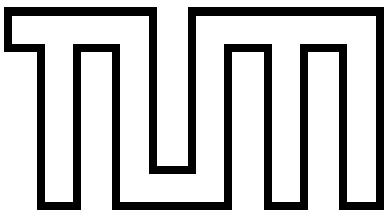
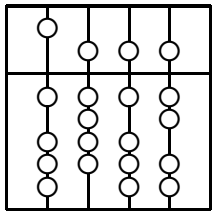




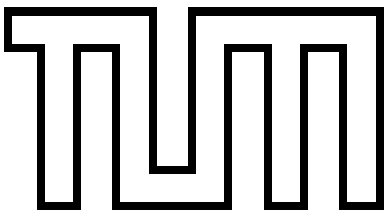
Technische Universität München

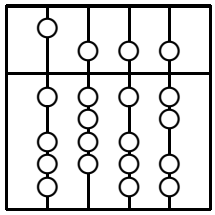
Institut für Informatik

Diplomarbeit

Generating Graphical Editors for Graph-like Data Structures

Gerwin Klein





Technische Universität München

Institut für Informatik

Diplomarbeit

Generating Graphical Editors for Graph-like Data Structures

Aufgabensteller:	**Prof. Dr. Dr. hc. Jürgen Eickel**
Betreuer:	**Alfons Brandl**
Bearbeiter:	**Gerwin Klein**
Abgabetermin:	**15. November 1999**

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 15. Oktober 1999 .....

Abstract

Generating Graphical Editors for Graph-like Data Structures

Many modern applications favor one or more graphical diagram editors, of which – despite their different look & feel and problem domains – on a closer look many have very similar functionality. Examples are UML class diagrams, finite automata, electronic circuit diagrams, or network charts. Their similarity is, that in an abstract way they all work on, edit and display a graph.

These editors and diagram viewers are mostly implemented by hand. Although this work is not very often done from scratch but rather with the help of some object oriented framework, it remains a complex task with usually high development costs.

This diploma thesis focuses on how the development and maintenance cost for this kind of graphical editors can be reduced. The problem is addressed in three steps:

We analyze example applications, extracting and structuring classes of requirements for graph editors to examine and clearly define the problem domain.

We develop a description language that allows to define and fully specify a graph editor in an abstract way. These specifications are thought to support the analysis and design phase in the development process. Compared to a full natural language description or a complete implementation, they are compact, easy to read, easy to communicate and easy to change.

We take a generative approach to automatically reach a full implementation of the graph editor defined by such specifications and discuss important aspects like the integration of generated code into existing applications and full customizability of generated editors in more detail.

Contents

1 Introduction **1**
1.1 Graph Editors . . . 1
1.2 Why a Generator? . . . 3
1.3 Related Work . . . 4
1.4 Overview . . . 7

I Requirements **9**

2 Requirements for generated Graph Editors **11**
2.1 Example Applications . . . 11
2.1.1 Rational Rose . . . 11
2.1.2 AutoFocus . . . 16
2.1.3 Flowchart Editor . . . 22
2.2 The Problem Domain . . . 24
2.3 Use Cases . . . 27
2.3.1 Modifying the Data Structure . . . 27
2.3.2 Changing Diagram Appearance . . . 30
2.3.3 Editing Attributes . . . 31
2.3.4 External Operations . . . 33
2.4 Functional Requirements . . . 33
2.5 Non Functional Requirements . . . 36

3 Requirements for the Graph Editor Generator **37**
3.1 Specification Language . . . 37
3.2 Functional Requirements . . . 39
3.3 Non Functional Requirements . . . 39

II Design **41**

4 The Specification Language **43**
4.1 Specifying the Problem Domain . . . 43
4.2 Specifying the Dialog Control . . . 45
4.2.1 Methods of Specifying the Dialog Control . . . 45
4.2.2 Dialog Control Automata . . . 45
4.2.3 Applying State Transition Systems to Graph Editors . . . 47
4.2.4 Extending Dialog Control Automata . . . 49
4.2.5 Providing multiple Layers of Abstraction . . . 51
4.2.6 Syntax . . . 55
4.3 Specifying the Presentation . . . 56
4.3.1 Presentation objects . . . 56
4.3.2 Connecting presentation and dialog control . . . 57
4.3.3 Syntax . . . 59
4.4 Styles . . . 60
4.5 Conclusion . . . 62

5 A Software Architecture for Graph Editors **65**
5.1 Overview . . . 65
5.2 Application . . . 66
5.3 Dialog Control . . . 69
5.4 Presentation . . . 71
5.5 Layout Algorithm interface . . . 74
5.6 Integration . . . 75
5.7 Persistence . . . 77
5.8 Conclusion . . . 78

III Implementation **81**

6 The Generator **83**
6.1 Generating Dialog Control . . . 83
6.1.1 Generating Abstract Dialog Control Automata . . . 83
6.1.2 Generating Interactors . . . 87
6.2 Generating Presentation and Styles . . . 90
6.3 Conclusion . . . 91

7 A Prototype **93**
7.1 Parallels to Compiler Construction . . . 93
7.2 Implemented Features and further research . . . 94

8 Conclusion **97**

IV Appendix **99**

A Grace **101**
A.1 Specification Language Overview . . . 101
A.1.1 Syntax . . . 101
A.1.2 Semantics . . . 103
A.2 Standard Library Reference . . . 109
A.2.1 Automata . . . 109
A.2.2 Interactors . . . 109
A.2.3 Presentations . . . 111
A.2.4 Figures . . . 112
A.2.5 Connections . . . 113

B Generating Editors with Grace **115**
B.1 Installing Grace . . . 115
B.1.1 System Requirements . . . 115
B.1.2 Getting Grace . . . 115
B.1.3 Installation . . . 115
B.1.4 Running Grace . . . 116
B.2 Specifying a Graph Editor . . . 117
B.2.1 Describing the Problem Domain . . . 117
B.2.2 Choosing a Style . . . 118
B.2.3 Customizing Interaction . . . 122
B.2.4 Customizing Presentation . . . 124
B.2.5 Connecting Graph Editor and Application . . . 127
B.2.6 Persistence with Grace . . . 128

List of Figures

1.1 A typical graph editor 2

2.1 Screenshot of a typical Rational Rose class editor session 13
2.2 Manually rerouting a class relation 14
2.3 Editing class attributes 15
2.4 UML class diagram of the AutoFocus problem domain 17
2.5 Screenshot of an SSD editor 18
2.6 UML class diagram of the SSD problem domain 19
2.7 Component ports in an SSD 19
2.8 A typical AutoFocus editing session 20
2.9 Screenshot of an STD editor 21
2.10 Screenshot of the flow chart editor 22

3.1 The process of generating a graph editor 39

5.1 A software architecture for graph editors 65
5.2 The model 67
5.3 Direct problem domain integration 68
5.4 Example of direct problem domain integration 68
5.5 Application integration with the adapter design pattern 69
5.6 Abstract dialog control via delegation 70
5.7 The renderer concept 72
5.8 Figure relationships 73
5.9 Figure class and interface hierarchy 74
5.10 Graph event listeners 76

7.1 Phases of graph editor generation 93

B.1 The first generated prototype editor 120
B.2 A customized style 121
B.3 Customized interaction 128
B.4 More integration 130

1 Introduction

1.1 Graph Editors

This thesis is about graph editors and how to generate them. Before we can elaborate on the goals we want to achieve, we should first clarify the term *graph editor* as it can be understood in more than one sense.

A *graph* in this context is meant to be an abstract structure consisting of nodes and edges as used in graph theory. It is not a function graph mapping parameters to values in a graphical way or a chart presenting business figures or statistical data. A *graph editor* is then a program that allows to view and edit such graphs. The editors in this thesis will be *graphical* editors with direct manipulation [37, p. 185–233] in the sense that they allow the user to work on a graphical representation of the graph rather than editing a textual representation. A typical graph editor is shown in figure 1.1. It is part of the application discussed in section 2.1.1.

Graphs and graph theory have a very broad range of applications. These include communication networks, flow-graphs, electrical networks, petri nets, state transition systems, traffic networks and class diagrams. Graphs are not only employed in mathematics and computer science, but also in fields like linguistics, architecture, chemistry, social sciences and geography[1].

Because of this broad range of applications, it comes to no surprise that many commercial computer applications use graphs and also graphical editors for them at some point. As we shall see in chapter 2, these editors are usually very tightly integrated into a larger application and it is usually not obvious for the user that she is editing a graph. The user is – and should be – thinking in terms of the application specific problem domain instead. She thus e.g. edits states and transitions of a state transition system in contrast to vertices and edges of a graph. These applications have – depending on their problem domain and purpose – not only different visual presentations of the internal graph, but also very diverse presentation and interaction *styles*. A presentation of a state transition diagram will certainly look different to a network communication plan and applications from such very different domains might very well have completely different ways the

[1] see a book like [47] for a more elaborate description of the applications of graph theory

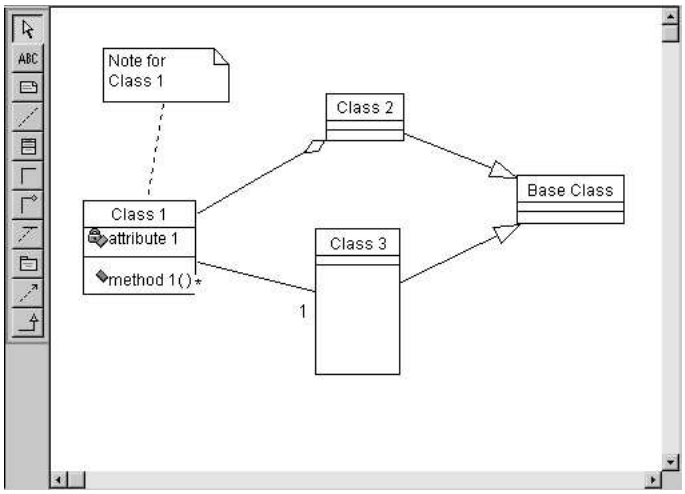


Figure 1.1: A typical graph editor

user interacts with the program. One application may allow a free drag & drop drawing interaction style, while another one provides only a small set of high level functions selectable by menu.

