as a profile  and is then recallable using the combobox on the left.

The following list explains the standard filters that are available when first installing the tool. They all use selection rules based on key-value pairs and can be customized.

Basic Coloring: Color filter that should be enabled by default. It sets a standard color and special colors for control flow specific nodes.

Matcher Flags Coloring: Before converting the ideal graph to a MachNode graph, the two flags `is_shared` and `is_dontcare` are calculated. This filter visualizes the flags when their value is available.

Register Coloring: Colors the nodes according to the selected register. The register allocator information is available at a late stage during compilation.

Extended Coloring: Gives a color to constant nodes, projection nodes, and nodes that have a `bci` property, i.e. nodes that are safepoints.

Line Coloring: Colors the connections according to the type of the source node. Differentiates between integer values, control flow, memory dependencies, tuple values, and the special type `bottom`.

Difference Coloring: In a difference graph all nodes have a state expressed as a property that can be either same, changed, new or deleted. According to this state, the filter sets a node color. When the nodes do not have a state property, the graph remains unchanged.

Only Control Flow: This filter is useful when the full graph is too complex and one only wants to focus on the control flow. It removes all nodes that do not produce a control flow value and are not immediate successors of a node that produces a control flow value.

Remove FramePtr, I_O, and ReturnAddress: Removes the three nodes `FramePtr`, `I_O`, and `ReturnAddress` from the graph as they normally are not of interest and disturb the view of the graph. They are connected to all safepoint nodes.

Remove Memory: Removes any node that produces a memory dependence as its value.

Remove Root Inputs: Every possible end of a method has a backward edge going to the root node. So for large graphs the number of inputs into the root node can be high, which disturbs the drawing. Therefore this option should be enabled as the root inputs are not of interest in most cases.

Remove Safepoint Inputs: The inputs of a safepoint specify the values of the expression stack and local variables at the safepoint's bci. This is interesting when the graph is built from the bytecodes, but not very important afterwards. Removing the inputs improves the overview and the drawing performance.

Combine: When a node produces more than one output value, it produces a tuple and the specific values must be selected using projection nodes. This filter combines such a node with all its projection nodes and creates a single node with multiple output slots.


Split: Constants are shared among all nodes. So when a constant is used multiple times, all usages refer to a single node representing the constant. This is reasonable to save memory capacity but is impractical for displaying the graph. This filter removes constant nodes and writes their value directly to all slots where they were used.

Filters are written in JavaScript using Java objects and shortcut functions. Double clicking on a filter opens a dialog that allows editing its name and its code. Filters are programmed based on selection rules applied to the graph. There exist some predefined functions that cover most filtering tasks. They are listed in the following table:

<code>colorize(name, regexp, color)</code>	Colors all nodes whose property name matches the regular expression <code>regexp</code> . Predefined color variables are: <code>black</code> , <code>blue</code> , <code>cyan</code> , <code>darkGray</code> , <code>gray</code> , <code>green</code> , <code>lightGray</code> , <code>magenta</code> , <code>orange</code> , <code>pink</code> , <code>red</code> , <code>white</code> , and <code>yellow</code> .
<code>remove(name, regexp)</code>	Removes the matching nodes from the graph.
<code>removeInputs(name, regexp, start, end)</code>	Removes all inputs from the matching nodes from the index <code>start</code> to the index <code>end</code> .
<code>split(name, regexp)</code>	Splits the matching nodes.

`regexp` stands for a string representing the regular expression that the value of the property with the specified name must fulfill. The syntax corresponds to the standard Java regular expression syntax used by the classes in the `java.util.regex` package. Amongst others, the following rules are defined: "." stands for any character, "*" means that the preceding element is repeated zero or more times, and "|" expresses alternatives.

Figure 4.4 shows each of the predefined functions applied to an example graph. The plain graph without any filters applied is displayed top left. Then the graph is colored using the `colorize` function and regular expressions. Afterwards the `Phi` node is removed. The function `split` removes the `ConI` node from the graph and displays the short name for the node at every use. So "0" is drawn at the third input of the `CmpI` node. The last applied step removes the second input of the `Start` node.

The search panel in the toolbar of the center window works in a similar way to a filter. A property name can be selected in the combobox, and a regular expression that this property of the target nodes must fulfill can be entered in the textfield. After pressing enter in the textfield or the search toolbar button , all matching nodes are selected.

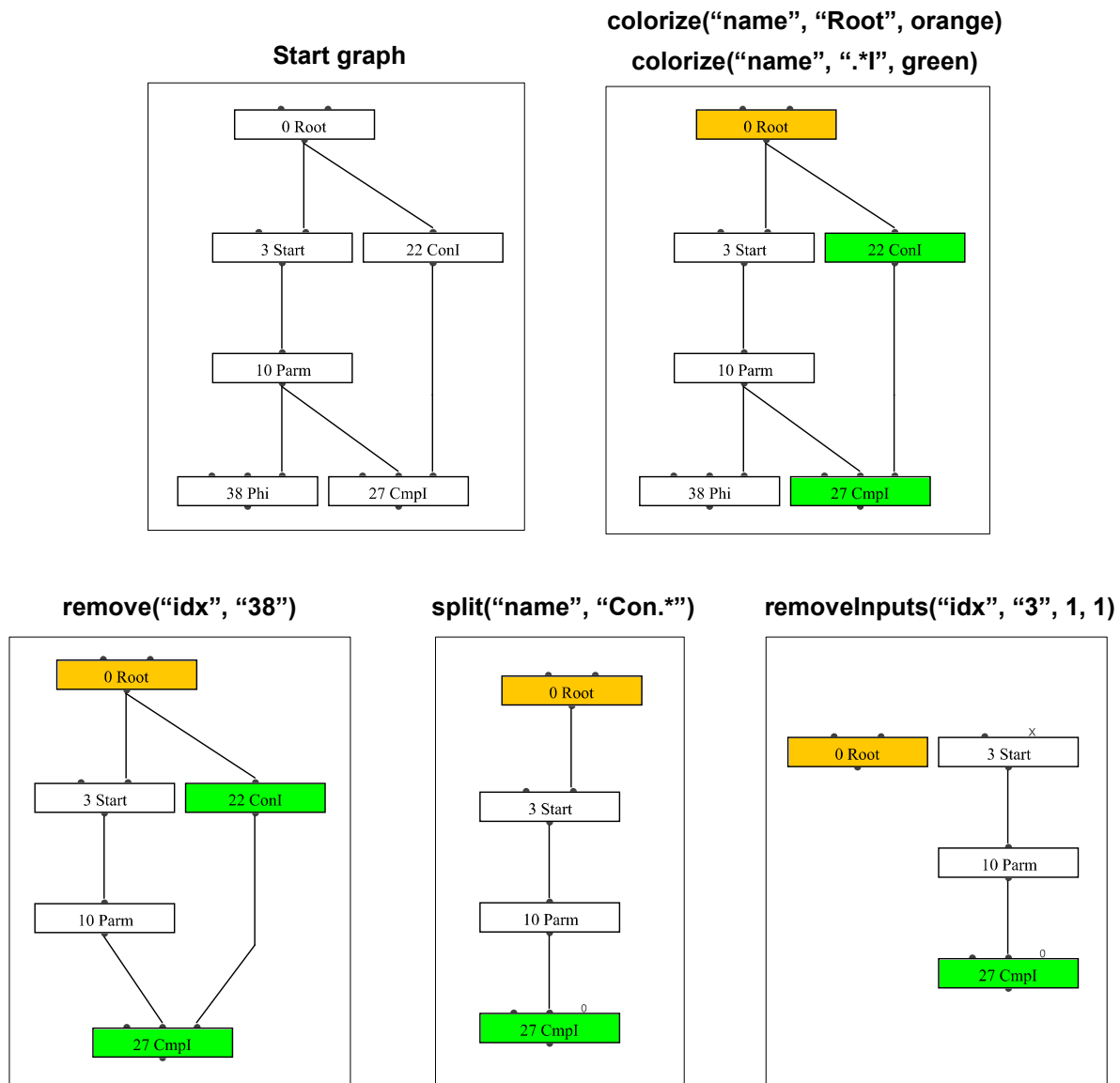



Figure 4.4: Four functions applied to an example graph.

4.6 Bytecode Window

The Bytecode Window shows the Java bytecodes of the method from which the currently opened graph was generated. Bytecodes that are referenced from nodes through the bci property are shown with a special icon. Double clicking on such a bytecode will select all nodes that have a reference to it. These are mainly safepoint nodes.

The server compiler inlines small methods to avoid the overhead of calling them. The bytecodes of inlined methods are shown as child elements of a node beneath the bytecode that would invoke the method. Figure 4.5 shows the bytecode view for an example method. The bytecode with the index 3 is a call to another method, which the server compiler decided to inline. The bytecodes of the inlined method are all shown in the sublist. There are two safepoint nodes in the graph that reference the bytecode indices 13 and 25 of the inlined method. Therefore, those bytecodes have a special icon  and by double clicking on them, the corresponding nodes in the graph are selected.

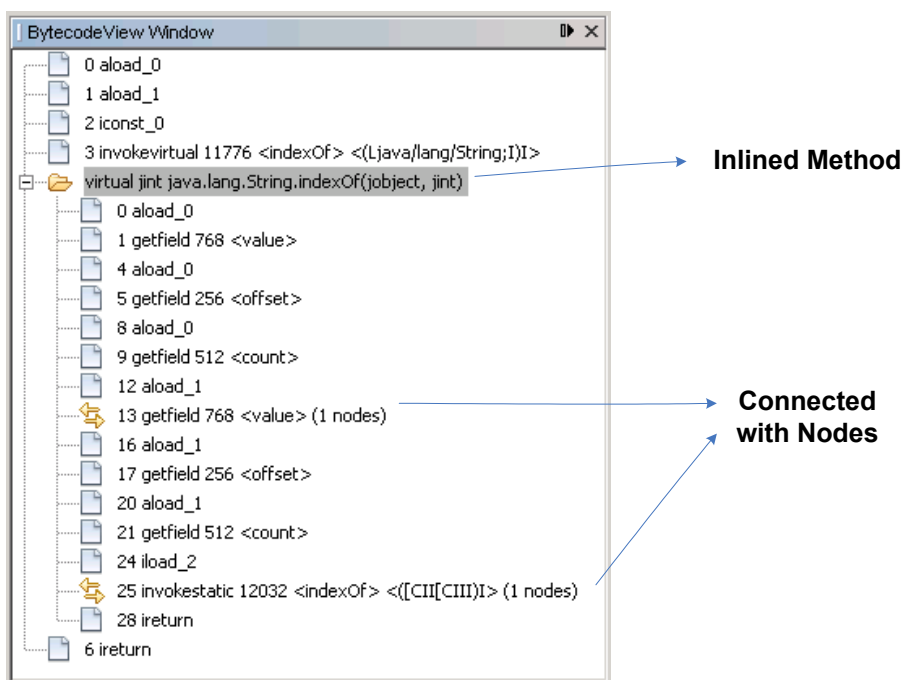


Figure 4.5: Bytecode Window showing the Java bytecodes in tree-form.

Chapter 5

Visualizer Architecture

In this chapter, the architecture of the visualization application that is based on the NetBeans platform is explained. Additionally, it contains a description of the algorithm for finding the differences between two graphs. The layout algorithms and the code added to the server compiler are presented in two subsequent chapters.

5.1 Module Structure

The NetBeans application is split into several modules represented by NetBeans projects. For a discussion of modular programming see Section 2.3. Figure 5.1 shows the modules and their dependencies. Transitive dependencies are omitted for simplicity. Two modules represent third party libraries: RhinoScripting for the execution of JavaScript code and BatikSVG for the export of SVG files. There are three top-level modules: Bytecodes, ControlFlow, and Coordinator. Here is a list of all modules in alphabetical order:

BatikSVG: The Batik SVG Toolkit is part of the Apache XML Graphics Project. SVG stands for Scalable Vector Graphics and is a vector-based standardized graphics format. The Batik SVG library allows to create a Java `Graphics` object that writes into an SVG file instead of painting on the screen. The visualization tool uses this functionality to export the currently selected graph into an SVG file.

Bytecodes: A top-level module that is responsible for the bytecode view of the method of the current active graph. It listens to the lookup of the window with focus and displays the tree of bytecodes of the current graph's method.

ControlFlow: Contributes the control flow view of the currently active graph and listens to the lookup of the current active window similar to the Bytecodes module. It is also a top-level module and uses the NetBeans visual library for displaying the control flow graph. It forwards the changes of the block selection to the active graph editor window.

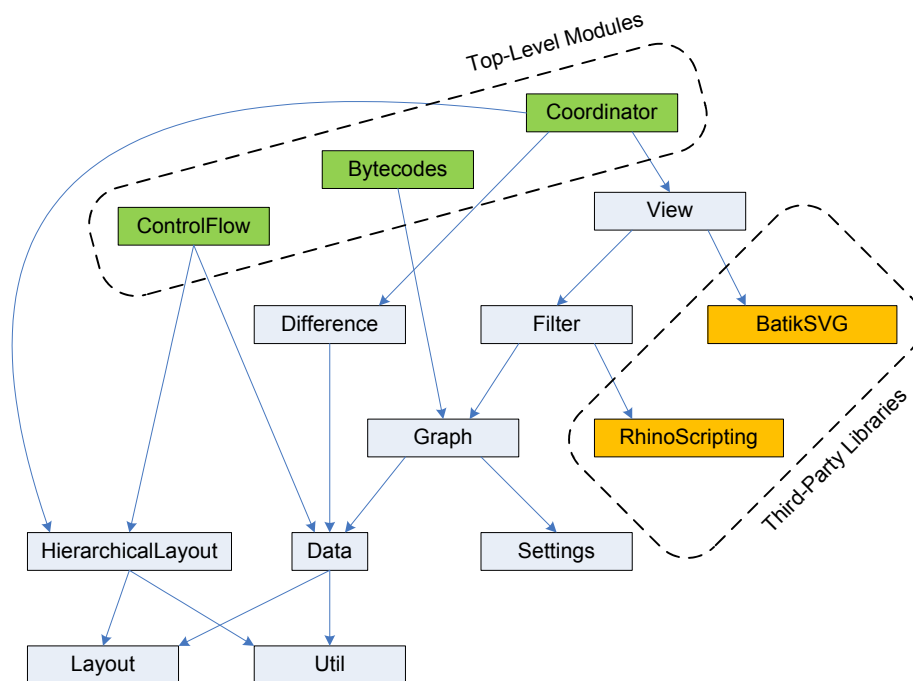


Figure 5.1: Dependencies of the NetBeans modules.

Coordinator: Top-level module that contains the code for the outline and the filter window. All predefined filters are defined in this module. It is responsible for opening new graph editors in the center area. The task of actually creating the window and displaying the graph is forwarded to the View module.

Data: Contains the data model used to transfer data between the server compiler and the tool. It parses the XML data coming from the server compiler and converts it to an internal data structure. It uses a SAX XML parser. For a detailed description of the data model see Section 5.2.1.

Difference: Used to create a difference graph of two input graphs. It depends only on the Data module and the Util module. The difference algorithm is explained in Section 5.5.

Filter: Contains the filters that can be applied to the graph model and the dialog to edit a filter. It depends on the Graph module and the RhinoScripting module for executing JavaScript code. Section 5.4 contains a description of the filter architecture.

Graph: The data model used internally for the graphically enriched graph. In comparison to the data model defined in the Data module, it adds display information to the nodes and edges of the graph. For a detailed description see Section 5.2.2. Additionally, it contains the graph selectors that are used by the filters and are explained in Section 5.4.

Layout: Defines an API for layouting a graph which is explained in Section 5.2.3. It contains the interface definitions for the nodes and edges of the graph and also an interface for clusters of nodes.

HierarchicalLayout: Actual implementation of a graph layouter as defined in the Layout module. It also contains a hierarchical layouter that can handle clustering. The layout algorithms are explained in Chapter 6.

RhinoScripting: Rhino is an open source implementation of a JavaScript engine and is part of the Mozilla project. It contains all features of JavaScript 1.5. The visualization tool uses the library to allow the user to write filters in JavaScript code.

Settings: Module that is responsible for loading and storing settings that should be persistent. It also contains the code for displaying the settings dialog.

Util: Contains some utility classes that are used by other modules. It contains the definitions for the properties mechanism described in Section 5.3.

View: Responsible for displaying the graph in a new editor window. It uses the NetBeans visual library to render the graph. For a description of the visual library see Section 2.6.

5.2 Graph Models

The application uses three different models for the graph: The *data model* for the transfer of the data from the server compiler to the Java application, the *display model* for displaying and filtering the graph, and the *layout model* for layouting the graph. Working with different models comes with the cost of converting between them. The data model is converted to the display model whenever a graph is opened in a new editor window. The layout model is a submodel of the display model, so no conversion is needed there. An advantage of using different models is the avoidance of unused fields. For example, all fields containing display information such as the color of the nodes would be undefined for all loaded graphs and would only get a meaning after a graph is opened.

The application saves for every node of the display graph model the corresponding nodes in the data model. When the graph of the display graph model is created for the first time, every new node represents exactly one node of the data model. Through filters, however, a node of the display model may represent several nodes of the data model.

Figure 5.2 shows the lifecycle of a graph and how it is converted between the models. For transmitting the data from the server compiler to the Java application and for storing it on the hard disk, an XML structure based on the data model is used. The Java application reads the data and builds a memory representation of it. When a graph is opened in a new editor window, the data model representation of the graph is converted to a display model representation. The application applies all activated filters on the graph and invokes the layout manager. The layout

manager works on the layout model, which is a submodel of the display model. The editor windows use the NetBeans visual library to show the display model representation of the graph on the screen. The following three sections describe the three models in detail.

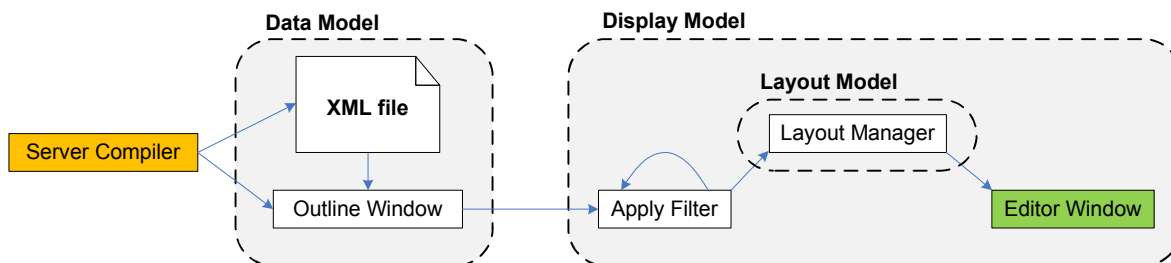


Figure 5.2: Lifecycle of the graph data.

5.2.1 XML File Structure

The data transferred from the server compiler to the visualization tool is represented in XML. The main advantage of using XML instead of a custom binary format is better changeability. When new XML elements are introduced, it is simple to maintain backward and forward compatibility: On the one hand, the old application can read the new XML data because it ignores the new elements. On the other hand, the new application can read the old XML data by ignoring the elements that are no longer valid. The main disadvantage of using XML is the higher storage requirement compared to a binary format.

One of the goals when designing the data model was that it should be generally usable for describing directed graphs. This is an important design decision that opens the possibility that the visualization tool is used to analyze graph-like data structures of a completely different application.

Another concept of the data model are properties based on key-value pairs. All descriptive information about an entity is stored uniformly as a list of key-value pairs, instead of introducing XML attributes for the properties of Java methods, graphs, and nodes. This concept is used throughout the application and is explained more detailed in Section 5.3.

Figure 5.3 shows the XML elements and their relations. A `graphDocument` is the top-level element and can contain `group` child elements. The server compiler creates a `group` element for every traced method. A `group` element has exactly one `method` child element that describes the bytecodes and inlining of the method. A `group` element can have an arbitrary number of `graph` child elements that describe the traced states of the graph during compilation of the method. A `graph` element has one `nodes`, one `edges`, and one `controlFlow` child element.

Concerning the nodes and edges, only the difference to the previous graph of the method is saved. Therefore, the `nodes` element can contain definitions of nodes as `node` elements or

removeNode elements, which state that a certain node of the previous graph is no longer present. A similar mechanism is used for the edges. Succeeding graphs of a method often have similar nodes and edges. The number of equal nodes and edges depends on the number of traced intermediate steps. Using a difference-based storage format highly reduces the necessary storage space.

Every node has a unique identifier that is referenced from the edges and the control flow blocks. An edge is defined by the identifiers of its source and destination node and the index at which the edge ends at the destination node. The controlFlow element contains the information necessary to cluster the nodes into blocks. For each block, its successor blocks and the nodes that are related to that block are specified. The nodes are referenced via their unique id.

A method element contains two child elements: The inlined element can have method child elements expressing inlining. The bytecodes of a method are stored in the textual format that is used to trace the bytecodes in the server compiler.

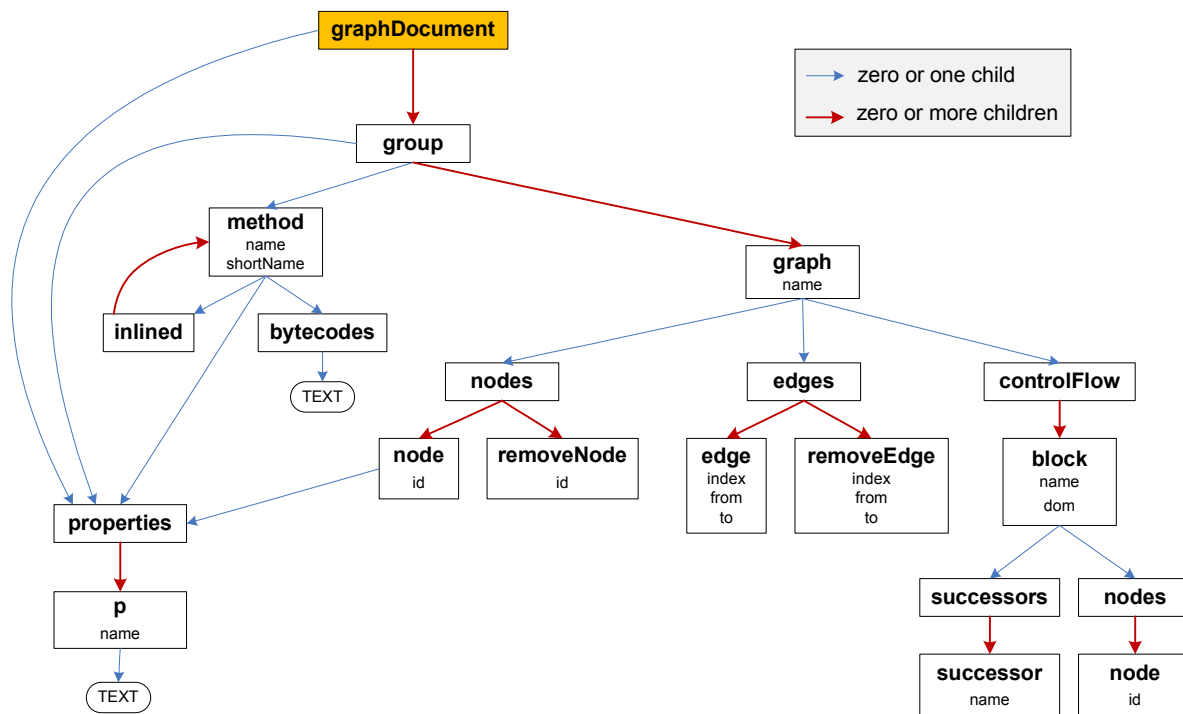


Figure 5.3: XML file structure.

The elements graphDocument, group, method, and node can have a properties subelement that specifies key-value pairs via its child elements. The concept of properties is also present in the data and the display model and is described in detail in Section 5.3.

The XML data is read using a parser that processes the elements while reading it. Reading from the network stream and reading from a file is treated in a uniform manner. The network

communication between the server compiler and the visualization tool is interactive: After the server compiler has sent a `group` element and its properties, the client compiler writes the character 'y' if it wants to receive that group of graphs or 'n' otherwise.

From the XML data, the data model memory representation is built. Figure 5.4 shows the classes that represent the data model. The structure is similar to the XML structure. The nodes and edges are however no longer represented as the difference to the previous graph. The parser resolves this differences without any additional memory requirements by sharing the `InputNode` objects. If a node does not change between two graphs of a method, then both graphs contain a memory pointer to that node.

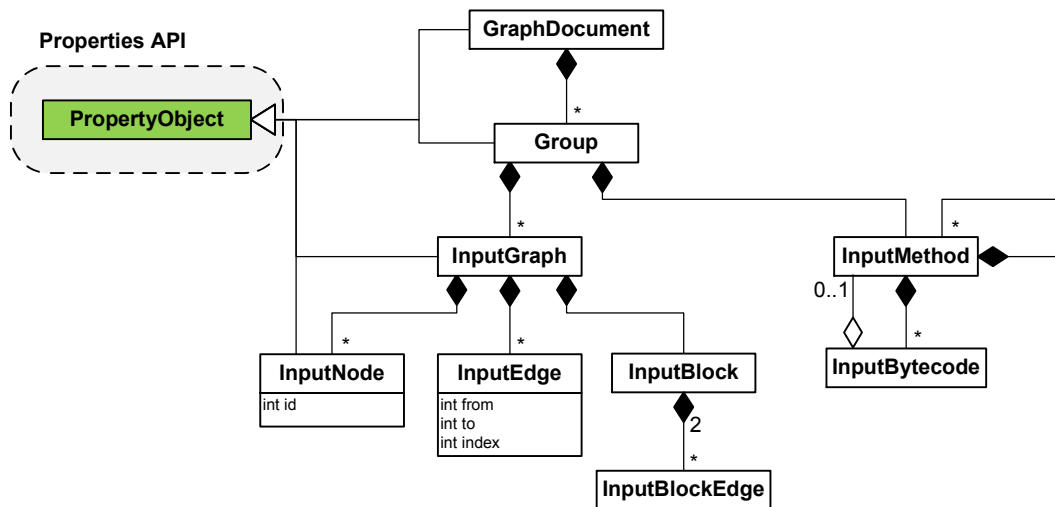


Figure 5.4: Data module class diagram.

5.2.2 Display Model

The display model is similar to the data model, but contains display information for the nodes and edges of the graph. When a new editor window is opened, the application first creates a new display graph based on the data model graph. Then the filters are applied to the display graph and it is drawn on the screen.

There is a semantic difference concerning the graph between the data and the display model: In the data model every node produces exactly one output, so it has exactly one output slot. This is due to the structure of the ideal graph in the server compiler. It is however possible that a filter combines several nodes into a single one with more than one output slot. Therefore, this can be expressed in the display model, but cannot be expressed in the data model. Additionally, slots can have a shortcut that is displayed on the screen and a short description that is shown as a tooltip.

Figure 5.5 shows the classes of the display model. The class `Diagram` corresponds to the class `InputGraph` in the data model. A diagram can contain any number of `Figure` objects.

Figures can have any number of input and output slots. A connection is always between exactly one `InputSlot` object and one `OutputSlot` object.

A figure has a `Source` object that points to one or more `InputNode` objects of the data model that this figure represents. When the display model representation is constructed from the data model representation, every `Figure` object has exactly one `Source` object pointing to one `InputNode` object. Filters may alter the graph and are responsible to keep the pointers to the data model up-to-date. The filter that combines multiple `Figure` objects into a single one sets the node pointers of the `Source` object of the new figure to the union of the node pointers of the `Source` objects of the original figures.

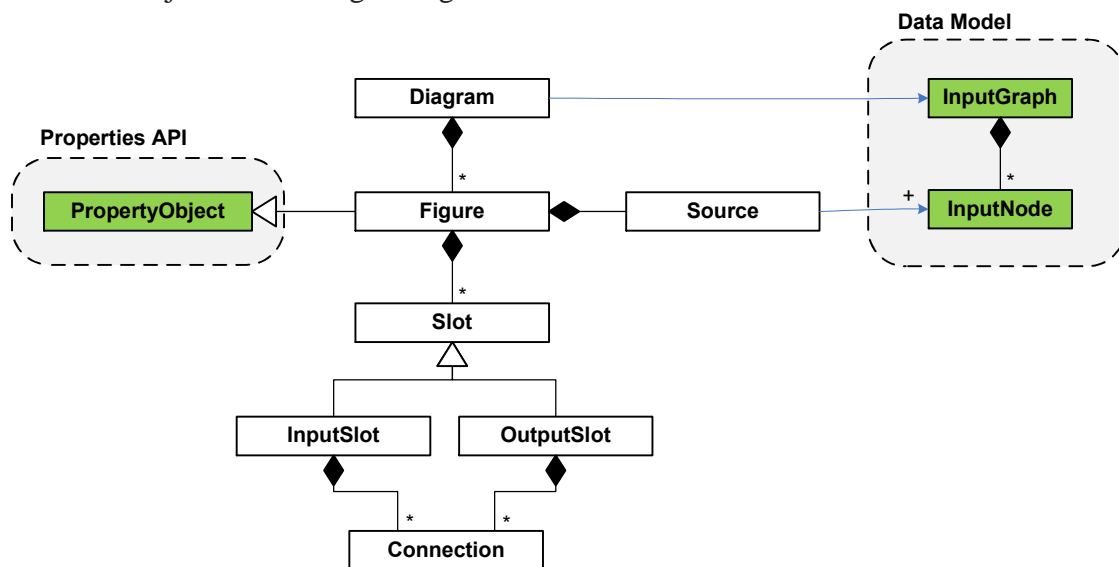


Figure 5.5: Display model class diagram.

5.2.3 Layout Model

The layout algorithm should be commonly usable, so a dependency on the display model should be avoided. Converting the graph in the display model representation to another data structure is however computationally intensive. So the layout model is a submodel of the display model. This is implemented using the Java interface mechanism. Figure 5.6 shows a class diagram of the layout model.

In the layout model, a graph consists of `Vertex` objects that have a size and a position. `Port` objects are assigned to a `Vertex` object and have a position relative to the `Vertex` object. A `Link` object represents an edge between two `Port` objects. Each `Link` object has a list of points in which the first and last are the start and end points of the edge. The other points are interpolation points. The value `null` in the point list is valid and means that the edge should be cut off at the previous point and resumed at the next point in the list.