From these few points it already becomes apparent, that a graph editor can be a fairly complex component of an application user interface. If coded by hand, the development cost for these editors becomes very high[2]. Research has gone into how to reduce this development costs for more than 20 years now[3], but only frameworks have made it into industry and are used widely in commercial applications. Although they reduce the amount of work to build a graph editor by orders of magnitude, frameworks do still not provide a satisfying solution, because the task of building such widely used components as graph editors still remains on the level of programming languages and is still quite error prone. More problems of frameworks are discussed in section 1.3.

The goal of this thesis is to provide a specification technique and a generator that transforms these specification into program code for a graph editor such that

- generated editors can be fully integrated into existing applications
- interaction and presentation styles can be specified freely
- specifications are as concrete as necessary and as abstract as possible

[2]see for instance section 2.1.2
[3]see also section 1.3

Because one or more of the following points are often included in other graph editor research, it is necessary to state explicitly that it does not belong to the goals of this thesis

- to provide automatic layout algorithms; there will be an interface for third party layout algorithms, though.
- to include a specification technique for syntax directed editing. The editors should not be limited to one particular interaction style; if syntax directed editing is desired, an independent specification technique can be employed additionally.
- to provide automatic support for algorithms of graph theory. The algorithms used on the internal structure are considered to be a part of the application rather than the editor.

1.2 Why a Generator?

Section 1.1 has established, that there is need for support in the development of graph editors. We have also seen that full integration and a freely specifiable interaction style are vital for this support to be useful. We will now discuss why it is reasonable to use a generator for this purpose.

There are in general three ways to arrive at a piece of software:

1. code it by hand
2. use code from someone else, e.g. libraries and frameworks
3. produce a specification and generate the code automatically from this specification, e.g. compilers and generators

Usually a combination of these three ways is used in software development.

Since graph editors are widely used and complex components, it is not acceptable in terms of development costs to code them by hand each time anew. This leaves us with either frameworks or generators.

Most frameworks have the advantage, that many details can be directly controlled by the developer, and that most of them are designed to be integrated into a larger application. On the other side, this integration has to be done by hand and still adds to the development costs. A framework usually suggests a certain interaction and presentation style. The quality of the framework is to a very large degree determined by how easy it

is to tailor this style to the concrete demands of a certain application. Another problem is the level of detail the editor developer has to know about the framework. The more detail is accessible to the editor developer the better can the presentation and interaction styles be customized; but on the other hand, a very detailed and feature rich interface is hard to learn, hard to use correctly, and also hard to maintain. The tradeoff between a lean, usable interface and the support for as much customization detail as possible is made by the developer of the framework and not by the developer of the application and the editor. It is thus almost impossible for a framework to be appropriate for a very broad range of applications and still be easy to use at the same time. It is also hard to build up a library of interaction and presentation styles and to switch between them. In short: there is still a lot of code to write by hand.

This may sound as if it was a bad idea to use a framework to implement a graph editor; it is of course usually not. When a framework fitting the application is used and applied efficiently in the development process, the amount of work can be reduced significantly compared to coding the editor completely by hand. We claim however, that a generator with a suitable specification technique can reduce this development cost further for a broad range of applications. The specification language presented in this thesis allows the editor developer to determine herself what level of detail the specification contains. It also allows her to describe the interaction and presentation in a more abstract way than a programming language, and thus to focus on the more important aspects of the user interface design. This also should make the task of producing a graph editor significantly less error prone. The specification language provides important aspects like modularity and extensibility and thus makes it possible to provide libraries of specifications, e.g. containing different interaction and presentation styles, or just different parts of a specification developed by different people.

If it is better to use a generator than a framework, why then is it, that frameworks are widely used and generators not? Most research prototypes are not intended for industrial use, but are proofs of concept instead. As we will see in section 1.3, many show very interesting specification techniques in special directions, but none have yet all the properties we have identified as vital for a broad usage of these tools: full integration, freely specifiable interaction and presentation styles and multiple layers of abstractions.

1.3 Related Work

Since it is about the specification and generation of graph editors, this thesis is related to the field of model based user interface development as described in [41] or [10]. As opposed to tools like BOSS [38, 39, 40] and FUSE [26] we do not aim to generate a complete user interface for an application but rather focus on a small part of it. The

generated graph editors could be integrated into those more general model based user interface development tools and be used as complex, abstract interaction objects.

Being about a generator, this thesis is also related to compiler construction and to tools like lex & yacc [25] or the Cornell Synthesizer Generator [34]. Like these tools the generator applies techniques from the field of compiler construction, and it transforms a human readable specification into more efficient executable code.

More closely related research includes:

- **Frameworks**
  Frameworks are related to our topic, because they are the only widely used tools for developing graph editors. Popular ones include e.g. the commercial graph library by Tom Sawyer Inc. or the framework LEDA [30] which is more specialized on providing graph theory algorithms. Other ones are GraphViz [11], or the Graph Editing Framework GEF [35]. We already discussed the properties of frameworks in some detail.

- **Customizable editors**
  A customizable editor is somewhat similar to a framework, but more fixed in its interaction and presentation style. It usually is an already finished program, that allows customization of a predefined set of attributes. In general it has an a priori anticipated range of intended applications like e.g. the visualization of graphs and graph algorithms. A customizable editor is therefore not aimed at and usually not suitable for integration into other applications.

  Some of the tools in this category are: The Graph Visualization System daVinci [12], Graphlet [17], its predecessor Graph^{Ed} [16] and VGJ [28].

  Most of them provide a sophisticated presentation, often include automatic layout algorithms and depending on their respective intention further specialized features.

- **Generators**
  There is a wide range of generators, that are closely related to generating graph editors.

  GEGS (Generation of Editors for Graphical Structures) [42] claims to provide a specification technique that allows to fully specify all interaction and presentation aspects. These even include menus and other interaction elements normally handled by native widgets. [42] does not discuss integration. Every visible item on screen is internally considered a node, semantic aspects as “being connected” are modeled as edges. No running graphical prototype has been implemented, work seems to have been discontinued.

GeneSys [18] focuses on the transformation of graphically modeled user data into a textual representation. In this way it supports e.g. automatic code generation for graphical specifications. GeneSys generates a standalone graph editor application. Presentation and interaction styles are fixed, integration into larger applications other than by communication over a textual representation is not discussed.

The Expert Directed Graph Editor EDGE [33] is a very customizable editor with generator support for application integration. It supports multiple layout algorithms and graphical abstraction. The interaction style is fixed, the presentation style is to some point, but not fully customizable.

The following generators are aimed at the more general topic of generating diagram editors stemming from the field of visual programming languages. They have to address a broader range of problems that do not occur for graph editors.

Göttler uses attributed graph grammars [15] to specify syntactically correct structures. Interaction is specified by the application of grammar productions, presentation is specified in attributes. It is akin to text oriented syntax directed editors as produced by e.g. the Synthesizer Generator [34]. The grammar approach makes some basic transformation very hard [2]. Graphs as opposed to general visual languages usually don't have a complex syntactical structure, so specifications for usual graph editors would degenerate into a special case and not provide very much support for development. [15] does not discuss integration. The editors are limited to syntax directed interaction styles.

Arefi et al developed *a conceptual framework for a system that automatically generates object-oriented, syntax-directed editors for visual languages from their specification* [2, p. 351] The user chooses transformations on the internal abstract structure visualized by the editor. The transformations are here not limited to grammatical expansion and allow a more natural interaction specification. Integration is not discussed. The editors are limited to syntax directed interaction styles.

DiaGen as described in [43] is a *framework [...] using an internal hypergraph model and offering syntax-directed editing* [29, p. 230]. It has been extended [29] by an incremental hypergraph parser to allow for a drawing-tool-like free style of diagram editing. Integration is not discussed.

Chok and Marriott describe in [9] a technique that uses graph parsing to recognize correct syntactical structures and uses constraints to allow for an incremental layout. The recognition of correct syntactical structures is considered to be a part of the application in this thesis, and could be used in addition to the techniques presented here. While constraints are a very interesting technique for general diagram editing, most of the constraint techniques presented in [9] are not necessary

for the case of graph editors, because e.g. the connectedness of edges and nodes is an inherent property of graph editors and doesn't need to be specified explicitly. Constraints could very well be used for automatic layout of graphs, which is also considered to be a separate module in this thesis. Integration of generated editors into larger applications is not discussed.

1.4 Overview

This section gives a brief overview of the contents of this thesis.

We have finished the introductory chapter with a motivation for the presented work, the goals to be achieved and other research related to this topic. Chapter 2 concentrates on the requirements generated graph editors should meet, while chapter 3 is about the requirements for the specification language and the generator itself. Part II starts with a discussion of the design of the specification language in chapter 4 and continues with the design and the software architecture of the generated editors in chapter 5. Part III presents implementation aspects of the specification language in general in chapter 6, and of the generator prototype in more detail together with pointers to further research in chapter 7. Chapter 8 provides a summary and concludes this thesis.

The appendix in part IV contains an overview of the syntax and semantics of the specification language, the standard library reference, and a tutorial on how to generate graph editors with the generator prototype *grace*.

Part I

Requirements

2 Requirements for generated Graph Editors

This chapter will collect and structure requirements graph editors have to meet.

We will first take a look at three example applications in section 2.1, each using one or more graph editors. We will not try to review all these applications in full detail, but rather give a brief overview and then limit ourselves to the points most relevant for the graph editors they involve. In section 2.2 the problem domain relevant for the graph editor of these applications is examined and an abstraction fitting the purposes of all graph editors is provided. After having defined the problem domain, we discuss in section 2.3 the most common operations on this domain. Sections 2.4 and 2.5 provide a summary of the functional and non functional requirements for the class of graph editors we want to specify and generate.

2.1 Example Applications

2.1.1 Rational Rose

The first application we examine in some detail is *Rational Rose*, a commercial CASE tool by Rational Software Corporation. The application supports the graphical notation of the Unified Modeling Language [36]. It contains among other things editors for class diagrams, use case diagrams, and message sequence charts. It provides automatic code generation for different object oriented programming languages and reengineering of existing code.

As a commercial application and CASE tool it features a couple of sophisticated[1] graphical editors. Since they have a consistent look & feel, provide a multitude of features, and since we are mainly interested in the different ways graph editors are integrated,

[1] "sophisticated" is not meant to express any kind of superiority of the application to its competitors. We are not interested in a market analysis at this point, but rather in the fact, that with a commercial application the company will certainly pay attention to a professional looking user interface

used, and presented within different applications, we will limit ourselves to only one of these[2]: the class diagram editor.

Problem domain

The problem domain of this editor is the UML diagram type "class diagram" that is also used throughout this thesis.

The problem domain resembles a graph structure. Classes and notes describing them may be seen as the nodes, relations between classes and anchors of notes as the edges of a graph. Note, that there are different kinds of nodes – classes and notes – and also multiple different types of edges: directed ones, i.e. inheritance, aggregation, etc., and undirected ones between notes and classes. This structure does not yet describe the problem domain of the editor in full detail. The name and methods of classes as well as the arity of relations between classes for instance are not accounted for. We treat them as attributes of the nodes and edges. We also note, that there can exist more than just one edge of one type between every two nodes of the graph, and that the graph does not need to be connected.

The screenshot in figure 2.1 shows a typical configuration of the application at work:

1. The class diagram editor itself
2. The tool bar, context sensitive to the currently activated editor
3. The menubar of the application
4. The project browser
5. The information window

Presentation and interaction

The presentation style is that of a MS Windows[3] application. The application itself provides a typical menubar with a standard multiple document interface, short MDI. It contains informational windows like the toolbar, the project browser etc., and the actual specification editors.

The screen representation of the editor conforms to the UML definition of class diagrams. Color and size of the screen objects are adjustable by the user. Class relations are

[2] for more information on application details see `http://www.rational.com`

[3] Microsoft and Microsoft Windows are registered trademarks of Microsoft Corporation.

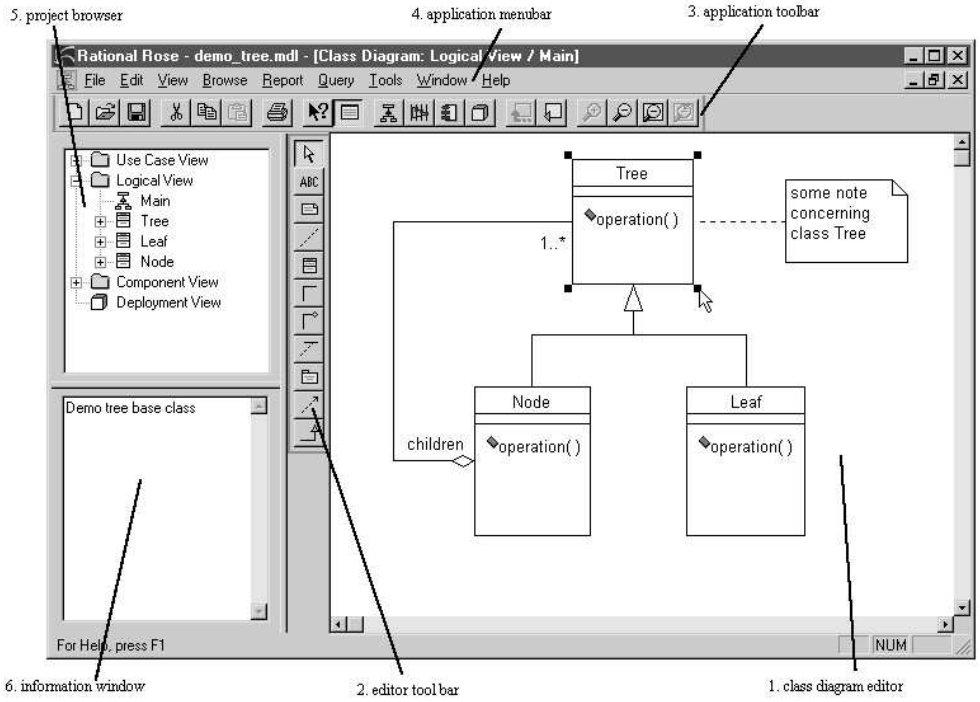


Figure 2.1: Screenshot of a typical Rational Rose class editor session

represented by straight lines by default; they can be broken up into multiple connected lines as illustrated in figure 2.2.

The class diagram editor is a large component and supports many ways of user interaction. We list a few of the most common ones:

- The user creates new classes and notes by choosing the desired symbol on the toolbar and clicking into the editor.
- She creates new relations between classes by dragging the mouse from one class to another.
- Edges can be rerouted arbitrarily by dragging handles with the mouse. Dragging a straight line portion of an edge breaks the line up into two new parts as in figure 2.2.
- Editor objects can be selected by mouse for further operations on them. Multiple selected objects are supported. The editor visualizes the selection state of objects

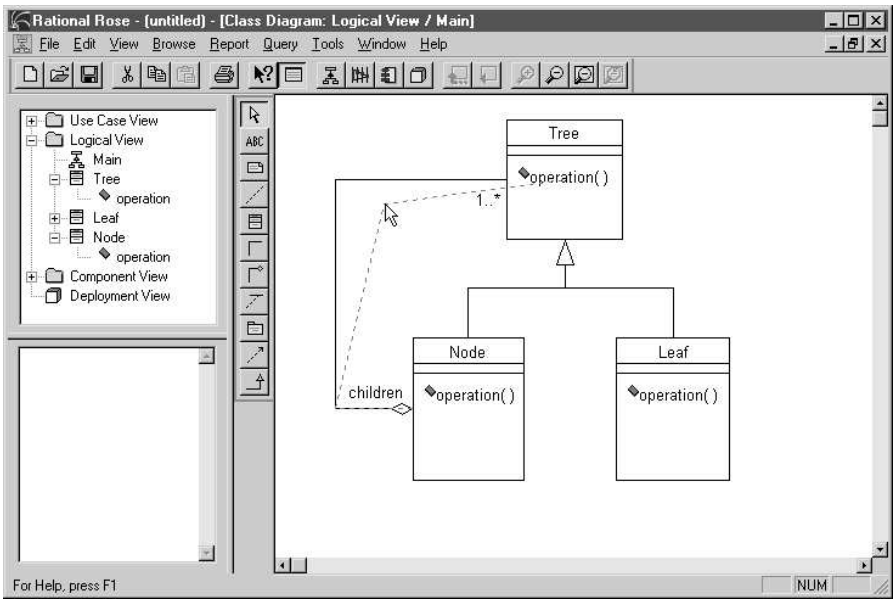


Figure 2.2: Manually rerouting a class relation

by drawing handles around them like around the `Tree` class in figure 2.3.

- A click with the right mouse button brings up a context menu, providing operations on currently selected objects. These operations can include common attribute modifications, deletion of the object from the diagram, a help dialog etc.
- Note texts, class names, and labels can be edited directly in the editor without an extra dialog box. A mouse click brings up a small in-place editor for the text.
- Node objects can be resized and moved by mouse operations if they are selected.
- The representations of classes and class relations stay connected when either object is moved on screen.
- The diagram supports arbitrary zooming and scrolling.

Integration

We will now take a look at how the class diagram editor is related to the application as a whole, and if and how it is integrated into the application.

The most prominent integration aspects are:

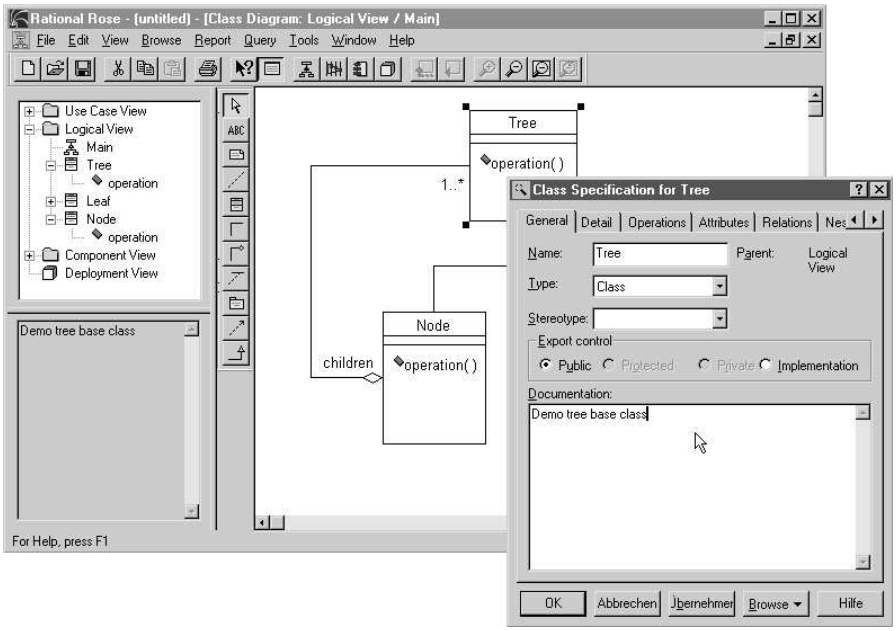


Figure 2.3: Editing class attributes

- The editor is just one of many document windows in the MDI user interface of Rational Rose.
- Menu items in the main menu of the application are context sensitive to the state of the class diagram editor.
- Help is context sensitive to the editor.
- Context sensitive pop-up menus provide a number of operations that are not directly related to editing the graph structure. They provide access to other application functionality.
- Selections and actions in the browser window affect directly the state and presentation of the diagram editor and vice versa.
- The information window is context sensitive to the editor state and current selection.
- The toolbar is context sensitive to which kind of editor is currently active and is also context sensitive to the state of that editor. It provides commonly used functions for the editor.