There is an additional aspect in the layout model that is relevant to layout algorithms that allow clustering of nodes. Each vertex is assigned to a `Cluster` object. A `Cluster` object itself can have a parent `Cluster` object and preceding and succeeding `Cluster` objects.

`LayoutGraph` is a concrete class that represents a layout graph consisting of `Link` objects. The `Port` objects and `Vertex` objects are implicit defined by the `Link` objects as unconnected vertices are not interesting for layouting. A `LayoutManager` is capable of layouting the graph by setting the positions of the `Vertex` objects, the point lists of the `Link` objects and the bounds of the `Cluster` objects.

The `Figure` class of the display model implements the `Vertex` interface. The `Slot` class implements the `Port` interface and the `Connection` class implements the `Link` interface. The cluster information is only available in the data model and is not converted to the display model. The block of a `Figure` object is retrieved by getting the block of the `InputNode` objects the figure was created from. When the `InputNode` objects of a figure are in different blocks, an arbitrary block is chosen.

A diagram can be given to a `LayoutManager` object by creating a new `LayoutGraph` object and adding all `Connection` objects of the diagram to the `LayoutGraph`. There is no need to reconstruct the graph.

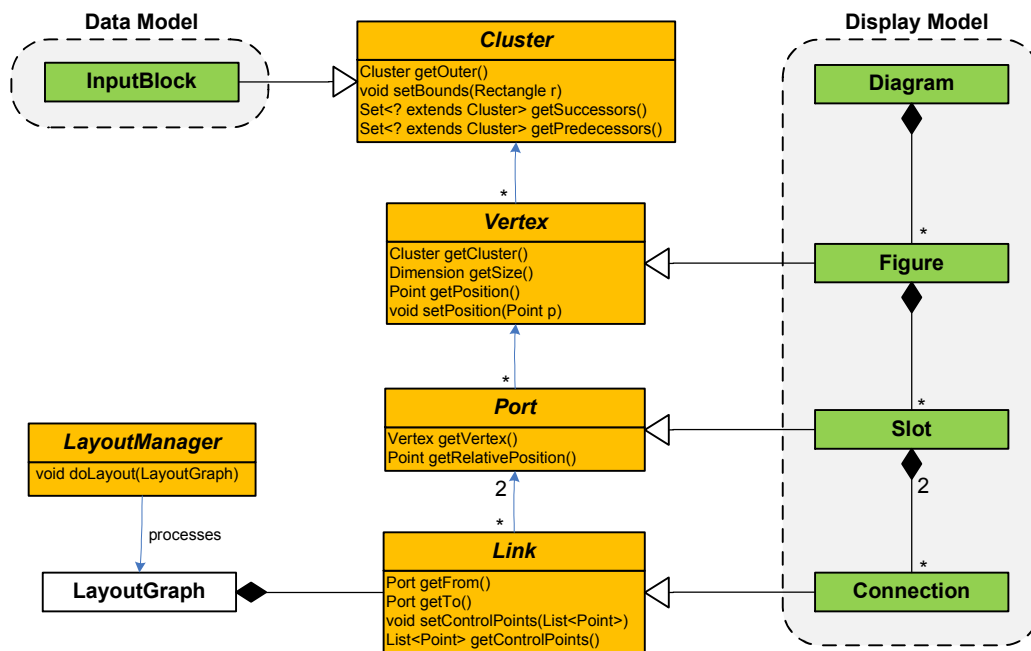


Figure 5.6: Layout model class diagram.

5.3 Properties and Selectors

The mechanism for handling properties is uniform throughout the application. This makes the application more robust against modifications and increases coding efficiency. Properties are stored in the most general form as key-value pairs where the key and the value are both Java `String` objects.

Figure 5.7 shows parts of the architecture that are related to properties and how they interact. The Properties API defines the basic classes for property handling. The class `Property` consists of two `String` fields representing the name and the value of a property. A `Properties` object is a collection of `Property` objects. `Provider` is an interface for objects that offer a `Properties` object. `PropertyObject` is a simple implementation of the interface. By subclassing from `PropertyObject` instead of `Object`, a class can define that its objects have attributes specified as key-value string pairs.

The Properties Window is implemented using the NetBeans lookup mechanism (see Section 2.5). It listens for `Provider` objects in the lookup of the current active window. When such objects are available, it updates its view to show their properties. Therefore, there is no difference for the Properties Window if a user selects an item in the Outline Window or a node in an editor window. In both cases, it retrieves `Provider` objects and displays the corresponding key-value string pairs.

Other parts of the application that make use of properties are the search panel and the filters. In both cases, a set of nodes of the graph need to be selected. For this purpose there exists an interface `PropertyMatcher`, which can match string key-value pairs. Objects of type `PropertyMatcher` specify the name of the property they are matching and have a function `match`. For checking whether a `PropertyObject` matches or not, the value of the property with the name associated with the `PropertyMatcher` object is given to the `match` function as an argument. The `PropertyMatcher` object returns `true` or `false`.

There are two different implementations of the `PropertyMatcher` interface available: The `StringPropertyMatcher` checks whether the value equals a given string value. The more complex class `RegexpPropertyMatcher` allows regular expressions for the value. The matching of `Figure` objects is possible, because `Figure` subclasses `PropertyObject`.

The Selector API defines a `Selector` interface with a single method that returns a list of figures of a diagram. The `MatcherSelector` has an associated `PropertyMatcher` and returns all figures of the diagram for which the `match` function returns `true`. The subclass `OrSelector` returns the union of the result of two selectors. The `AndSelector` returns the conjunction of the result of two selectors. The `NotSelector` returns the inverse of the result of its inner selector. The `SuccessorSelector` returns the immediate successors of the result of its inner selector, and the `PredecessorSelector` returns the immediate predecessors of the result of its inner selector.

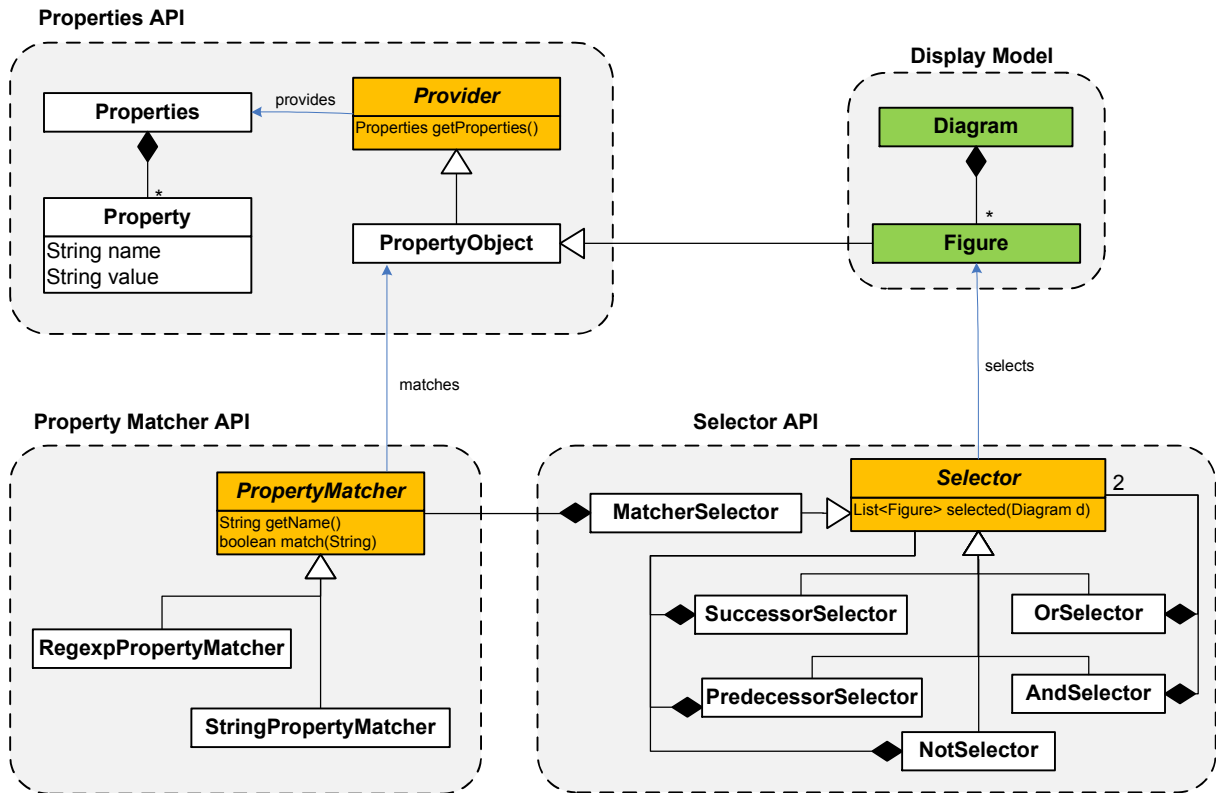


Figure 5.7: Properties and selectors class diagram.

5.4 Filters

When designing the architecture of the filters, the main goal was to make them highly customizable. The application should easily adapt to changes in the server compiler and even be able to display and filter completely different graph data. The basis for this goal is the properties mechanism explained in Section 5.3. The attributes of the nodes in the graph are not hard-coded as Java fields, but represented by key-value string pairs. Filters define rules that apply a certain function on a set of nodes. The set of nodes is selected using a `Selector` object. A filter takes a `Diagram` object and modifies it based on the rules.

Predefined filters are available for coloring nodes, coloring edges, splitting nodes, combining nodes, removing input edges and removing self edges. The `CustomFilter` is a special filter that carries out the changes of the graph based on JavaScript code. The available JavaScript functions and their effects are explained in Section 4.5.

The order in which the application applies the filters is important. For example, if the affected sets of nodes of two color filters intersect, the order in which they are applied changes the appearance of the graph. So the Filter Window lets the user not only activate and deactivate the available filters, but also change their processing order. A `FilterChain` object contains a list of ordered `Filter` objects. The filter chain can apply its filters in the specified order to

a diagram. The `EditFilterCookie` class is used to add an edit action to the filters that is available via the context menu or via double clicking on a list entry.

The Java 6 scripting mechanisms are used to execute the JavaScript code of the custom filters. The Rhino Scripting Library registers a JavaScript scripting engine. The custom filter uses the `ScriptEngineManager` class to retrieve the registered `ScriptEngine` object that is capable of executing the code. The application provides several JavaScript functions that apply filters to the graph. The functions are explained in the user guide in Section 4.5.

The listener pattern is frequently used in the tool. Interested listeners register themselves at some object to be notified when the object changes. In Java this is often implemented by adding `addListener`, `removeListener`, and `notifyListener` methods and a list that stores the listeners to a class. The `ChangedProvider<T>` class can be used to avoid to program the three methods over and over again. It is also not necessary to define new interfaces for every listener. The `Filter` class, for example, has a field of type `ChangedProvider<Filter>` that is accessible via a method. Objects that implement the `ChangedListener<Filter>` interface can register at the `ChangedProvider<Filter>` object and are then informed of changes by the `Filter` object.

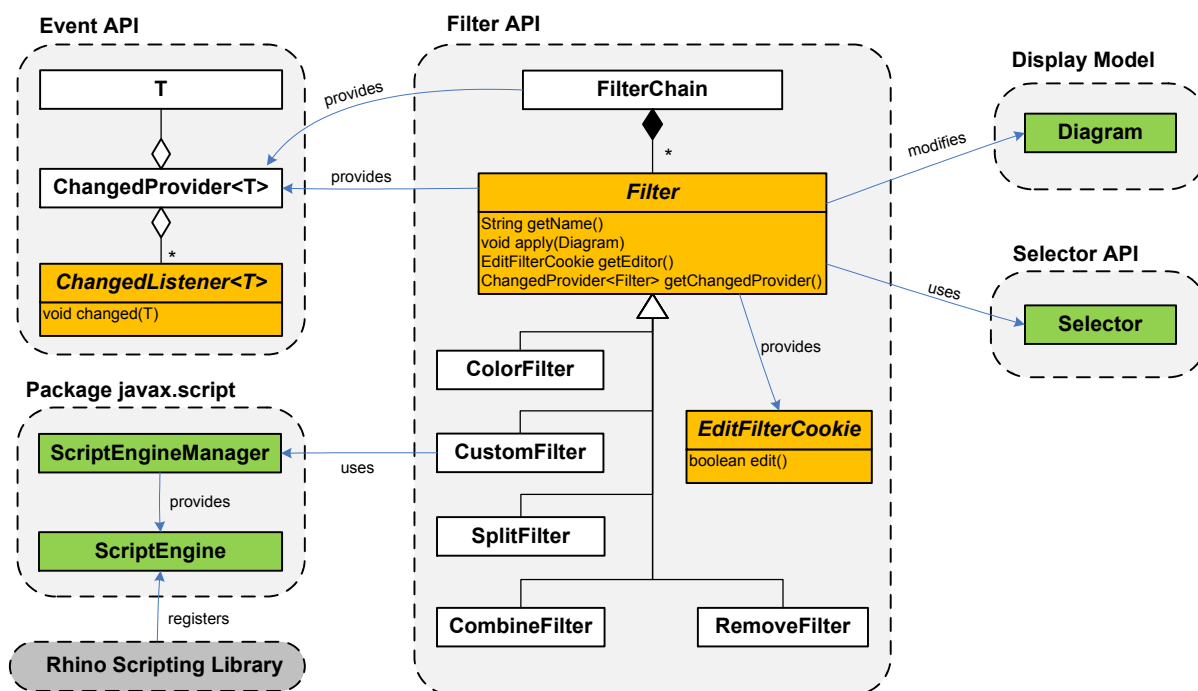


Figure 5.8: Filters class diagram.

5.5 Difference Algorithm

The displayed graphs are snapshots of the data structure of the server compiler during the compilation of a method. The difference algorithm takes two graphs as the input and outputs a difference graph. The nodes of a difference graph have a special state property for every node. This property can be either `same`, `changed`, `new` or `deleted`. The Difference Coloring filter (see Section 4.5) can be used to color the nodes according to their state. Additionally, each edge gets either the state `same`, `new`, or `deleted`. The algorithm is invoked on the input graph coming from the server compiler, before it is converted to the display model.

Optimal graph matching is an NP-complete problem, so for the large graphs of the application it is only reasonable to solve it using heuristics. Creating a difference graph means choosing related pairs of nodes (u, v) where u is a node from the first graph and v a node from the second graph. Every node may occur in at most one such relation. All nodes in the first graph that do not appear in a relation are marked as `deleted`. All nodes in the second graph that do not appear in a relation are marked as `new`. Nodes that appear in a relation are either marked as `same` if they do not differ in important properties or otherwise as `changed`.

Two edges starting at nodes $(n1, n2)$ and ending at nodes $(m1, m2)$ are marked as `same` if the pairs $(n1, n2)$ and $(m1, m2)$ appear in the matching relation calculated for the nodes and their end slot index is equal. All other edges of the first graph are marked as `deleted`, those of the second graph are marked as `new`.

The difficult task is to select pairs of nodes to be related. When the two input graphs are both snapshots of the same method, this is straightforward. In the server compiler, the nodes are allocated on the heap and therefore have a unique memory address throughout their lifetime. This address is used as a unique identifier for the nodes. The difference algorithm adds a pair of nodes (u, v) to the matching relation if their unique identifier is equal.

For two arbitrary graphs, the task of choosing "good" pairs to be matched becomes more difficult. The implemented algorithm is based on a cost function for matching a pair of nodes (u, v) . The cost is calculated by comparing their properties, predecessors, and successors. First, the set of possible matches is built by adding all theoretic possibilities. To reduce the total number of inserted pairs, nodes that differ in the important property "name" are not added, so they will never be matched.

Then the algorithm selects the pair (u, v) with the least cost from the set of possible matches. The selected pair (u, v) is added to the set of matching nodes. Afterwards, all pairs containing either the node u or v are removed from the set of possible matches. Then the algorithm continues by choosing the next pair (u, v) with the least cost. If the least cost for matching a pair of nodes exceeds a predefined threshold or the set of possible matches is empty, it is no longer reasonable to match nodes and the algorithm terminates.

Listing 5.1 shows the difference algorithm for two arbitrary graphs in pseudo-code. The most difficult part of the implementation is to find a "good" function that computes the penalty for matching two nodes. The quality of this function determines the result of the matching.

Listing 5.1 Algorithm for calculating the difference between two graphs.

```

Difference: graph a, graph b
s = set of possible node pairs
result = empty set of node pairs
while s not empty do
  (n, m) = best matching pair of s
  if penalty of (n, m) > THRESHOLD then
    return result
  end if
  add (n, m) to result
  remove all tuples (n, ?) and (?, n) from s
  remove all tuples (m, ?) and (?, m) from s
end while
return result
    
```

Figure 5.9 displays the difference graph for two graph snapshots of an example method. The Root node has changed and the If, Region and Phi node were replaced by the CMoveI node. New edges are drawn as thick lines, while deleted edges are drawn as dashed lines.

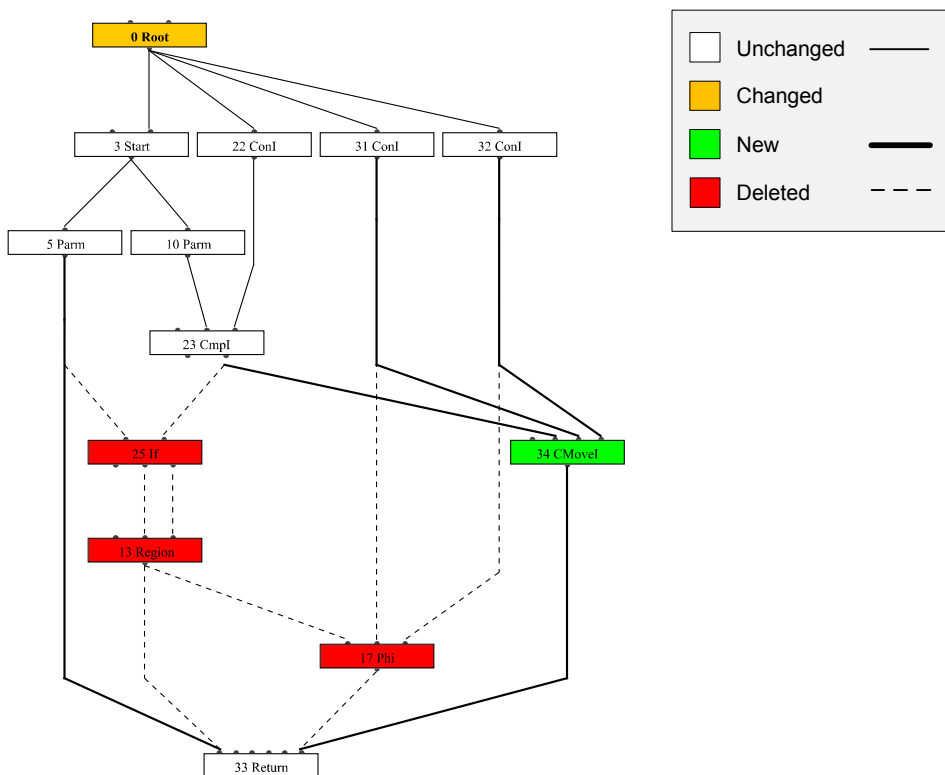


Figure 5.9: The difference algorithm applied on two example graphs.

Chapter 6

Hierarchical Graph Layout

This chapter describes a hierarchical layout algorithm and two different approaches for the assignment of x-coordinates. The data model used as input for the algorithm are presented in Section 5.2.3. The layout algorithm assigns a position to each node and a list of control points to each edge.

6.1 Why Hierarchical?

Most layout algorithms are either tree-based, hierarchical, force-directed, or planar. Choosing the type of layout algorithm has a high impact on the understandability of a graph. Additionally, some algorithms require a graph to be in a certain form: Tree-based layouting can only be applied on trees, hierarchical layouting on directed acyclic graphs, and a planar layout algorithm only works when the graph is planar. Preprocessing steps can change a graph such that it fulfills the requirements of the layout algorithm. In a postprocessing step, the results must then be mapped back to the original graph.

The graph that is displayed by the visualization tool is a directed graph, which has cycles in most cases. The tool uses a hierarchical layout algorithm, as it fits best for the visualization of a program dependence graph. The advantage of using a hierarchical algorithm is that there is a natural flow of data and control in one direction. The only preprocessing needed is to remove directed cycles. The tool layouts the nodes from top to bottom. Nodes that were generated from a bytecode with a low index are likely to be in the top section, those that were generated from a bytecode with a high index are likely to be in the bottom section of the graph. So there is a local coherence between the position of a node and the Java bytecode that it was generated from. The root node is always at the top. Edges that must be reverted to create a cycle-free graph are called backward edges and are drawn in a special way. They arise mostly due to variables that are changed in a loop.

A negative aspect of using a hierarchical layout is that the total length of the edges is higher compared to a force-directed approach. This is especially critical for large graphs, because

some of the edges nearly span from the top to the bottom of the graph. Section 6.11 describes a solution for this problem.

6.2 Processed Steps

The layout algorithm performed by the visualization tool is an adaptation of the algorithm described in [11][12]. The algorithm performs the steps shown in Figure 6.1. These steps are common to most hierarchical layout algorithms. There are however differences in how the steps are performed, especially the crossing reduction and the x-coordinate assignment are varied. The sections in this chapter explain the steps in detail and present two different algorithms for assigning x-coordinates. The following list introduces the steps applied by the algorithm:

Input Graph: First, a copy of the input graph is constructed. This is necessary as the algorithm alters the graph and needs a lightweight graph data structure for fast processing. There is always a mapping back to the original graph to be able to write the result back in the last step.

Remove Cycles: The only condition for the graph to apply a hierarchical layout is that it does not contain any directed cycles. Edges in the intern graph are reversed to make the graph cycle-free. In Figure 6.1, the three nodes B, D, and F form a cycle. The algorithm selects one of the edges involved in the cycle, e.g. the edge from B to F, and reverses it. In the last step, this is taken into account and the result edge goes in the correct direction.

Assign Layers: This step assigns a number to each node, indicating the *layer* to which it belongs. A layer consists of a set of nodes that are placed in the same row. The postcondition of this step is that each edge ends at a node whose layer number is higher than the layer number of the start node. All edges point in one direction.

Insert Dummy Nodes: The crossing reduction and x-coordinate assignment steps require that an edge only connects nodes of adjacent layers. Therefore, intermediate nodes are inserted when an edge is longer. The position of these nodes in the final layout form interpolation points for the corresponding edge. In the example graph, the edge from B to F spans across two layers, so one intermediate dummy node is inserted.

Assign Y-Coordinates: Assigning an y-coordinate to a node basically means to multiply the layer number with the size of a layer. For a good layout, it is also reasonable to center the nodes within a layer and to allow variable layer heights.

Reduce Crossings: The number of crossings in the final drawing can be reduced by reordering the nodes within their layers. After this step, the order of the nodes within a layer is not changed any more. The initial ordering in the example graph had three crossings while there are no crossings at all after reordering the nodes in the third layer.

Assign X-Coordinates: Retaining the ordering constraints given by the crossing reduction, this step assigns x-coordinates to the nodes. The goal of this step is to minimize the total length of all edges and producing a symmetric layout.

Result: After all steps are applied, the result calculated for the intern graph is written back to the original graph. The algorithm flips again all edges reversed in the cycle-removal step, i.e. the edge from B to F, and converts dummy nodes to interpolation points.

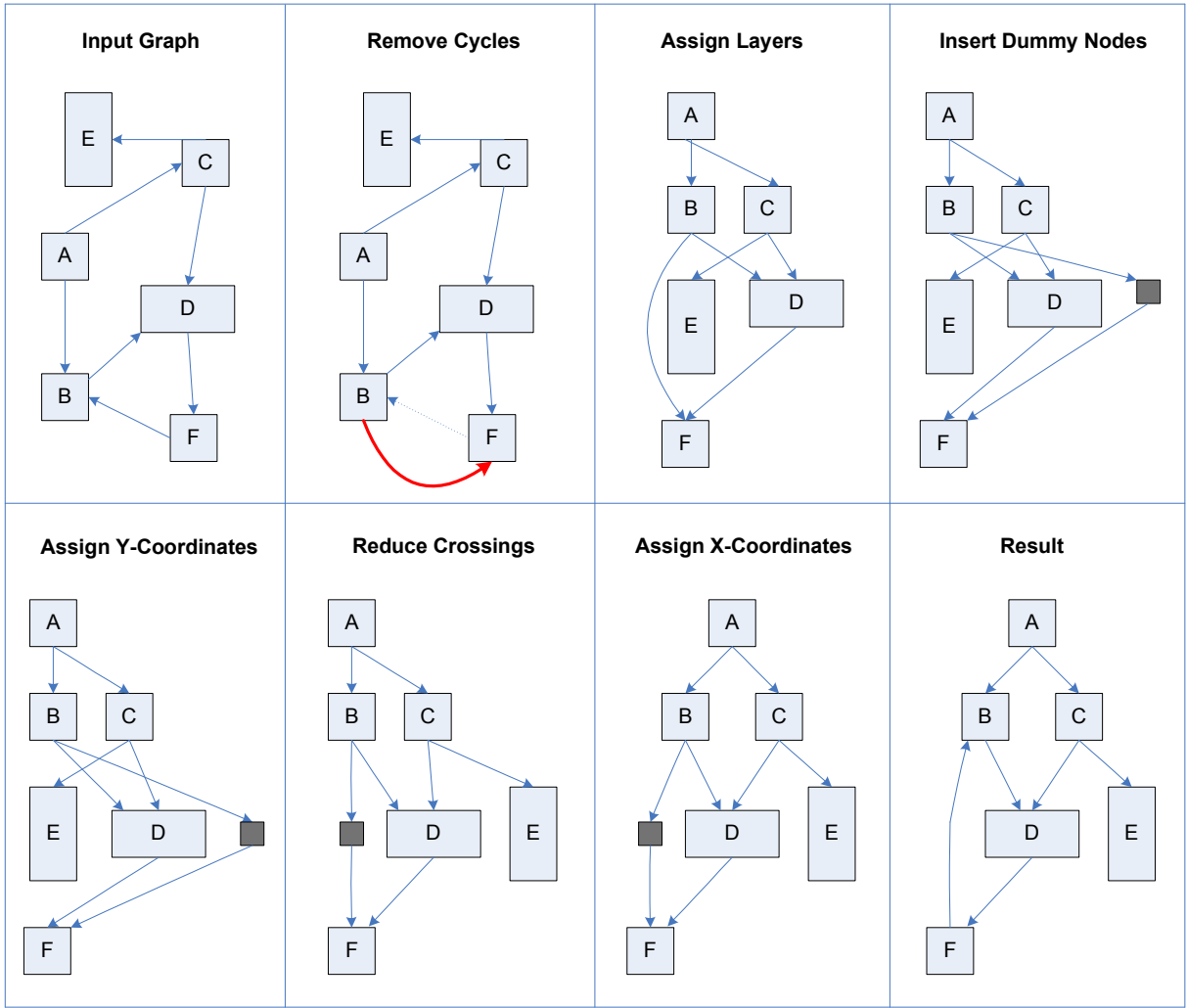


Figure 6.1: Steps of a hierarchical layout algorithm.

6.3 Breaking Cycles

The hierarchical layout algorithm can only process cycle-free graphs, because it is impossible that all edges point in one direction when a graph contains cycles. There are usually different sets of edges that can be reversed to make the graph cycle-free. When deciding between two different sets, either the total number of reversed edges or the length of all edges could be a good criteria.

The algorithm used by the tool does not use any sophisticated selection strategy for the reversed edges, but performs a simple depth first search. This is appropriate for program dependence graphs, because reversed control flow edges usually are due to jumps back to the loop header. The algorithm runs in linear time to the number of edges.

Listing 6.1 shows the pseudo-code of the cycle-removal algorithm. A depth-first search is started at every node without an incoming edge. When there is no such node, the depth-first search is started at the node specified by the `root` parameter. This guarantees that every node is reached. The `visited` flag of a node ensures that a node is visited at most once. The currently active nodes form a path from the start node without input edges to the current node.

When the algorithm finds an edge between the current node and an active node, it knows that this edge is part of a directed cycle and reverses it. Such edges are called *backedges*.

Edges between the current node and a visited node, which is not active, are no problem as these edges would only be part of cycles if the edges of the graph were undirected. Such edges are called *crossedges*.

Listing 6.1 Algorithm for breaking cycles in a directed acyclic graph.

```

BreakCycles: directed graph g, node root
  for each node n without any incoming edge of g do
    BreakCyclesRecursive(n)
  end for
  if root was not visited then
    BreakCyclesRecursive(root)
  end if

BreakCyclesRecursive: node n
  mark n as visited
  mark n as active
  for each outgoing edge e of n
    t = target node of e
    if t is active then
      reverse edge e
    else if t was not visited then
      BreakCyclesRecursive(t)
    end if
  end for
  unmark n as active

```

Figure 6.2 shows the cycle-removal algorithm applied step-by-step to the example graph. The algorithm starts with the node A, because this node has no input edges. Afterwards B, D, and F are visited and also marked as active. The edge between F and B now connects two active nodes. Therefore, it is a backedge and needs to be reversed. As all inputs of F have been processed, the active flag is cleared and the algorithm continues by processing D. This node also has no unvisited outgoing edges, so its active flag is cleared too. When processing B, the edge between B and F forms a crossedge and is therefore ignored. Nodes C and E are processed in a similar matter, the crossedge between C and D is ignored again.