- The editor provides access to extensive attribute dialogs as for instance in figure 2.3. Many of these attributes do not affect the editor, but are application information only.
- Load/Save operations of the application refer to whole specifications, not only to a single diagram presented in one editor.

As we have seen, the application affects the state of the editor while the editor also directly and indirectly affects the state of the application. The operations on nodes and edges include more than just the graph aspects like "move a node", "create a node", "connect two nodes by an edge", but also extends to very application specific operations like "generate code for a class". A generated graph editor must provide communication and integration facilities to allow for these important aspects.

2.1.2 AutoFocus

AutoFocus[4] is a research prototype of a CASE tool for embedded systems [22]. It uses the FOCUS method and specification language [6] for graphical descriptions of distributed embedded systems.

It supports simultaneous distributed work by multiple developers, simulation of specifications [19], consistency checking of specifications [21], property proofs by modelchecking [4], and multiple specification languages.

A specification in AutoFocus consists of a set of diagrams. These diagrams include among others:

- System Structure Diagrams (SSD)

  They specify the statical structure of the embedded system by defining the set of components it consists of, and communication channels between them. Each component in an SSD may be decomposed and itself consist of other components described by another SSD.

- State Transition Diagrams (STD)

  An STD describes the behavior of a component. It consists of states and state transitions. States may again be decomposed into a set of states being defined in another STD. Transitions consist of input patterns, preconditions, post conditions, and output patters. Input patterns and preconditions specify when the transition is to be fired, output patterns and postconditions specify the output on communication channels and the new state of component data.

[4] see also `http://autofocus.informatik.tu-muenchen.de`

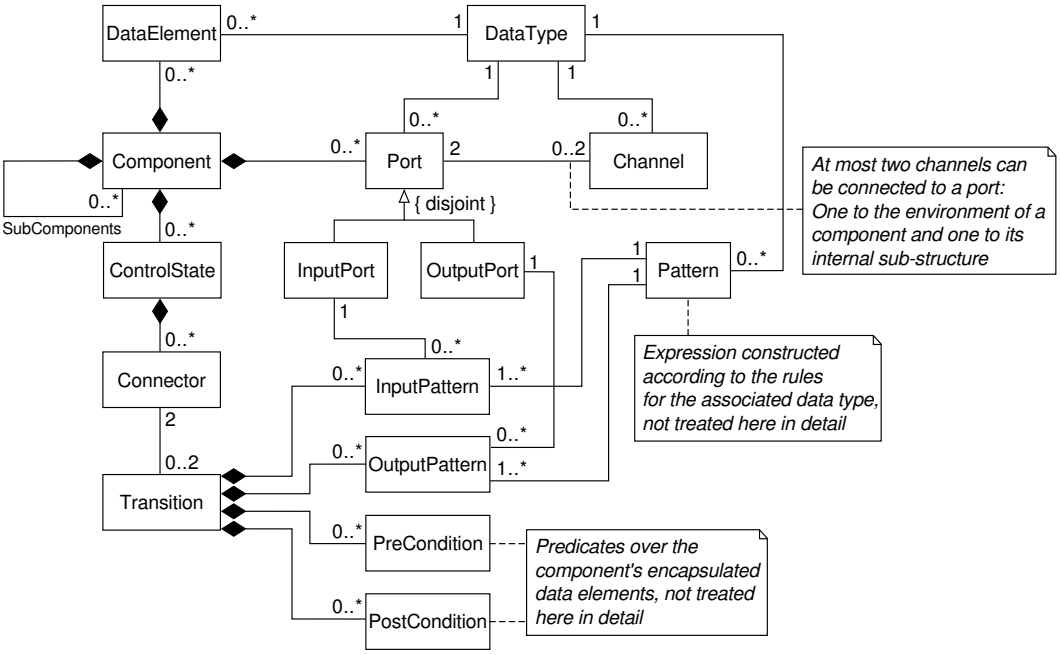


Figure 2.4: UML class diagram of the AutoFocus problem domain
(with friendly permission taken from [20])

- Extended Event Traces (EET)

  An EET similar to UML's Message Sequence Charts describes an example communication trace of the specified system. It consists of axes denoting components, and events denoting values on communication channels.

Figure 2.4 gives a simplified overview of the application's problem domain.

Since it is a tool for graphical specifications, AutoFocus similar to Rational Rose features multiple graphical editors.

Although AutoFocus is not a commercial application, a significant amount of work has been dedicated to develop usable graphical editors: they consist of ca. 30.000 lines of Java source code, one third of the source code of the whole application. Since there were no frameworks for graph editors in Java available at the time when the project started, almost all of this code is handwritten and developed from scratch.

We will discuss two of the editors in further detail because their respective problem domains resemble graph-like data structures.

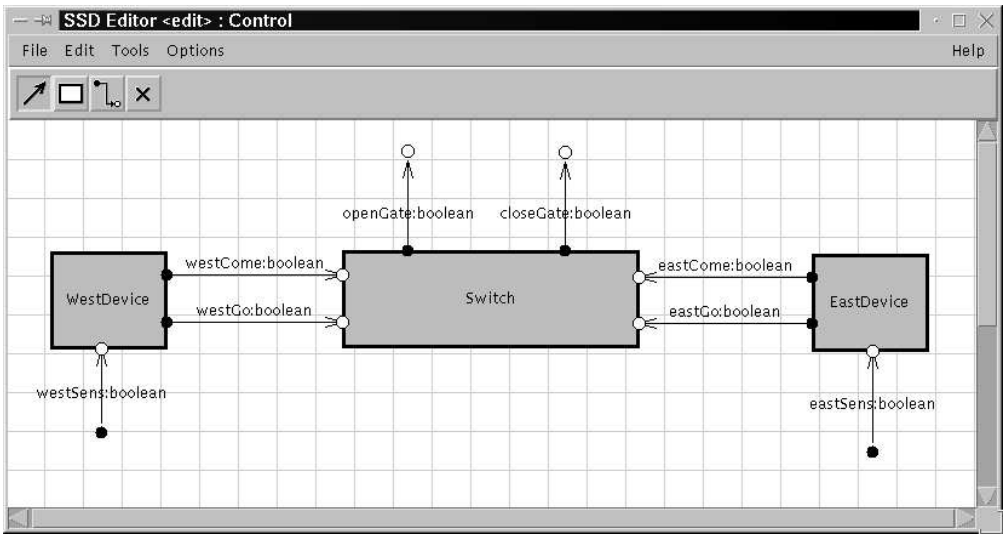


Figure 2.5: Screenshot of an SSD editor

System Structure Diagrams

Figure 2.5 shows a screenshot an SSD editor working on a specification for a rail gate control component. Although the problem domain of SSDs as in figure 2.6 is hierarchical and not at all a graph, the part of the SSD that actually is presented and modifiable in each one SSD editor is graph-like in the sense, that it can be viewed as just the set of components and communication channels displayed in that particular diagram. A hierarchical decomposition of an SSD can bee seen as an attribute of the component decomposed. Again names, types, etc. are attributes of nodes and edges.

Note, that channels lead from and to ports and not directly to components, which we have identified as the nodes in the graph editor. It is easily possible though to construct a mapping that yields the source and target component for every channel. This does not work with channels starting or ending in external diagram ports. They are not connected with any component. We could treat this kind of ports as a separate node class.

Presentation and interaction

The part of the SSD problem domain presented on screen includes components, external diagram ports and channels between components.

Components are represented as named solid boxes with component ports as dots. Component ports are painted empty or filled depending on their direction. If the mouse is

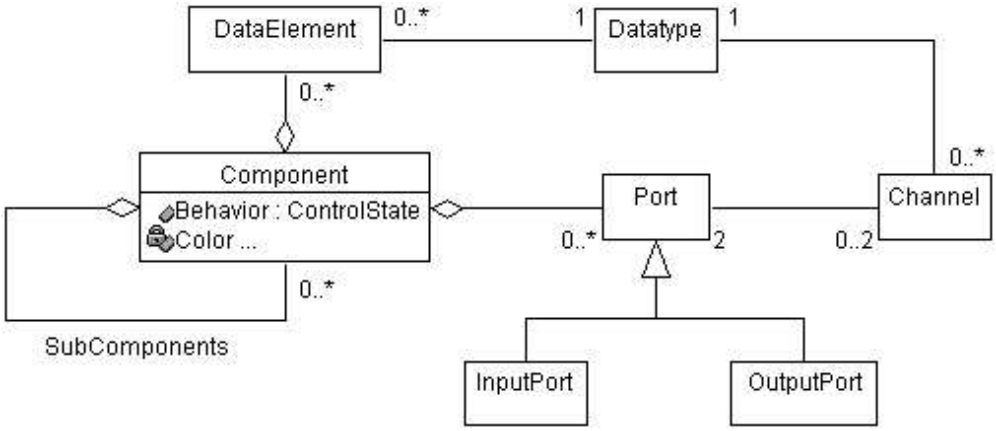


Figure 2.6: UML class diagram of the SSD problem domain

placed over a component port, the name and type of that port is displayed as in figure 2.7. An icon shows if there is a hierarchical decomposition available for a component. Colors can be adjusted by the user. Channels are labeled orthogonal lines with arrows showing the direction of the channel. The label consists of name and type of the channel. Channels are routed automatically.