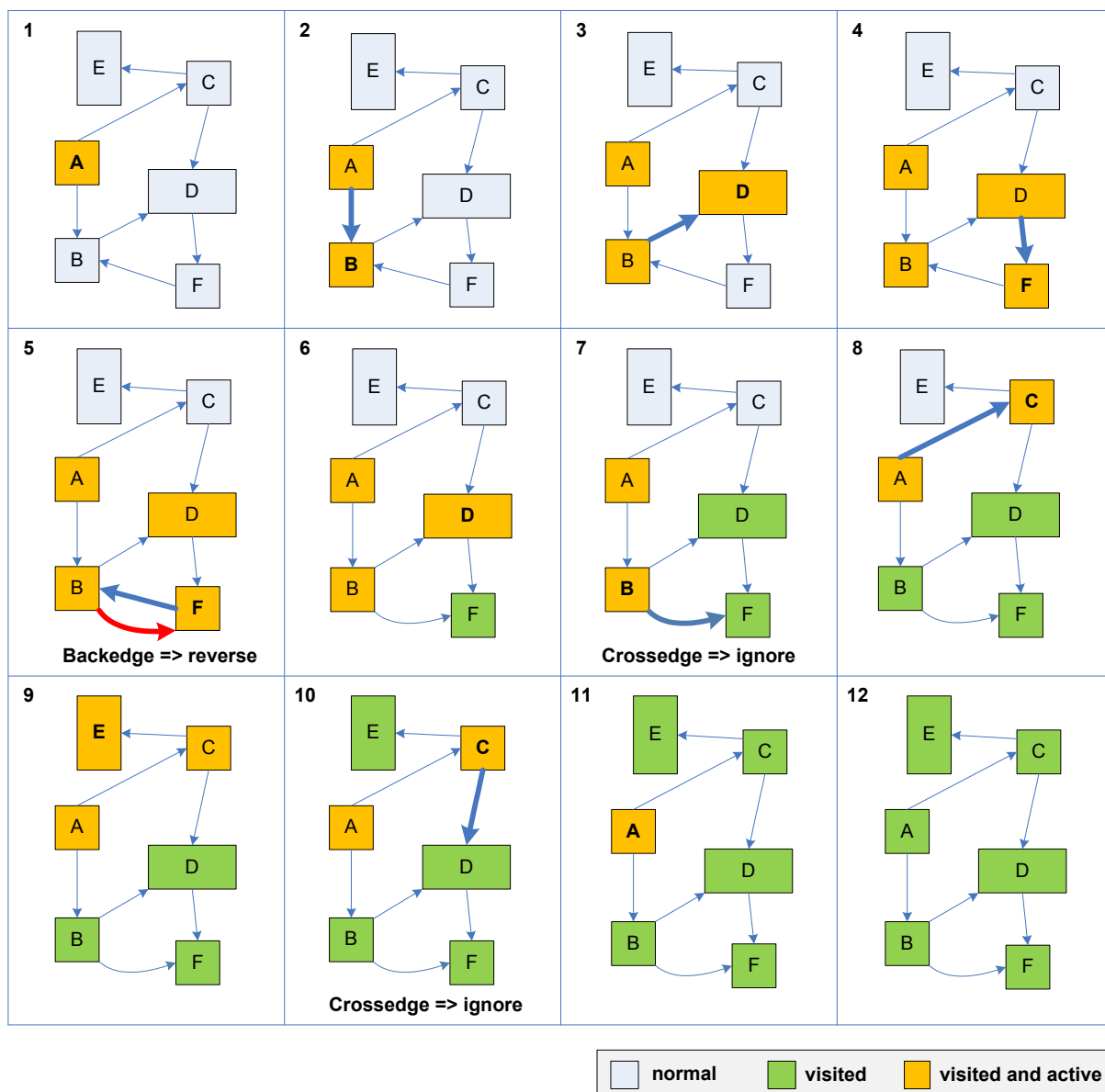


Figure 6.2: Breaking cycles of an example graph.

Reversed edges possibly need additional interpolation points. The tool always draws the edges such that they start at the bottom of a node and end at the top of a node. Therefore backedges need a special treatment, which is described in Section 6.10. Additional problems arise when the graph contains *self-loops*, i.e. nodes that are connected with themselves. It makes no sense to reverse a self-loop edge because it would not break any cycle. Therefore, self-loops must be deleted from the graph before the cycle-removal step. The visualization tool currently does not draw self-loops at all.

6.4 Assign Layers

After preprocessing the graph such that it is free of cycles, all nodes are assigned to layers. Layers are numbered starting with one. The postcondition of this step is that for any edge going from node n to node t , the layer of node n has a lower number than the layer of node t .

A subgoal of layer assignment is to minimize the total number of layers. The length of the longest directed path through the graph without visiting a node twice plus one is a minimum bound for the number of necessary layers. Every edge in the path must go from a node in layer i to a node in layer $i+1$. So we need at least one layer for every end of an edge and one additional layer for the start node.

Listing 6.2 Algorithm for assigning layers.

```
AssignLayers: graph g
  s = new empty set of nodes

  for each node n without any incoming edge of g do
    layer of n = 1
    add n to s
  end for

  i = 1
  while s is not empty do
    t = new empty set of nodes

    for each node p in s do
      for each node n in successors of p do
        if all predecessors of n are already assigned to a layer then
          add n to t
        end if
      end for
    end for

    assign layer i to all nodes in t
    s = t
    i = i + 1
  end while
```

Listing 6.2 presents a simple layer assignment algorithm in pseudo-code. First, it assigns all nodes without input to the first layer. In each loop iteration, it assigns nodes whose predecessors have all been processed before to a new layer. This way, the postcondition is guaranteed. Any incoming edge of a node starts at one of its predecessors. When all the predecessors have been processed before, they all are assigned to layers with a lower number fulfilling the condition. The algorithm terminates when there are no more unassigned nodes left.

All nodes are processed by the algorithm. Nodes without an incoming edge are automatically assigned to the first layer. For all other nodes, there must be a last predecessor that is assigned to a layer as the graph is cycle-free. One iteration after the last predecessor is assigned to a layer, the node itself gets assigned to a layer.

Figure 6.3 shows the layer assignment algorithm applied to the example graph. Node A is assigned to the first layer. In the next step, the graph without the already processed nodes is shown. In the second graph, the two nodes B and C have no unprocessed predecessor and are therefore assigned to layer two. E and D are assigned to the third layer and node F is assigned to the last layer.

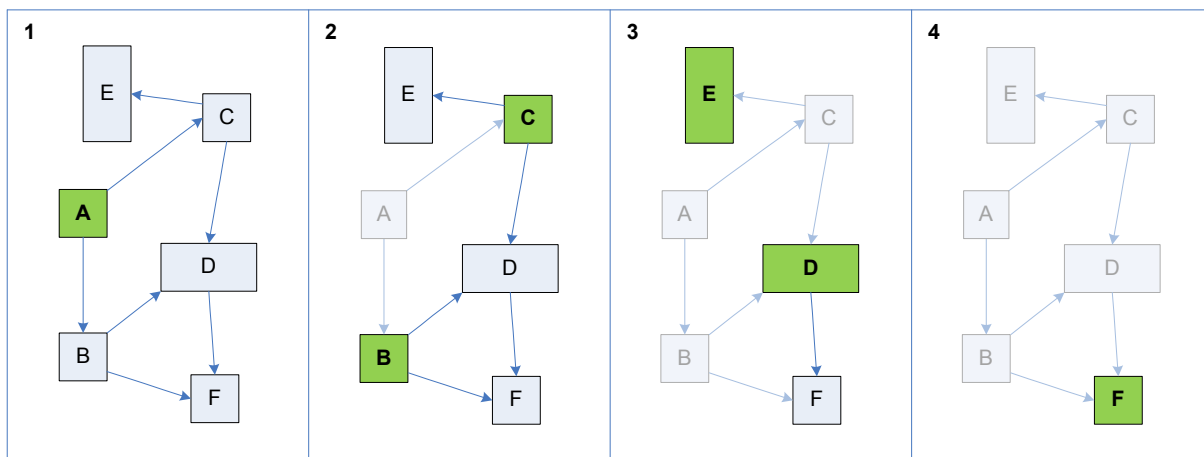


Figure 6.3: Assigning layers for the nodes of the example graph.

6.5 Insert Dummy Nodes

For the crossing reduction and x-coordinate assignment steps to work properly, a stronger condition for all edges must be met: Every edge must go from a node of layer i to a node of layer $i+1$. This is done by inserting *dummy nodes* to the graph. These node have no corresponding node in the input graph, but are later converted to interpolation points of the original edge. Their height and width is set to 0. The algorithm is presented in Listing 6.3. Every edge is checked and intermediate nodes and intermediate edges are inserted whenever necessary. In the example graph, only one dummy node must be inserted between the nodes B and F.

The tool has an additional ability that tries to layout edges coming from the same node as long as possible as a single edge. The dummy insertion step is responsible for this behavior. When there is already a dummy node for an edge coming from the same node and the same port in the current layer, this dummy node is shared among the two edges.

Listing 6.3 Algorithm for creating dummy nodes.

```
CreateDummyNodes: graph g

  for each edge e of g do

    s = source node of e
    t = target node of e

    i = layer of s plus 1
    last = s

    while i not equal to layer of t do
      d = new dummy node in layer i
      remove connection between last and t
      add connection between last and d
      add connection between d and t
      last = d
      i = i + 1
    end while

  end for
```

6.6 Assign Y-Coordinates

The assignment of y-coordinates benefits from assignment of nodes to layers. All nodes of the same layer should lie on a horizontal line. Nodes can have different height, so they are centered within the rectangular space of their layer.

Listing 6.4 shows the pseudo-code of the algorithm. The maximum height of a node within a layer is calculated. All nodes are aligned according to this height.

Listing 6.4 Algorithm for assigning y-coordinates.

```

SetYCoordinate: graph g
  base = 0
  for each layer l of g do
    max = maximum height of a node in layer l
    for each node n of layer l do
      y coordinate of n = base + (max - height of n)/2
    end if
    base = base + max + vertical node offset
  end for

```

Figure 6.4 shows the values calculated by the algorithm when aligning a node. The node D of the example graph is centered. In its layer, the node E is the node with maximum height.

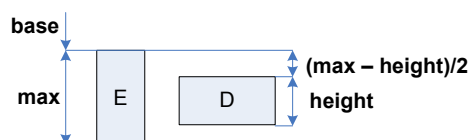


Figure 6.4: Calculations when assigning y-coordinates.

This simple y-coordinate assignment step can be enhanced by allowing a variable vertical offset between the layers. When the edges between two layers connect nodes that have a high horizontal offset, it is reasonable to increase the vertical offset to avoid sharp bends. In this case, the y-coordinate assignment step must be applied after the x-coordinate assignment.

6.7 Crossing Reduction

Until now the order of the nodes within their layer was arbitrary and did not affect any of the processing steps. The crossing reduction step tries to find an order that minimizes the edge crossings of the graph.

Let n_1 and m_1 be two nodes in the first layer and n_2 and m_2 be two nodes in the second layer. Two edges (from n_1 to n_2 and from m_1 to m_2) cross if and only if the order between n_1 and m_1 in the first layer is opposite to the order between n_2 and m_2 in the second layer. To simplify the analysis, the crossing reduction algorithms often optimize the crossings locally between two layers. Finding an optimal solution that minimizes the crossings is NP-complete even for graphs that have only two layers. Therefore, heuristics need to be applied that give good results in the normal case.

The algorithm iteratively sweeps down and up the layers. When sweeping down, the nodes of layer $i+1$ are reordered by looking at the edges from layer i to layer $i+1$. When sweeping up, the

nodes of layer i are reordered by looking at the edges from layer i to layer $i+1$. Listing 6.5 shows the pseudocode for a single downsweep. The new relative position of a node is calculated by forming the mean of the positions of its predecessor nodes. When sweeping up, the successor nodes are used respectively. After calculating the relative position for all nodes in a layer, they are reordered according to that position. The number of iterations is arbitrary, acceptable results can be achieved with only one or two iterations.

Listing 6.5 Crossing reduction algorithm.

```

DownSweep: graph g
  for the second to the last layer of g do
    for each node n in layer i of g do
      sum = sum of the position of all predecessors of n
      crossing number of n = sum / number of predecessors
    end for
    resort the nodes in the list of layer i according to crossing number
  end for
    
```

Figure 6.5 shows the downsweep step at layer 2 and 3 of the example graph. B and C are numbered 1 and 2 according to their position within their layer. The only predecessor of E is C, so E gets a relative position of 2. D has two predecessors, so the mean of the position of its predecessors is calculated and D gets a relative position of 1.5. The dummy node has only one predecessor, so it gets the relative position its only predecessor.

After reordering the nodes, the dummy node becomes the new first node, D stays in the middle and E becomes the new last node of layer three. When calculating the relative positions for the nodes in the next layer, the new positions 1, 2 and 3 are used.

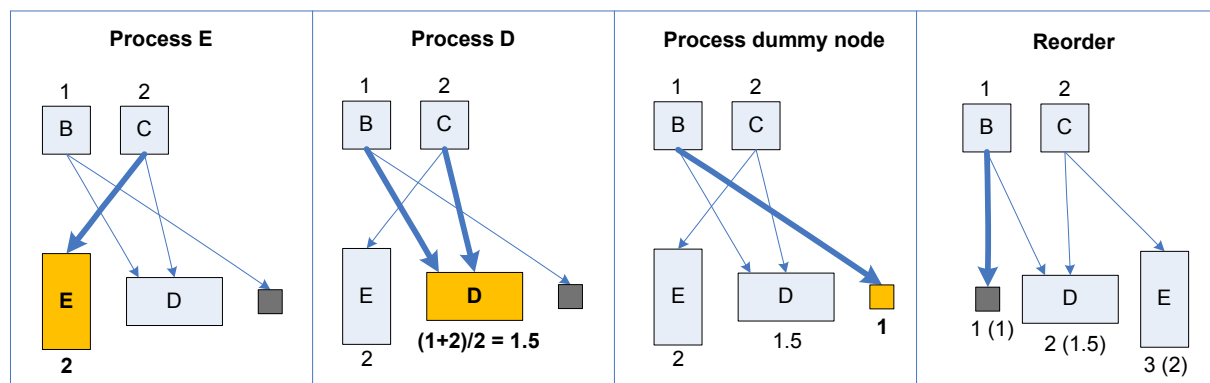


Figure 6.5: Crossing reduction applied on a layer of the example graph.

6.8 Assign X-Coordinates

In this section, two different approaches for assigning x-coordinates to the nodes are presented. The inputs to these algorithms are a graph that fulfills the postconditions of the dummy insertion step, the widths of the nodes, and the horizontal space requirements between nodes.

The goal of x-coordinate assignment is to find an x-coordinate for every node meeting the space requirements of the nodes. A subgoal is to minimize the total horizontal length of all edges. Another consideration is that the graph should look symmetric.

Both methods iteratively improve the layout of the graph, the number of iterations influences the layout quality. The order of the nodes within a layer is taken unchanged from the crossing reduction step.

6.8.1 DAG Method

After the initial layouting, the algorithm sweeps iteratively up and down the graph. In the remaining section only the downsweep is described. The upsweep works in a similar manner, but the outgoing instead of the incoming edges of a node are processed and the successors are used instead of the predecessors.

The initial positioning is arbitrary, however less iterations are needed if the first positioning is reasonable. One possibility is to align all nodes as left-most as possible meeting the space requirements of the nodes.

All nodes of a layer are repositioned in a special sequence: Nodes with less incoming edges are processed first. A node with a low degree in a bad position catches someone's eye more than a node with a high degree in a bad position. Additionally, all edges between dummy nodes are processed first. This way long edges, which are split across several edges connecting dummy nodes, are drawn as straight lines.

Then, the optimal position of a node is calculated as the median of the positions of its predecessors. This minimizes the horizontal length of the incoming edges. If the node has an even number of predecessors, the mean of the two median predecessors is taken to increase the symmetry of the drawing.

Listing 6.6 presents the pseudo-code for the downsweep part algorithm. After the optimal x-coordinate was calculated, positioning a node works exactly the same way for both up- and downsweep.

The final position of a node must take the layout constraints into account. If no other node on that layer is already positioned, we can just set the x-coordinate of the node to the optimal position. Otherwise there is a range of legal positions that must be considered.

Listing 6.6 Algorithm for assigning x-coordinates.

```

DownSweep: graph g
  for i from 2 to layer count of g do
    l = all nodes in layer i
    sort l in increasing order of predecessor count
    for each node n in l do
      pos = median of predecessor positions
      AddNode(n, pos)
    end for
  end for

AddNode: node n, optimal position p
  left = nearest already positioned node to the left of n
  right = nearest already positioned node to the right of n
  minX = position of left + width of left + horizontal node offset
  maxX = position of right minus width of right
  increase minX by width and offset of all nodes between left and n
  decrease maxX by width and offset of all nodes between n and right
  if pos < minX then
    pos = minX
  else if pos > maxX then
    pos = maxX
  end if
  position of n = pos

```

In the code listing, the method `AddNode` gives a pseudo-code representation of how the final position is calculated. Figure 6.6 shows graphically the allowed range for an example node `X`, that should be placed. Nodes `L` and `R` are the first nodes to the left and the right of `X` that are already positioned. `left_offset` is the minimum space required for all nodes between `L` and `X`. `right_offset` is the minimum space required for all nodes between `X` and `R` plus the width of `X` itself.

So the x-coordinate of `X` must be in the range between `minX` and `maxX`. When the optimal position lies outside this range, then the x-coordinate is either set to `minX` and `maxX`, whichever is closer to the optimal position.

Finding the nearest already positioned nodes `left` and `right` can be done in $O(\log(n))$ time, where n denotes the total number of nodes, using a balanced binary tree. The number of necessary iterations to get an acceptable result is low, a value of two is used by the tool. Therefore, the total time complexity of the algorithm is $O(n \cdot \log(n))$.

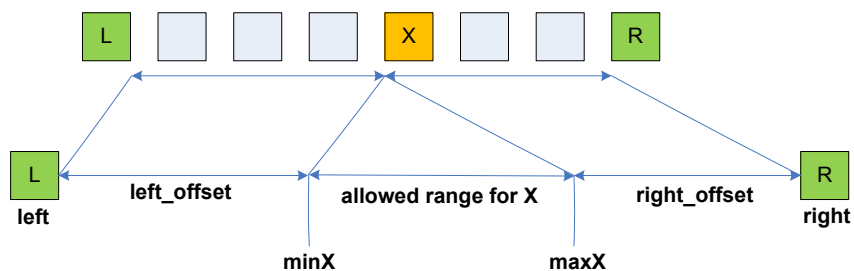


Figure 6.6: X-coordinate assignment example.

6.8.2 Rubber Band Method

Another way to assign x-coordinates to the nodes uses a physical model and is called rubber band method. The algorithm presented in this section was developed by Georg Sander [25][26]. The nodes of the graph are regarded as balls and the edges are rubber bands affecting the nodes. The vertical position of the nodes is fixed, so they can only move horizontally. Long edges represented by dummy nodes would be skewed, so their x-coordinates are fixed to a single value. They can be regarded as sticks.

Listing 6.7 presents the rubber band algorithm in pseudo-code. First, segments are formed out of nodes. A segment is regarded by the layout algorithm as a rectangular entity that can span across several layers. Each segment will get a single x-coordinate assigned. Basically, a segment is represented by exactly one node. Only dummy nodes representing the same edge are merged into a single segment to assure that the edge is drawn as a straight line.

The algorithm constructs a graph with the segments as nodes and an edge between two segments (s and t) if and only if s contains a node that is the immediate left neighbor of a node of t. Based on this graph, a topological ordering of the segments is calculated. Afterwards, they are positioned as left as possible. Segments without predecessors in the intermediate graph are positioned at the minimum x-coordinate. The leftmost possible x-coordinate for the other segments is calculated by taking the maximum of the right boundary of all its predecessors. The topological sorting ensures that the predecessors of a segment are processed before the segment itself is processed.

On this initial ordering, the physical model is now applied iteratively. The algorithm usually produces a reasonable layout faster when in a downsweep only the predecessors of a node and in an upsweep only the successors of a node are taken into account. It is however also possible to incorporate both predecessors and successors in the calculations, in the code listing this way to calculate the forces is used.

Two segments that touch and whose forces cross are combined to a single region to conform better to the physical reality. So the algorithm starts by putting every segment into a single

region. When two such interacting regions are found, they are removed from the set of regions and the combined region is added.

The desired position for a region is its old position plus the force on the region. The region is moved to this position, however stopping when touching any other region. There is no specific processing order of the regions, and the result can vary according to this order. A high number of iterations should however balance this.

Listing 6.7 Rubber-band algorithm for assigning x-coordinates.

```
RubberBand: graph g
  generate segments
  sort segments topologically
  set initial positions
  for i from 1 to MAX_ITERATIONS do
    Process(g)
  end for

Process: graph g
  for each segment s of g do
    force on s = mean of predecessors and successors horizontal offsets
  end for

  regions of g = new set of regions with one region for each segment
  for each region r of g do
    for each region t contacting r on the right do
      if force on r > force on t then
        combine the regions of r and t
      end if
    end for
  end for

  for each region r of g do
    position p = old position of r + force on r
    move r to p stopping when hitting another region
  end for
```

Figure 6.7 shows the main parts of the algorithm. In a preprocessing step, the nodes are grouped into segments. In this example, the two dummy nodes are grouped into a single segment. The third graph shows the edges of the graph used for topological sorting and one legal topological sorting expressed by the numbers of the nodes. It also shows the initial positioning. The fourth graph shows the regions created when performing a downsweep. The segments 2, 3, and 4 have only node 1 as their predecessor. If there were no other segments, the rubber band would move them exactly under the segment 1. But in the example, the three segments would hit another and move together such that they are all three side by side centered under node 1. So the algorithm combines the three segments to a single region. Nodes 6 and 7 represent a similar case. The fifth graph shows the segments with the x-coordinates assigned after the first downsweep.

When continuing with an upsweep, the regions must be recalculated. Now only the two segments 3 and 4 are interacting. After a few iterations, the positions of the segments are translated to positions of the nodes of the original graph.

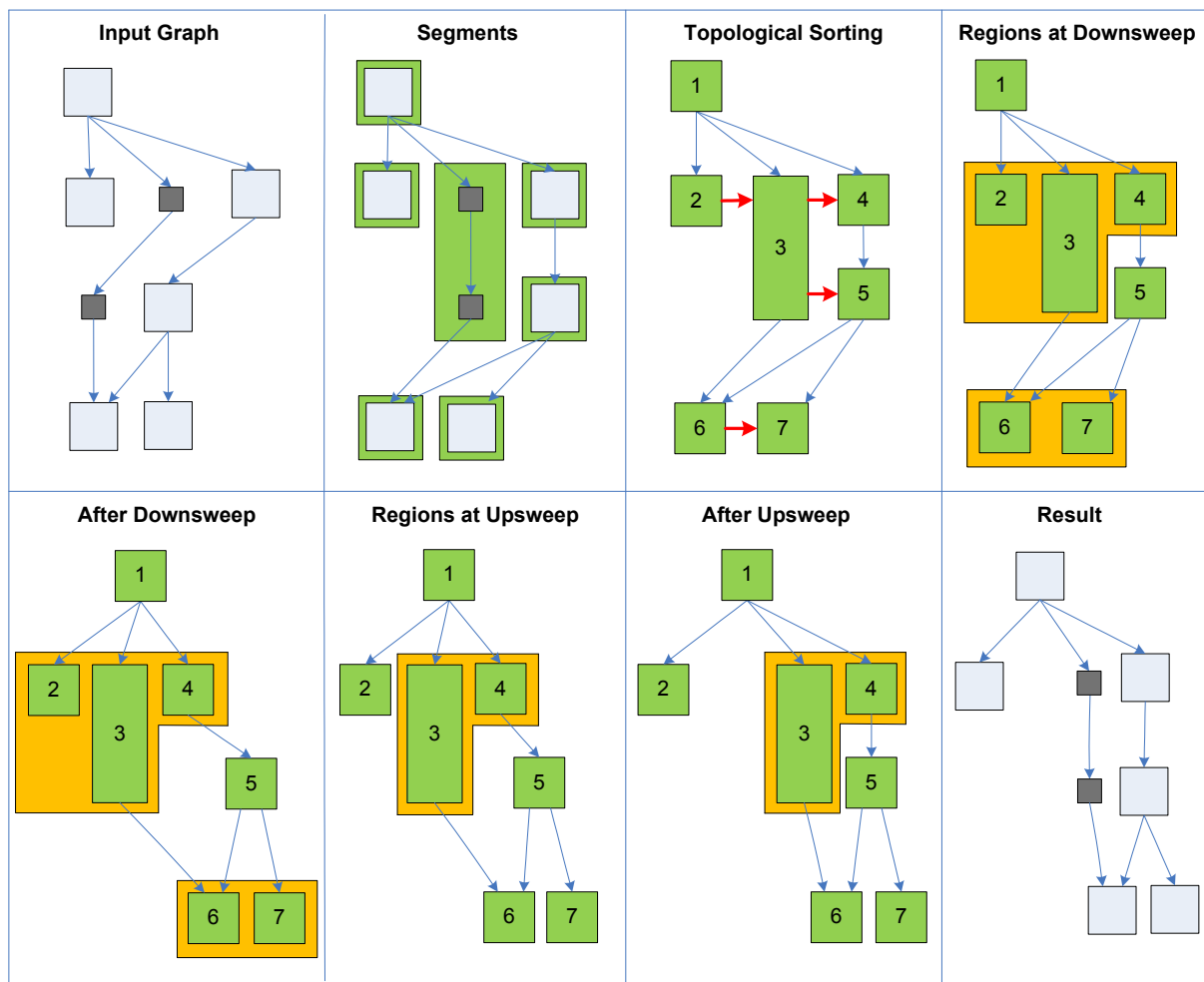


Figure 6.7: X-coordinate assignment using the rubber band method.

6.9 Cluster Layout

The general goal of clustering is to layout nodes of the same cluster close together. The visualization application draws a rectangle around nodes belonging to the same cluster. Therefore, the nodes need to be positioned in rectangular regions, such that the application can draw the cluster rectangles without intersections. This is achieved by first layouting each cluster separately and then performing the hierarchical layout recursively by treating the cluster rectangles as nodes.

Figure 6.8 shows a simple clustered graph with two nodes. Node A is assigned to cluster 1, node B is assigned to cluster 2. There is a connection between A and B. First, the connection is split into three parts by inserting two intermediate nodes. The leftmost graph shows the resulting structure. The connection between A and B is split into an edge going from A to a dummy node, an edge to another dummy node, and a third edge going to B. The dummy nodes have a dual function: When performing the separate layout of the clusters, they are treated just like dummy nodes. However, when performing the global layout, they are slots of the nodes that represent the rectangular cluster areas.

First, the application performs the layout algorithm on the two blocks separately. The dummy node, which is a successor of node A, is always put in the last layer. The dummy node, which is a predecessor of node B, is put in the first layer. After the layout for the two blocks is calculated, the width and height of the rectangle encapsulating all nodes of a cluster is known. Then, an artificial graph is built: The cluster rectangles form the nodes and the two dummy nodes form slots. The hierarchical layout algorithm is applied again on the artificial cluster graph.

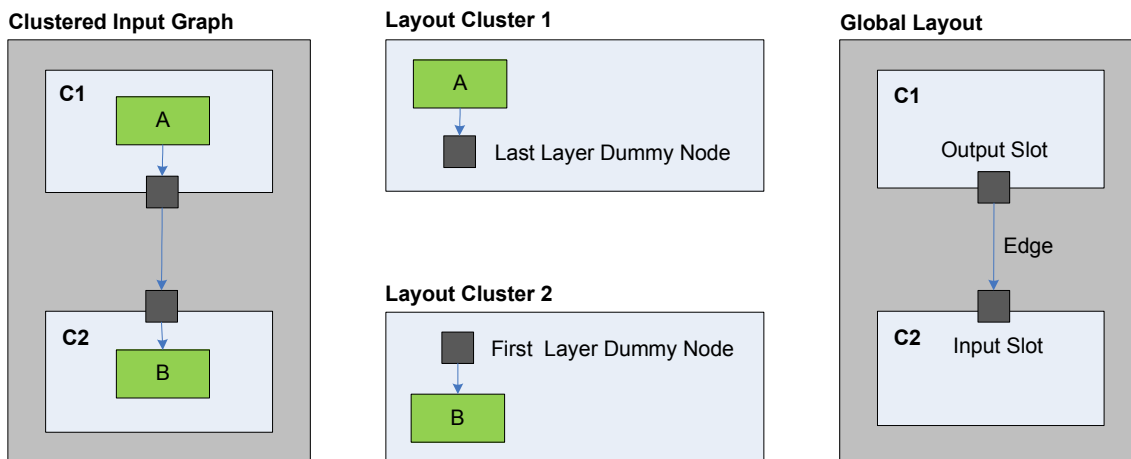


Figure 6.8: Recursive cluster layout.

6.10 Drawing of Backedges

In Section 6.3, an algorithm for reversing edges to make the input graph free of cycles was presented. The visualization tool always paints edges such that they start at the bottom of a node and end at a top of a node. For backedges, a special routing for adding interpolation point is needed.

All backedges are processed in a right-to-left order. The additional interpolation points and the port are calculated as shown in Figure 6.9. A node with backedges will not only get new ports, but also change its size. The horizontal and vertical space needed for the backedges are added to the size of the node.

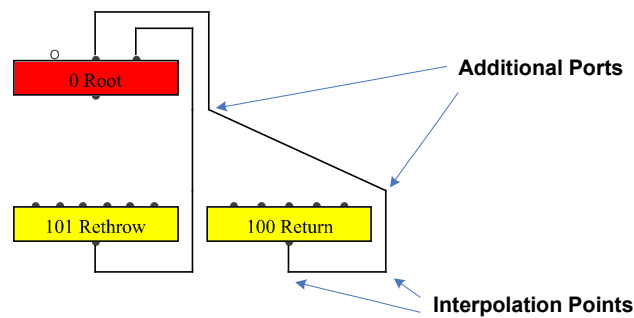


Figure 6.9: Example for the routing of a backedge.

6.11 Optimization for Large Graphs

When the server compiler processes long Java methods, the resulting graph is so large that the time for performing the layout is annoying for the user. The graph of the server compiler contains a high number of edges, which leads to some edges nearly spanning over the whole graph. Long edges are a problem for two reasons: First, a lot of dummy nodes need to be created and later processed by the coordinate assignment and the crossing reduction. Second, they disturb the drawing, because they are represented by long straight lines and the nodes along their way are either drawn on the right or on the left.

The benefit of drawing the long edges is not high, because mostly only a small part of them is visible. An edge that starts and ends outside of the current screen extract is not informative. Therefore, the visualization application cuts long edges and draws only their beginning and ending. This way, the drawing performance as well as the quality of the drawing is improved.

Chapter 7

Compiler Instrumentation

This chapter describes the code added to the Java HotSpot™ server compiler. The current version is based on JDK 7 build 13 [16]. The design goal for the instrumentation was to change as little code of the server compiler as possible and to encapsulate the added code.

7.1 Overview

Figure 7.1 shows the classes added to the server compiler and how they are connected to already existing classes. `IdealGraphPrinter` is the main class that manages the tracing of the graphs. As explained in Section 5.2.1, a graph is serialized as the difference to the previous graph. The server compiler does not store old versions of the graph, so the graph printer object must handle this. `Description` objects are used for this purpose.