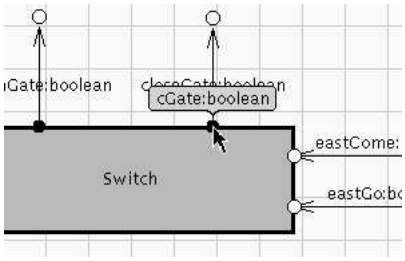


Figure 2.7: Component ports in an SSD

The user interacts with the editor solely by mouse operations. Objects are selected by clicking on them or by choosing an appropriate menu item. The state of an object being selected is presented by highlighting the border in a different color. When selected components are moved or resized, channels and internal ports are rerouted automatically. Channels and internal ports cannot be moved directly. External diagram ports can be moved independently but not resized. As usual, the editor also supports selection of multiple objects.

The editor has two different presentation styles of objects being moved: opaque and shadowed. In opaque mode, the user sees the the new diagram instantly, while in shadow mode, the new position of a component is indicated by an empty rectangle. The diagram is updated when the user has completed the move operation.

When moving or creating objects, the editor ensures geometric constraints on the diagram. It will for instance not allow diagrams with overlapping components.

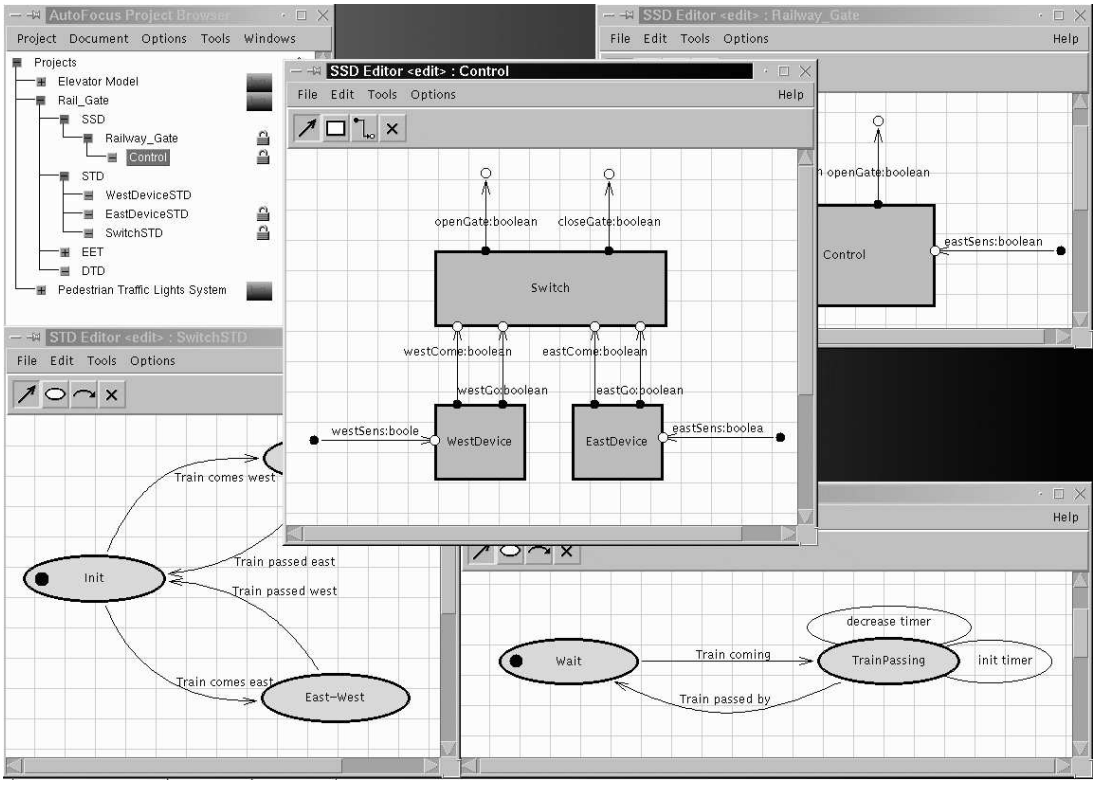


Figure 2.8: A typical AutoFocus editing session

New components are created by choosing the component icon on the toolbar or by selecting the appropriate menu item, followed by a mouse click on the desired location of the new object. Channels are created by choosing the channel icon and dragging the mouse from source to target. Source and target can both either be components, external ports or nothing. If the user drags the mouse to an empty spot in the diagram, a new external diagram port is created automatically at this location.

Integration

Like Rational Rose, AutoFocus supports multiple documents to be opened and edited at the same time in one session. In AutoFocus there is not one application window containing all documents, but each editor is an own toplevel window with own menubar and toolbar. A typical editing session might look like figure 2.8.

Although it is not as visually apparent as in section 2.1.1, these editors are still integrated into the context of the larger application:

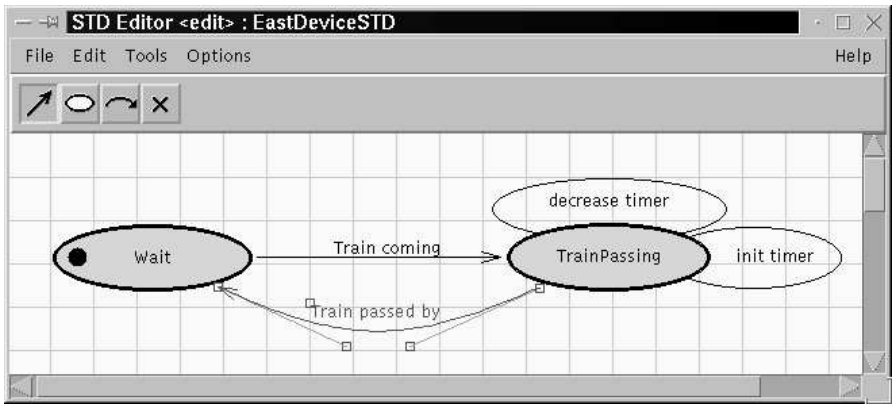


Figure 2.9: Screenshot of an STD editor

- For each component with a hierarchical decomposition, the SSD describing this decomposition can be opened from within the editor by mouse operation. If there is not yet any hierarchical decomposition, the user can create one directly from within the editor. The editor is thus not only concerned with the flat graph structure being edited, but also allows navigation in the larger application problem domain.
- The menubar is context sensitive to the selected object. It allows for instance to open an associated STD for a selected component. Other operations are: creating a new STD for a component or associating an existing STD with a component, open/edit data definition for components etc.
- The editor provides access to attributes of the presented objects. As for instance internal data declarations of components, most of these attributes are only relevant to the rest of the application, not to what is presented in the editor.
- The help system is context sensitive to the state of the editor

System Transition Diagrams

The look & feel of STD editors is consistent with the one of SSDs. We will therefore skip most of the discussion and concentrate on a few differences.

The problem domain of STDs are state transition systems. Like in SSDs, these systems can be hierarchical and are usually described by multiple documents, each of which can be created in an own STD editor. Figure 2.9 shows a typical editing session. Although the editor features the same presentation style as the SSD editor presented above, the

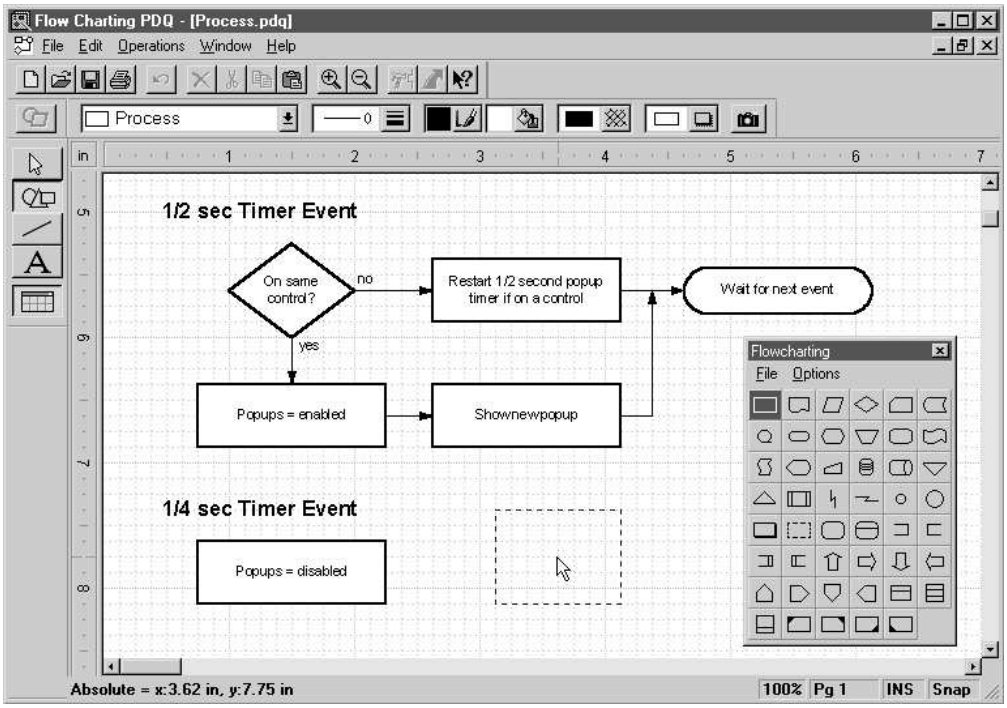


Figure 2.10: Screenshot of the flow chart editor

actual presentation is completely different. SSDs have rectangular components and automatically routed orthogonal channels, STDs have states represented by ovals and transitions between them represented by labeled straight lines or bezier curves. The interaction style is the same as with SSDs and similar integration aspects apply.