When printing a graph, first the `Description` objects of the previous graph are marked as `invalid`. `Description` objects of the new graph start with the state `new`. If the new object equals an old one, then the new one replaces the old one and gets the state `valid`. For an `invalid` `Description` object, only a mark that it has been removed is stored. A `valid` `Description` object needs no storage space at all, as it was serialized with the previous graph. Only for new `Description` objects, the graph printer outputs the full data.

`Description` has two subclasses: `EdgeDescription` and `NodeDescription`. An object of class `EdgeDescription` has a reference to the start and end node of the edge. It stores the input index at which the edge enters the end node. The server compiler works with an object of class `Compile`, which consists of `Node` objects. A `NodeDescription` object stores a reference to the described `Node` object. It is an object that has properties represented by string key-value pairs. The C++ memory address of a `Node` object forms the unique identifier of a node.

The scheduler constructs a control flow graph by grouping the nodes into blocks (see Section 7.2). This process is repeated for every graph. The control flow graph is not stored

difference-based like the normal graph, because a small change in the normal graph can cause a big change in the control flow graph.

Every `CompileThread` object has its own `IdealGraphPrinter` object to avoid multi-threading conflicts. The graph printer uses the `PhaseChaitin` class to get information about the selected registers and life ranges of the nodes (for details on the register allocation see Section 3.6). The `Matcher` class is used to retrieve the matcher flags that identify the subtrees of the graph, which are converted to `MachNode` objects (see Section 3.5).

For actually outputting the data, the printer either uses the `fileStream` class of the server compiler or the `networkStream` class. They are both subclasses of `outputStream`.

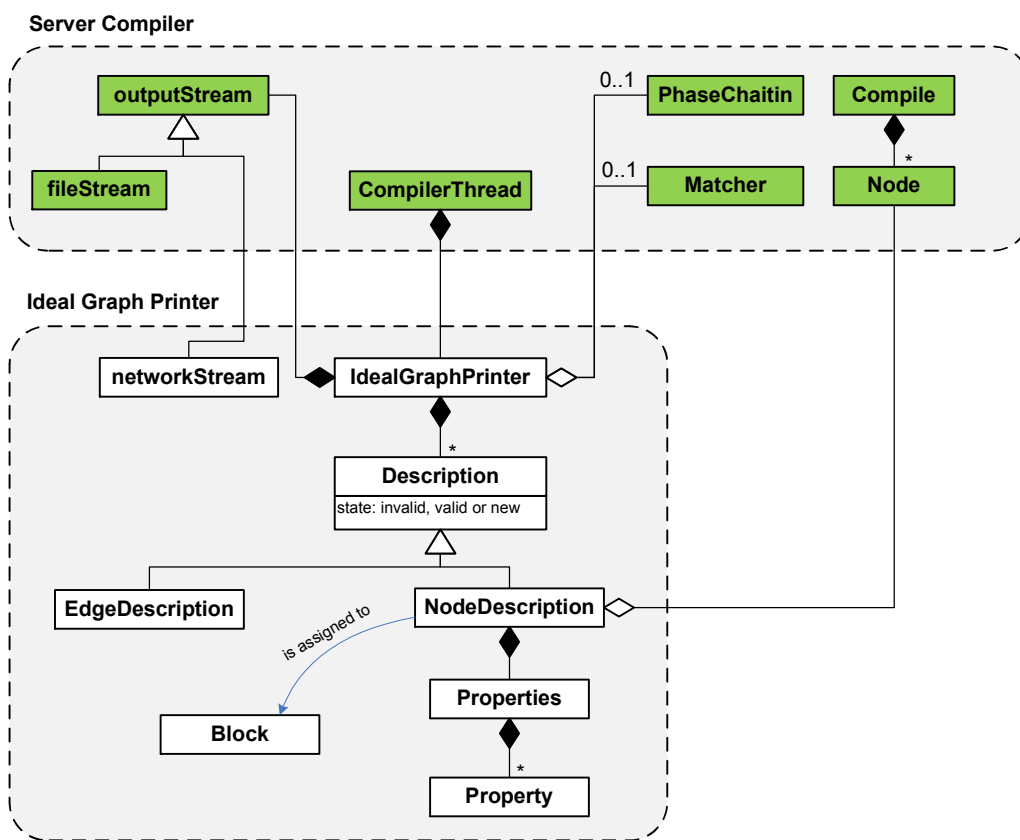


Figure 7.1: Compiler instrumentation class diagram.

7.2 Identifying Blocks

The nodes of the ideal graph are ungrouped. The control flow is only expressed as control dependence edges. The printer schedules the nodes into blocks to allow the visualization application to group the nodes. This increases the overview of the graph and is essential for large methods.

In a first step, all nodes that are related to control flow, i.e. either have a control dependence input or output edge, are assigned to blocks. The control dependencies link the control flow related nodes of a block together in a linear list. This ensures that there are no branches in a block. The start node is either a node that joins several control flow dependencies, e.g. a `Region` node, or a node that represents a successor of a node that splits control flow, e.g. an `If` node. The end node of a block is either a node that is the predecessor of a node that joins control flow or a node that splits control flow.

The algorithm processes the graph in reversed order, starting at the `Root` node. Figure 7.2 shows the steps applied for each node that is popped from the stack. When the node was not visited before, it forms the end of a new basic block. The algorithm walks up the graph along the control dependence edges and adds the nodes to the same block. The end of the walk is reached when the algorithm either finds a node marked as block projection, i.e. a successor node of a node that splits control flow, or a node marked as `block_start`, e.g. a `Region` node. This last node is the start node of the block and its predecessors are pushed onto the stack. Listing 7.1 presents the pseudo-code for identifying basic blocks.

Listing 7.1 Algorithm for identifying blocks

```

FindBlocks: graph g
  push root node of g onto stack

  while stack not empty do
    node n = pop from stack
    if n not visited then
      mark n as visited
      create new block b
      add n to b

      while n not block_proj and n not block_start do
        n = control dependence predecessor of p
        add n to b
      end while

      for each node p in predecessors of n do
        push p onto stack
      end for
    end if
  end for

```

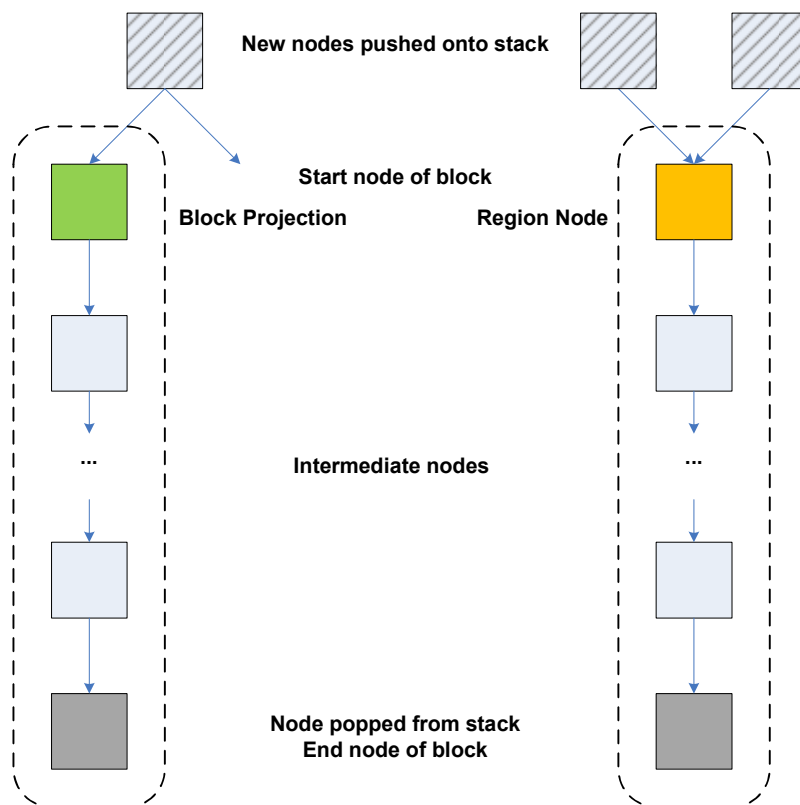


Figure 7.2: Steps applied for identifying a block.

7.3 Building Dominator Tree

A block is the dominator of another block if it is always executed before the other block is executed. Knowing the dominators is essential for scheduling the nodes, because the arguments of an operation must be executed in all control paths. Therefore, the operands of an operation must be either in the same block or in one of the dominators of the block of the operation itself. A block *a* is the *immediate dominator* of another block *b*, if all other dominators of block *b* dominate block *a*. The dominator tree consists of the blocks as nodes and edges between a block and its immediate dominator. The immediate dominator of a block is the parent of the block in the dominator tree. The start block of a method dominates all other blocks. Therefore, it is the root of the dominator tree.

There is an iterative $O(n^2)$ solution for calculating the dominators of all nodes of a graph. It starts by initializing the set of dominators of each node to the full set of all nodes. Then the algorithm reduces the dominator set of the root node to only the root node itself. The algorithm uses the following rule to iteratively update the dominator sets of the other nodes: The dominator set of a node is the conjunction of the dominator sets of its predecessors. If a node dominates all predecessors of another node, it must also dominate that node too. Additionally,

the node itself is always kept in its the dominator set. The algorithm iteratively updates the dominator sets until a full iteration over all nodes brings no further changes.

Lengauer and Tarjan [19] present a faster algorithm for calculating the immediate dominators, which runs in $O(n \cdot \log(n))$ time. First, the algorithm numbers the nodes of the graph based on a depth-first search traversal and stores the parents of a node in the depth-first search tree. The depth first tree has some special properties that are useful when calculating the dominators. A parent node in the depth-first tree has a lower number than all its children. As the dominator of a node is always reached on any path from the root to the node, it must also be reached by the depth-first search algorithm before the node itself is reached. There cannot be another way to reach the node. Therefore, the dominator of a node is an ancestor of the node in the depth-first search tree.

The second step of the algorithm is to calculate *semi-dominators*, i.e. approximate dominators that are later used to calculate the immediate dominators. Dominator candidates for a node are all nodes that are ancestors of the node in the depth-first search tree. Another node cannot be the dominator of that node, because there would be another path from the root to that node using the ancestors of the depth-first tree. When a node has only one predecessor, this predecessor must be the dominator of that node. There cannot be a path that reaches that node without using the edge from its predecessor to the node.

When there are two possible paths from node a to node b, then the depth-first search numbers of the second path are all higher than the numbers of node b. The semi-dominator of a node is the node closest to the root that is the start of two possible paths to that node. Figure 7.3 shows the relations between a node, its semi-dominator and the root node. The nodes between the node and its semi-dominator in the depth-first search tree cannot be dominators of the node, because there exists an alternative path from the root to the node.

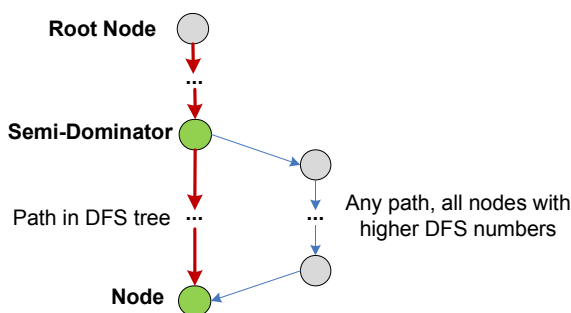


Figure 7.3: Calculation of the semi-dominators.

The semi-dominator is not necessarily the real immediate dominator of a node, because there might exist a path from the root node to one of the nodes between the node and the semi-dominator, which does not contain the semi-dominator. The dominator is always the semi-dominator, the root, or it lies between the semi-dominator and the root in the depth-first search tree.

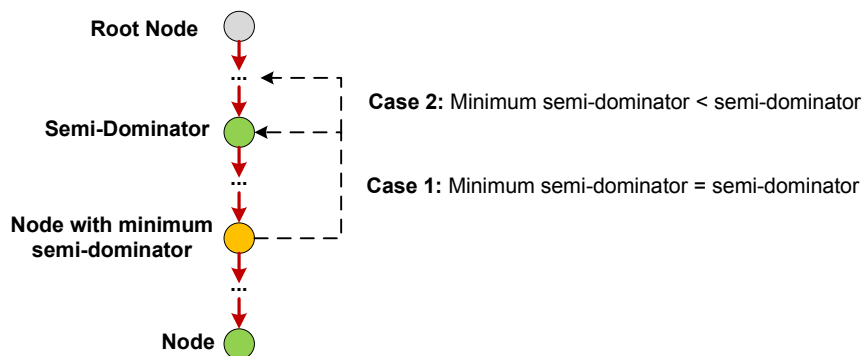


Figure 7.4: Calculation of the dominators.

Figure 7.4 shows the two cases that arise when calculating the immediate dominator of a node. First, the algorithm runs through the nodes between the node and its semi-dominator in the depth-first search tree. It selects the node that has the semi-dominator with the minimum depth-first search number. If this minimum semi-dominator is equal to the node’s semi-dominator, then the node’s dominator is equal to the node’s semi-dominator. Otherwise, the node’s dominator is equal to the dominator of the node with the minimum semi-dominator.

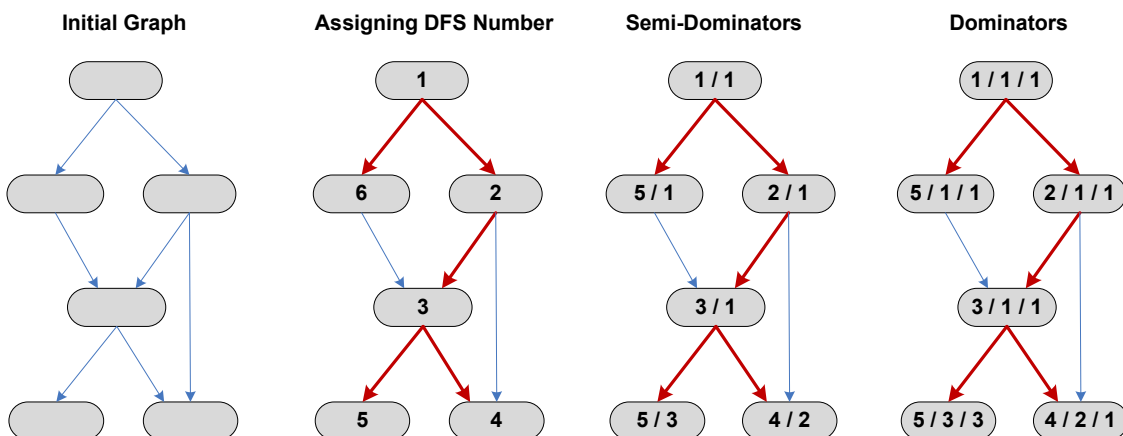


Figure 7.5: Algorithm for finding dominators applied on an example graph.

Figure 7.5 gives an example of a dominator calculation. First, the nodes are traversed in depth-first order and get a number assigned. The thick lines represent edges that are also present in the depth-first search tree. Next, the semi-dominators are calculated. For example, the semi-dominator of node 3 is the node 1 as there exists a path (1 - 5 - 3) that visits only nodes with a higher depth-first search number. The last step is the calculation of the immediate dominators based on the semi-dominators. Node 4 represents a case in which the semi-dominator and the dominator differs. This is because the node 3 is between the node 4 and its semi-dominator, and the semi-dominator of node 3 is the node 1, which has a lower number than the semi-dominator

of node 4. There exists an alternative path from the root to node 3 (1 - 5 - 3) that does not use the semi-dominator node 2. Therefore, the dominator of node 4 is equal to the dominator of node 3, which is the root node 1.

7.4 Scheduling

When scheduling nodes, the dominator tree plays a main role. A node has several input nodes that form parameters and the result of a node is used by other nodes. Before a node itself is scheduled, all input nodes need to be evaluated. On the other hand, a node must be scheduled before its result is used for the first time. Therefore, it must be scheduled in a block that dominates all the blocks in which the node is used.

The graph printer uses a scheduling strategy that puts a node in the latest possible block, i.e. in the block that dominates all blocks in which the result of the node is used. First, the dominator tree is constructed from the dominator information calculated by the algorithm of Lengauer and Tarjan (see Section 7.3).

Figure 7.6 explains how the common dominator of two blocks is calculated. First, the dominator tree is constructed. In this tree, each block is the child of its immediate dominator. Finding the common dominator of two blocks corresponds to finding the first ancestor of two blocks in the dominator tree. For example, the common dominator of F and E is the block B. The algorithm goes from the block F upwards to the root and marks each visited block with a flag. Then, it goes from the block E up to the root and the first block which has the flag set is the common dominator of the two blocks.

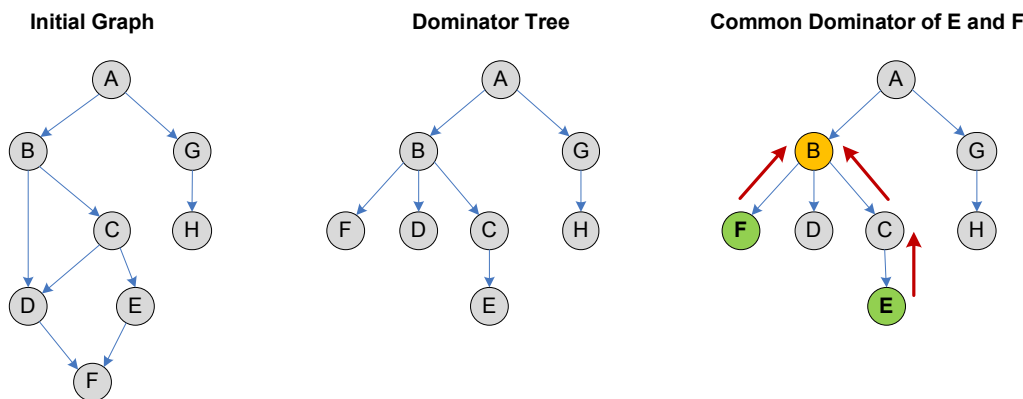


Figure 7.6: Finding common dominators.

The control flow related nodes are already scheduled (see Section 7.2). Now, the other nodes are scheduled iteratively: Whenever all successors of a node have been assigned to certain blocks, the node is put into the common dominator of the blocks of its successors. The common dominator of n blocks is equal to the common dominator of the last block and the common dominator of the first n-1 blocks.

7.5 Adding States

Additional states of the graph can be traced by calling the `print` method of the printer object of the current thread. A call to the static method `IdealGraphPrinter::printer()` returns the current printer object or `NULL` if printing is disabled. Figure 7.7 explains the sequence, in which calls to the public methods of a graph printer object are valid. The compiler threads automatically call the constructor and destructor of their printer objects at startup and shutdown of the VM. For every method, the `new_method` function must be called before any graphs are printed, and the `end_method` function must be called afterwards. In between, any number of calls to the `print` method are allowed.

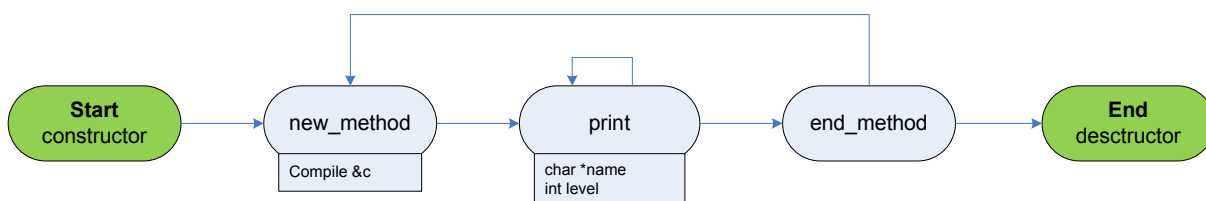


Figure 7.7: Life cycle of an `IdealGraphPrinter` object.

When calling the `print` method, the name and the level of the state must be specified. The call to `print` is ignored when the level set by the user is lower than the level of the state.

Listing 7.2 lists the code necessary to add a new state. The `IdealGraphPrinter` object is obtained by a call to a static function. This function automatically returns the graph printer object of the current compiler thread. It returns `NULL` when printing graphs is disabled. In this example, the level of the new state is set to 2.

Listing 7.2 Code for adding a state.

```

IdealGraphPrinter *printer = IdealGraphPrinter::printer();
if (printer != NULL) {
    printer->print("New State", 2);
}
    
```

Chapter 8

Conclusions

In this thesis, a tool for the visualization of the graph data structure of the Java HotSpot™ server compiler was presented. It assists at debugging the server compiler and at understanding optimization steps. The tool allows the analysis of large graphs and has built-in navigation and filtering possibilities. It clusters the nodes of the graph based on control flow and displays the graph using a hierarchical layout algorithm. Code in the server compiler allows the user to generate data for the visualization tool by using a compiler flag.

One of the most challenging problems during the development of the tool was to understand the graph of the server compiler. Its mixture of a control flow and a data dependence graph is less intuitive than the graph of the client compiler, which clearly separates these two concepts. The high number of edges of the graph in comparison to the node count is prejudicial to a clear drawing. Difficult to handle are nodes that resemble shared variables such as the node for the frame pointer and the return address. They are defined in one place, but used very often in the graph. The visualization tool uses a splitting filter for such nodes.

The first attempts to draw the graph ended up in a confusing scribble, even for small methods. A lot of time was necessary for incrementally increasing the drawing by filters and better layout algorithms. Each of the optimizations contributes a little to the clarity of the graph. Applying all of them finally gives an acceptable drawing. The performance was also a big issue, because the graph of a large Java method can have a high number of nodes and edges. For such a graph, not only efficient layout algorithms are necessary, but also a user interface that allows to focus on specific parts of the graph.

One of the planned improvements is the enhancement of the drawing performance of the user interface. Drawing performance is a critical issue, because the graph can contain a few thousand nodes with multiple slots and more than ten thousand edges with multiple interpolation points.

The Java HotSpot™ compiler group of Sun Microsystems is currently testing the tool on real-world examples. Based on these testing experiences, the tool will be further enhanced. Concerning the hierarchical layout algorithm, there are plans to integrate it into the official version of the NetBeans visual library.

List of Figures

1.1	Conventions used in the class diagrams.	2
2.1	Screenshot of Xelfi, the ancestor of NetBeans, running with JDK 1.1.	6
2.2	Using lookup, there is no dependency between service user and service provider.	8
2.3	Class diagram of the NetBeans visual library.	10
2.4	Screenshot of the visual library example program during execution.	10
3.1	Architecture of the Java HotSpot™ Virtual Machine.	13
3.2	States during the execution of an example method.	14
3.3	Architecture of the Java HotSpot™ server compiler of Sun Microsystems.	16
3.4	Program dependence graph when processing $p*100+1$	17
3.5	Graph when processing an empty method.	18
3.6	Graph when processing an <code>if</code> statement.	19
3.7	<code>SafePoint</code> node after parsing $5+x+7$	20
3.8	Identity optimization: $(x+0)$ is transformed to x	21
3.9	Constant folding: $(5+p+7)$ is transformed to $(p+12)$	22
3.10	Global value numbering: $(x+1) * (x+1)$ is transformed to $(x+1)^2$	23
3.11	Nodes that define a counted loop.	24
3.12	Matching and register allocation example.	25
4.1	Architectural overview.	28
4.2	Viewing a graph using the Java application.	30
4.3	The satellite view gives an overview of a graph.	32
4.4	Four functions applied to an example graph.	35
4.5	Bytecode Window showing the Java bytecodes in tree-form.	36
5.1	Dependencies of the NetBeans modules.	38
5.2	Lifecycle of the graph data.	40
5.3	XML file structure.	41
5.4	Data module class diagram.	42
5.5	Display model class diagram.	43
5.6	Layout model class diagram.	44
5.7	Properties and selectors class diagram.	46
5.8	Filters class diagram.	47
5.9	The difference algorithm applied on two example graphs.	49
6.1	Steps of a hierarchical layout algorithm.	52
6.2	Breaking cycles of an example graph.	54

6.3	Assigning layers for the nodes of the example graph.	56
6.4	Calculations when assigning y-coordinates.	58
6.5	Crossing reduction applied on a layer of the example graph.	59
6.6	X-coordinate assignment example.	62
6.7	X-coordinate assignment using the rubber band method.	64
6.8	Recursive cluster layout.	65
6.9	Example for the routing of a backedge.	66
7.1	Compiler instrumentation class diagram.	68
7.2	Steps applied for identifying a block.	70
7.3	Calculation of the semi-dominators.	71
7.4	Calculation of the dominators.	72
7.5	Algorithm for finding dominators applied on an example graph.	72
7.6	Finding common dominators.	73
7.7	Life cycle of an IdealGraphPrinter object.	74

Code Listings

2.1	An XML layer file defining an action and hiding a menu item.	7
2.2	Java source code of a visual library program with a label widget and an action. .	11
3.1	Architecture description file extract.	26
5.1	Algorithm for calculating the difference between two graphs.	49
6.1	Algorithm for breaking cycles in a directed acyclic graph.	53
6.2	Algorithm for assigning layers.	55
6.3	Algorithm for creating dummy nodes.	57
6.4	Algorithm for assigning y-coordinates.	58
6.5	Crossing reduction algorithm.	59
6.6	Algorithm for assigning x-coordinates.	61
6.7	Rubber-band algorithm for assigning x-coordinates.	63
7.1	Algorithm for identifying blocks	69
7.2	Code for adding a state.	74

Bibliography

- [1] *A Brief History of NetBeans*, URL: <http://www.netbeans.org/about/history.html>, 2007.
- [2] Tim Boudreau, Jaroslav Tulach, and Geertjan Wielenga, *Rich Client Programming: Plugging into the NetBeans Platform*, Prentice Hall, 2007.
- [3] Preston Briggs, Keith D. Cooper, and Linda Torczon, *Improvements to Graph Coloring Register Allocation*, In ACM Transactions on Programming Languages and Systems, ACM Press, 16, 428-455, 1994.
- [4] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson, *Value Numbering*, Software: Practice and Experience 27, 6, 701-724, 1997.
- [5] Gregory Chaitin, *Register Allocation and Spilling via Graph Coloring*, In Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, ACM Press, 98-105, 1982.
- [6] Cliff Click, *Global Code Motion/Global Value Numbering*, In Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, ACM Press, 246-257, 1995.
- [7] Cliff Click and Keith D. Cooper, *Combining Analyses, Combining Optimizations*, In ACM Transactions on Programming Languages and Systems, ACM Press, 17, 181-196, 1995.
- [8] Cliff Click and Michael Paleczny, *A Simple Graph-Based Intermediate Representation*, In ACM SIGPLAN Workshop on Intermediate Representations, ACM Press, 1995.
- [9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, In ACM Transactions on Programming Languages and Systems, ACM Press, 13, 451-490, 1991.
- [10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, *The Program Dependence Graph and its Use in Optimization*, In ACM Transactions on Programming Languages and Systems, ACM Press, 9, 319-349, 1987.
- [11] Emden R. Gansner, Stephen C. North, and Kiem-Phong Vo, *DAG - A Program that Draws Directed Graphs*, In Software, Practice and Experience 18, John Wiley & Sons, 11, 1047-1062, 1988.

- [12] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo, *A Technique for Drawing Directed Graphs*, In IEEE Transactions on Software Engineering 19, 214-230, 1993.
- [13] *Graph Visualization Software (GraphViz)*, URL: <http://www.graphviz.org>, 2007.
- [14] *Graph Layout Software aiSee*, URL: <http://www.aisee.com>, 2007.
- [15] Robert Griesemer and Srdjan Mitrovic, *A Compiler for the Java HotSpot Virtual Machine*, In The School of Niklaus Wirth, "The Art of Simplicity", dpunkt.verlag, 133-152, 2000.
- [16] *Java Platform Standard Edition 7 Source*, URL: <http://jdk7.dev.java.net>, 2007.
- [17] *Java HotSpot Client Compiler Visualizer*, <https://c1visualizer.dev.java.net>, 2007.
- [18] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox, *Design of the Java HotSpot Client Compiler for Java 6*, Technical Report, 2007.
- [19] Thomas Lengauer and Robert E. Tarjan, *A Fast Algorithm for Finding Dominators in a Flowgraph*, In ACM Transactions on Programming Languages and Systems, ACM Press, 121-141, 1979.
- [20] Stefan Loidl, *Compiler Data Flow Visualization*, Master's Thesis at the Johannes Kepler University Linz, 2007.
- [21] Adam Myatt, *Pro NetBeans IDE 5.5 Enterprise Edition*, Apress, 2007.
- [22] *NetBeans*, URL: <http://www.netbeans.org>, 2007.
- [23] Michael Paleczny, Christopher Vick, and Cliff Click, *The Java HotSpot Server Compiler*, In Proceedings of the Java Virtual Machine Research and Technology Symposium, 1-12, USENIX, 2001.
- [24] Eduardo Pelegrí-Llopart and Susan L. Graham, *Optimal Code Generation for Expression Trees: an Application BURS Theory*, In Proceedings of the 15th ACM Symposium on Principles of Programming Languages, ACM Press, 294-308, 1988.
- [25] Georg Sander, *Graph Layout Through the VCG Tool*, In Proceedings of the DIMACS International Workshop on Graph Drawing, Springer-Verlag, 194-205, 1994.
- [26] Georg Sander, *A Fast Heuristic for Hierarchical Manhattan Layout*, In Proceedings of the Symposium on Graph Drawing, Springer, 447-458, 1996.
- [27] Georg Sander, *Visualization of Compiler Graphs (VCG)*, URL: <http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html>, 2007.
- [28] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha, *The Java(TM) Language Specification (Third Edition)*, Prentice Hall, 2005.

- [29] Tim Lindholm and Frank Yellin, *The Java(TM) Virtual Machine Specification (Second Edition)*, Prentice Hall, 2007.
- [30] *uDraw(Graph)*, URL: <http://www.informatik.uni-bremen.de/uDrawGraph>, 2007.
- [31] Christian Wimmer and Hanspeter Mössenböck, *Optimized Interval Splitting in a Linear Scan Register Allocator*, In Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments, ACM Press, 132-141, 2005.
- [32] Thomas Würthinger, *Visualization of Java Control Flow Graphs*, Bachelor's Thesis at the Johannes Kepler University Linz, 2006.
- [33] *Xelfi*, URL: <http://tecfa.unige.ch/pub/software/win95/langages/java/xelfi>, 2007.