For our generator this means, that on the one hand it should be easy to provide different editors with the same presentation and interaction styles to remain consistent in one application or platform. On the other hand it must also be possible to specify exactly, what these styles should be like, to generate editors for multiple different applications and platforms.

2.1.3 Flowchart Editor

The last example application is a flowcharting program. For the purposes of this section, we chose the shareware program PDQ Lite illustrated in figure 2.10.

The main task of this application is to produce a wide range of diagrams like electric circuit diagrams, flowcharts and many of the others already mentioned in the introduction.

The main difference between this flowcharting program and the applications discussed above is, that this one is merely a drawing tool. It has no semantical knowledge of the diagrams or any underlying problem domain. The concept of nodes and edges between them is the only syntactical constraint. It is in essence a large and sophisticated graph editor that is not embedded in a larger application. Its purpose is to support the drawing of diagrams but not to offer any further processing of them.

The most prevalent interaction and presentation aspects are:

- It supports multiple templates, i.e. graphical kinds of nodes that can be used in diagrams. These templates are divided into classes for specific diagram kinds like for instance process diagrams, business models etc.
- The presentation of nodes can be customized in color, size, position, shading and shadow.
- Nodes contain an arbitrary text that can be edited directly from within the editor and customized in font, size, alignment position etc.
- Edges can be connected to nodes or disconnected from any node.
- Edges between nodes are routed automatically.
- It features the standard components of a multiple document interface, an application toolbar, and a menubar.
- The diagrams support arbitrary zooming and scrolling.
- Diagrams can be saved in multiple standard formats and thus be exported to and imported from other applications.

In the example there is no integration into a larger application problem domain. If however we view only the drawing canvas of the flowchart application as the actual graph editor, we could say, that the editor is also controlled by other parts of the application such as menus and toolbars. The state of menus and toolbar in turn depend on the state of the editor.

We can see, that even in this simple example, where there is no deeper logical structure involved, integration aspects are still important. On the one hand, the editor gets its controlling input not only directly from the user, but also from other application parts; on the other hand the editor must provide some sort of state that the application can use to modify its other interaction devices.

2.2 The Problem Domain

In the last three sections we used the concept of nodes and edges intuitively. It is the task of the following one to establish a common terminology and to clearly define the notion of graphs and problem domains suitable for graph editors.

The abstract structure

We begin with a look at the formal definition of the abstract problem domain structure called *graph*.

There is no universally agreed upon standard definition of what a graph is.[5] Almost all of the definitions have in common, that a graph $G = (V, E)$ is a pair of vertices V and edges E. Depending on their respective purpose, they disagree on whether E is a set or a multiset, if edges are ordered or unordered pairs of vertices, if loops are allowed or not et cetera.

For this thesis, we will use the following

Definition 2.1
A graph $G = (V, E, s, t)$ is a quadruple consisting of a set of vertices V, a set of edges E and two total mappings s and $t : E \to V$ with $s(e)$ yielding the source and $t(e)$ the target node of the edge $e \in E$.

It is not necessary that $(s(e), t(e))$ be distinct for any two edges e or that $s(e) \neq t(e)$ for any $e \in E$. So our graphs are directed, allow multiple edges between vertices, and may have loops. Since both s and t are total mappings, each edge must always be connected to two nodes.

For convenience, we also introduce the incoming and outgoing edges of a node.

Definition 2.2
Let $G = (V, E)$ be a graph. $out : V \to 2^E$ is the function that yields for each vertex $v_1 \in V$ the set of edges that lead out of v_1 and $in : V \to 2^E$ is the function that yields for each vertex $v_2 \in V$ the set of edges that lead out of v_2:

$$\forall v_1, v_2 \in V.\, \forall e \in E.\, s(e) = v_1 \wedge t(e) = v_2 \iff e \in out(v_1) \wedge e \in in(v_2)$$

It may seem to be an unnecessary complication to use mappings s and t instead of a multiset of ordered pairs. We use functions in this place to provide a slightly higher grade of abstraction and to not suggest any preferred style of implementation of the

[5] see also an introductory work on graph theory like [46]

graph structure. The idea is, that an edge in an existing application does not need to be a pair of vertices with some attributes, but will usually be represented by some kind of object instead. We only demand, that this object provides a method yielding two vertices. The editor then treats them as source and target nodes of the edge.

Having established the definition of graph used in this thesis, we spare the reader all further definitions of the usual concepts like loop, degree, connectedness, planarity etc. They are analogous to those presented in [46].

We proceed with the also not really unusual yet necessary definition of attributes instead:

Definition 2.3
An attributed graph $G = (V, E, s, t, A)$ is a normal graph as defined above with a set A of attributes and two functions $attribute : V \to A$ and $attribute : E \to A$ yielding the attributes of vertices and edges. An attribute is a triple of name, type and value completely analogous to the concept of object members in most object-oriented programming languages.

V and E are both partitioned into multiple classes. This is convenience rather then necessity. It is convenient, because many applications use the notion of different kinds of nodes and edges that each share common properties, have the same or similar set of attributes and support the same operations.

We stated above, that our definition of graphs yields directed graphs that allow multiple edges between any two nodes. What if these properties are not desired by a certain application? We can apply the following techniques to simulate different constraints and properties while keeping the notion of the formal structure of definition 2.1:

- Undirected graphs are easily simulated by treating the mappings s and t as one mapping or by constructing a new mapping that constructs unordered pairs from the values of s and t.
- In our graphs all edges must at all times be connected to exactly two nodes. Some applications may want to apply a more liberal policy where an edge is connected to less than two nodes. This policy can be achieved by introducing a new node class with one node v_0. Each edge connected with this node can then be interpreted as having no node connected at the respective end.
- If the edges of an application structure may be connected to more than one node, i.e. when the application structure is not merely a graph, but a hypergraph, we can represent this again by introducing a new node class. A hyperedge e leading from v_0 to nodes $v_1, v_2 \dots v_m$ is then represented by a new node v_e with edges (v_0, v_e) and $(v_e, v_1), (v_e, v_2), \dots, (v_e, v_m)$.

- In many applications the graph structure is hierarchical in the sense that each node or edge may itself be a graph. We can represent this by introducing an attribute of e.g. nodes that contains the subgraph of that respective node.

- Some graph editors support the notion of ports, i.e. connection points between nodes and edges as a separate concept. In the cases where that is not a presentation issue only, the concept can again be simulated in the editor by introducing a new separate node and edge class. If there is an direction associated with a port, we represent a port between vertex v and edge e by a new node v_p and a new edge e_p between v and v_p. The original edge e is connected to the port, i.e. to the new node v_p, and the direction of the port is represented by the direction of the new edge e_p. Undirected port could have two edges between v and v_p – one in each direction.

Suitable problem domains

Now that we know what exactly is meant by the term *graph*, we are able to examine which application problem domains are suitable for our purposes of presenting them in a graph editor.

An application problem domain is suitable for a graph editor, if there is a mapping between the relevant part of the problem domain to a graph structure. Relevant are those parts of the problem domain that are to be presented to the user in the graph editor. So if a part of the problem domain is easily understood by the user as a bunch of objects – the nodes – and connections between them – the edges – then that part is suitable for a graph editor.

The user does of course not have to be aware of the fact the editor internally is working on a graph. She should on the contrary be able to think in the application specific concrete problem domain instead. The application developer should also not need to be aware of the fact, that her application problem domain is viewed as an abstract graph structure in the graph editor. It is the responsibility of the editor developer to provide the mapping between the problem domain and the abstract graph.

State transition systems provide an example for a suitable problem domain. States map to nodes and transitions to edges. An intended presentation may consist of ovals as nodes and straight line connections as transitions.

For a problem domain like UML's Message Sequence Charts one may still find a relatively intuitive mapping to a graph domain, but the intended presentation is not very graph-like. If one maps component axes to nodes and communication events to edges, the presentation of nodes is at least unfamiliar for a graph structure. The presentation

of edges, i.e. their physical screen location above or below other edges of the same component, carries important temporal information beyond the concept of a graph and does not only visualize a connection between two components. This example will certainly be harder to implement with the graph editor generator and require special, highly customized presentation and interaction specifications.

A problem domain not suitable for our kind of graph editors is for instance the Nassi-Shneiderman diagram [31]. There is no intuitive mapping from the problem domain to graphs, except if one resorts to extreme cases like treating everything as a node. The intended presentation is also not graph-like and depends heavily on geometric properties of the layout to express logical relationships other than mere connectedness of objects.

2.3 Use Cases

We have so far explored some example applications, have a clear idea of the problem domains we are working on, and can now divide interactions found in 2.1 into classes.

In a usual software project, one would at this point want to employ UML use cases for a general description of user tasks. Unfortunately, we cannot do so here, because we do not have one application but a whole class of applications instead. The thing these applications have in common is the graph editor, but they may differ extensively on the operations they provide to the user and also on the way how they provide them. We will therefore not develop actual uses cases, but work with a means similar to use cases to describe our interaction classes. There will not be a concrete, fixed basic course of action with some exceptions to them, but instead a class of similar actions with different properties or slightly different courses of action. In the following, we orient ourselves on a tabular form of use cases, describing a common basic course of action, and after that, discussing alternative courses, properties and examples of them, and some exceptions that should be expected to occur. Although these are not quite use cases in the strict sense of the word, we will still call them so, because they serve the same purpose with very similar means: They describe the system from a user point of view. In our case, the system is the graph editor, and the user may be the application or the person using the editor.

2.3.1 Modifying the Data Structure

The following two use cases represent the main user tasks involved in the graph editor. They both concern modifications of the problem domain data structure. These two use cases occur in almost all applications containing a graph editor.

Some applications provide "view only" graph editors though, that may or may not support changing diagram appearance, but that do not allow any structural modification.

Use Case "Object Creation"

Description: a new graph node or edge is created

Actors: user, editor, application

Triggers: the creation of a new object can be triggered by the user or the application

Result: a modified internal graph and screen representation with the new object

Basic course:

The kind of the new object is determined, i.e. if node or edge, and which node or edge class.

The topological position of the new object and initial attribute settings are determined.

The editor presents the new graph to the user.

Alternative courses, exceptions:

Depending on editor specification and application, each action above can be triggered by either the user or the application.

An explicit user choice for the kind of object being created would manifest itself for instance in pressing a toolbar button for a particular node or edge class. An implicit application choice could consist of some application default or a computed value that is applied automatically. In this case the user may only trigger the creation of the new object or the application may act completely on its own.

Another step involves the topological position of the new object. In case of a new node this is trivial, in case of a new edge, the source and target nodes have to be determined. Depending on the concrete editor and application, the user identifies these nodes for instance by clicking on them in a particular order or by dragging the mouse from source to target. At this point a check for syntactical correctness could take place to either allow or reject the new object. A syntax directed editing style would offer the user only correct choices in the first place.

The creation of a new node with a default name attribute at the position of the user mouse click is an example for both user and application triggered initial attribute settings. At this point a semantical check could either allow or reject the chosen attribute values.

The last step of presenting the new graph on screen may involve a new automatic layout, or may trigger application functions.

The order of the steps above is not mandatory and can vary from editor to editor, from application to application, or even within a single editing session, depending on the editor specification. They have in common, that the presentation of the new graph happens last, while the kind of the new object is determined in the first step.

On our graph problem domain, this constitutes the functions $addNode : \mathcal{G} \times V \to \mathcal{G}$ and $addEdge : \mathcal{G} \times \mathcal{E} \times V \times V \to \mathcal{G}$ where $\mathcal{G}$ is the set of all graphs and $\mathcal{E}$ the set of all edges in $\mathcal{G}$.

The function $addNode$ simply includes the new node in the graph:

$$addNode((V, E, s, t), v_0) = (V \cup \{v_0\}, E, s, t)$$

$addEdge$ changes the set of edges and also mappings s and t:

$$addEdge((V, E, s, t), e_0, v_s, v_t) = (V, E \cup \{e_o\}, s', t') \quad \text{with } v_s, v_t \in V \text{ and } e_0 \notin E$$

The new mappings s' and $t' : E \cup \{e_0\} \to V$ are simple extension of s and t

$s'(e) = s(e)$ for all $e \in E$ and $s'(e_0) = v_s$, and analogously
$t'(e) = t(e)$ for all $e \in E$ and $t'(e_0) = v_t$

If the graph editor is to support the task of creating new graph objects, the problem domain must at least provide operations yielding the effect of $addNode$ and $addEdge$ under the mapping to the graph structure.

Use Case "Delete Objects"

Description: a graph node or edge is removed from the graph

Actors: user, editor, application

Triggers: the removal of an object can be triggered by user or application

Result: a modified internal graph and screen representation with the object deleted

Basic course:

The object to be removed from the graph is determined.

The application removes the object from the internal structure.

The editor presents the new graph to the user.

Alternative courses, exceptions:

Again, depending on editor specification and application, the removal itself and the choice which object to be removed can be triggered by either the user or the specification.

An explicit user choice could be a mouse click on an edge to be removed; the application in another case could for example automatically choose to remove a node in a network diagram representing a computer that has gone offline.

Removing one object from the graph by user or application can result in the application removing more objects from the graph in response to reach a consistent state. Removal of a node in a network diagram by the user could for instance lead to an automatic removal of all communication links to that node. These actions depend on the application's notion of consistency of its problem domain.

Exceptions to the course above can occur when it is not legal in application terms to remove a specified node. Handling of these kind of exceptions depends again on application and editor.

Analogous to *add* above, this constitutes the functions $removeNode : \mathcal{G} \times V \to \mathcal{G}$ and $removeEdge : \mathcal{G} \times E \to \mathcal{G}$.

This time *removeEdge* is simpler:

$$removeEdge((V, E, s, t), e_0) = (V, E \backslash \{e_0\}, s, t)$$

removeNode removes a specific node, but must also remove all edges leading into and out of that node, because the mapping t is total:

$$removeNode((V, E, s, t), v_0) = (V \backslash \{v_0\}, E', s_{|E'}, t_{|E'}) \quad \text{where}$$

$$E' = \{e \in E \mid t(e) \neq v_0 \wedge s(e) \neq v_0\}$$

2.3.2 Changing Diagram Appearance

The next use case describes actions that affect the appearance of the diagram. They do not change the logical structure of the graph.

Use Case "Modifying Diagram Layout"

Description: a graph node or edge is moved or rerouted in the diagram

Actors: user, editor, application

Triggers: a new diagram layout can be triggered by user or application

Result: a new screen representation with unchanged logical structure

Basic course:

There are two main courses of action:

If the new layout is triggered by the application, the graph is laid out by an automatic layout algorithm.

If the change is triggered by the user, she first determines the objects to be moved or rerouted. The user then chooses a new position, size or other visual attribute of the object.

In each case, the editor at least finally presents the new layout to the user.

Alternative courses, exceptions:

The action of choosing one or more objects to reshape is often made explicit by the concept of selections as we have seen it e.g. in section 2.1.1. The concept is not mandatory though; moving a node for instance could happen in one simple gesture by dragging it directly with the mouse – without an explicit prior selection. The gesture does however include determining which node to move and where to move it. The difference is, that "selecting objects" is a user concept while the other one can be applied implicitly by editor and application.

When the presentation of a diagram object is changed, it is the responsibility of the editor to modify the presentation of other affected objects accordingly. If e.g. a node is moved, the editor must automatically reroute the edges to keep them visually connected to the node. If a handle of an edge represented by a bezier curve is dragged, not only the position of the handle is changed but the whole bezier curve.

An exception can occur, when editor or application apply a policy that restricts the layout in some kind. This includes the strict policy of allowing no change by the user at all (applying only automatic layout) or just restrictions in e.g. the size of nodes. The editor could for instance enforce a minimum size for all nodes, to make sure they are always easily recognizable on screen.

2.3.3 Editing Attributes

This third class is comprised of actions that concern attributes of graph objects. More specifically, they concern attributes that may not be accessible in the direct manipulation editing scheme. They do not directly affect the logical graph structure.

Use Case "Editing attributes"

Description: attributes of a graph node or edge are changed

Actors: user, editor, application

Triggers: the action can be triggered by user or application

Result: a new screen representation with unchanged logical structure

Basic course:

One or more graph objects are chosen.

User or application provide new values for one or more attributes of these objects.

The editor presents the graph with new attribute values to the user.

Alternative courses, exceptions:

Again, the action of choosing one or more objects can be explicit by selection, or implicit by e.g. convention.

The attributes to be changed are usually semantic attributes like the name of a node, or something like the class specification we have seen in section 2.1.1. Semantical attributes do not always have to be visually presented in the editor, as we also have seen in section 2.1.1. It is not necessary, that the attributes be semantic in nature. It is possible that they also concern presentation issues like color or font.

The editing facilities for most of these attributes are not covered by the direct manipulation paradigm of the editor. Usually something like a dialog box or another conceptually independent editor is used for them. This kind of task is included in the graph editor requirements, because they may be triggered from within the editor and because the editor has to provide access to the triggers. The common context sensitive popup menu is an example for means to provide this access.

Exceptions can for instance occur when attribute values are illegal in application terms, or the actor does not have permission to change or read the attributes.

Changing attributes does in general not lead to structural changes of the graph. It may be possible however, that new attribute values lead to an inconsistent state in application terms. Automatic measures to rectify this inconsistent state could then lead to new attribute values for other graph objects or even to structural changes.

2.3.4 External Operations

The last class of operations performed on the problem domain are those concerning persistence and calculations on the graph that are not directly related to the editor.

We do not give a semi formal task description here, because the course of action varies too much from application to application. As far as the user is concerned, requirements for persistence operations, i.e. "load" and "save", are straight forward: when the graph is made persistent by a save operation, it should later be possible to revoke this persistent representation by a load operation.

Applications vary in what parts of the graphs are to be made persistent. This includes the level of detail as well as the level of integration. The level of detail is graded from storage of the logical structure only, storage of semantic attributes, storage of the presentation, up to storage of the current state of the presentation, e.g. which objects are currently selected. Some applications may only want to store the appearance of the diagram but are not interested in which objects are currently selected. Other applications only concern themselves with the logical structure and semantical attributes, ignoring all presentation issues like color, size and node positions. The level of integration of these load/save operation also varies: some may want to treat them as operations on the whole application problem domain as seen in section 2.1.1, others as in 2.1.3 may want to store a specific diagram only; others again might want to be able to do both as in 2.1.2.

For most applications, the main task is not to edit and display graphs, but to perform operations on them. These operations are application specific functions. For some of them, the graph editor has to provide access to the user. Again, the context sensitive popup menu paradigm is an example of how to provide this access. Some of these operations will involve the whole application problem domain of which the graph may only be a small part, some may involve the whole graph, others may involve only parts of the graph. For operations on parts of the graph, the user may want to choose from within the editor on which graph objects the operation is to be performed. The selection concept of most editors gives an example on how to provide that.

2.4 Functional Requirements

We have reviewed only three examples of graph editors and can of course not claim any kind of completeness, but the examples above should already give an impression of what kind of editors we have in mind. This section will now define the properties and functional requirements we expect from generated graph editors more precisely.

Since we do not discuss a concrete graph editor but rather the whole class of editors we want to generate, the following section cannot describe concrete features the editor

should implement. We can also not demand such desirable goals as a good usability of the generated editors, since this depends heavily on the specification of that editor. We can certainly not demand to automatically get an editor providing high usability standards for all kinds of applications. For that to be possible, we would at least have to specify a priori which concrete usability features the editor should provide for all applications and would thus arrive at already fixed interaction and presentation styles. Our main goal on the other hand was to be free in the style of presentation and interaction. This includes the freedom to specify editors with user interfaces of inferior quality. It is however a requirement for the specification language and the generator to be able to produce editors with good usability.

As we have seen e.g. in section 2.1.1, a graph editor usually is part of a larger application. It is a component that interacts with the rest of the application. It is responsible for presenting a graph structure to the user and provide means for editing that graph structure.

The problem domain of the graph editor is a graph-like data structure as described in section 2.2. This data structure usually is a small part of the whole application problem domain. The graphical representation is in the style of a direct manipulation user interface with a pointing device, e.g. a mouse. Which exact presentation style is used may vary from editor to editor.

It is necessary to separate the responsibilities of the application from those of the graph editor, and thus from those of the graph editor specification.

With respect to the graph structure, the application is responsible for:

- Syntactical correctness of the structure
  This syntactical correctness is also often called consistency. Since the graph editor is not always the only component of the application that modifies the internal data structure, it is possible and quite usual, that an existing application already employs a certain policy towards consistency of the data structure. This may range from only allowing consistent or syntactically correct structures at any time like in syntax directed editing, over structures, that are not always required to be consistent and only checked at specific times, up to completely arbitrary structures that do not even have a concept of syntactical correctness. We have seen an example for structured data that is only checked at specific times in section 2.1.2 and a completely free style of drawing in section 2.1.3.

  The editor should be able to use functions of the application to enforce the desired style of editing behavior.

- Semantical correctness of the structure
  Again existing applications usually already provide means to check or enforce se-

mantical relationships in their problem domain. The editor should be able to use these facilities in way of application functions, but does not provide such mechanisms by itself.

- Automatic layout according to semantic properties
  While this is usually not a part of the existing application, automatic layout, especially if it is based on semantic or other properties, is not an integral part of the graph editor, but a separate component that interfaces with the editor. This kind of architecture is more flexible and is also used widely in industry[6].

Responsibilities of the graph editor are:

1. The editor provides a component that displays a graphical and graph-like presentation of the internal data structure.

2. It is required that this editor component can be integrated into the user interface of the application. The integration should be seamless for the user.

3. The editor should provide the interactive operations identified in section 2.3:

   a) modifying the internal data structure

   b) changing diagram appearance

   c) editing attributes of the graph objects

   d) provide access to application functions from inside the editor

   In point c) it is also acceptable to just provide access to application functions for editing the desired attributes. See also section 2.3 for a detailed discussion of these four interactive operation classes.

4. It should provide a mechanism for making diagrams persistent and use those persistent diagrams in the editor again at a later time. This load/save mechanism will usually be integrated into a persistence mechanism of the whole application problem domain.

5. The editor architecture has to provide an interface to the above mentioned layout algorithm

6. The editor should enforce the desired interaction style by ways of the above mentioned application functions for syntactical and semantical consistency of the problem domain structures.

[6] see for instance the graph editor and graph layout libraries from Tom Sawyer Software

Interactions may be triggered by the user, the editor itself and the application. In the case of application triggered data structure modifications, a third party incremental layout algorithm is usually needed.

2.5 Non Functional Requirements

The non functional requirements for generated graph editors are:

1. They should support multiple platforms
2. The average response time of the graph editor should lie below the limit of 150 ms for mouse operations on graphical user interfaces [37, p. 367] on a current technology computer[7]

[7] the prototype will be evaluated on a Sun Ultra 60 workstation

3 Requirements for the Graph Editor Generator

This chapter covers the requirements the specification language and the generator for graph editors have to meet. Section 3.1 deals with aspects of the specification language, section 3.2 describes functional requirements of the generator while section 3.3 states the generator's non functional requirements.

3.1 Specification Language

The high level goals we stated for the specification language were: a free description of interaction and presentation style, the possibility for full integration of the editor into a larger application, and multiple layers of abstraction.

Below the list of requirements for the specification language in more detail:

Concepts The language should reflect a conceptual separation of application, dialog control and presentation.

Representation The language must at least have a textual representation that can be edited with an usual text editor.

Communication The specification language should be easy to use, i.e. more readable and compact than writing code or common natural language, and provide clearly defined concepts for easy communication of the content of specifications.

Abstraction Our main purpose was to reduce the development time and cost for graph editors. Therefore specifications should be shorter and more abstract than a normal general purpose programming language like C++ or Java for this particular topic.

Detail In chapter 2, we identified it as vital, that the editor developer has full control over all details of the underlying graphics system. The specification language has to reflect that.

Reuse To reduce development costs, it is necessary that specifications of graph editors can be reused for new editors, and similar to object-oriented techniques that they can be extended without being changed. In this way it is possible to build up libraries of presentation or interaction specifications that can be used in more than one place.

Modularity To support the above mentioned ability for reuse and extensions, the specification language should provide a module concept.

Integration It has to be possible, to specify and customize the integration of the generated editor with existing applications.

Usability The language must allow the specification of editors that meet the functional and non functional requirements described in sections 2.4 and 2.5.

Please note, that the goals of providing full access to all detail of an underlying graphics system and at the same time abstract specification are contradicting. The specification language should leave the decision to the editor developer, which level of detail control to use.

The following two items are statements, what the specification language is explicitely not required to provide.

1. As we already stated in the introduction, the scope of this thesis does not allow for a specification language that also covers automatic layout algorithms. The language and the generated editors should allow for the integration of third party modules for this purpose though.
2. It is not part of the specification language to describe any kind of consistency or semantical correctness of the application problem domain. The developer should instead be able to use application functions in the specification to produce an editor that employs the desired style of interaction ranging from rigid syntax directed editing to free drawing. These application functions could be provided by a technique complementary to the one presented here.

The specification language should allow for the following extensions to be made in future research without major changes to the underlying concepts:

1. A graphical or partially graphical representation of graph editor specifications.
2. An integrated development environment for graph editors, using these graphical or semi-graphical specifications.

3. Either a collection of or a method for specification of automatic and/or incremental layout algorithms.

3.2 Functional Requirements

It is the task of the graph editor generator to transform an abstract graph editor specification into code for the concrete graph editor defined by that specification. Since the editor will be integrated into a larger application, it will at some point be necessary for the generator to use information about the application. This can either be by analysis of application code as shown in figure 3.1 or by other means.

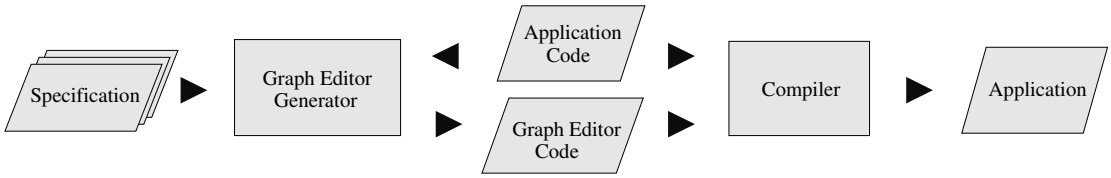


Figure 3.1: The process of generating a graph editor

From the main requirement of the generator to transform a graph editor specification into semantically equivalent graph editor code it also follows that, with an appropriate specification, the generator is required to produce code that meets the functional and non functional requirements defined in sections 2.4 and 2.5.

The design of the generator should allow at least for the following extensions:

- A frontend able to cope with a graphical representation of the specification language.
- Use of the generator in an integrated development environment.
- A relatively easy extension of the specification language that reflects minor changes in the underlying graphics system, e.g. the introduction of new low level interaction events.

3.3 Non Functional Requirements

The generator should support at least the same platforms as the generated editors. The implementation language does not need to be the same as the language of the generated code.

Part II

Design

4 The Specification Language

This chapter describes the specification language for graph editors. It is structured into one section about the specification of the problem domain, one about dialog control, one about presentation, and one about editor styles.

The graph editor specifications of this thesis are based on the object oriented programming language Java [14]. It provides a standardized and relatively high level access to the graphics system and is supported by multiple platforms, thus directly satisfying the portability requirement of section 2.5. The concepts of the specification language are not restricted to Java and should be directly applicable to most other, comparable object oriented programming languages.

4.1 Specifying the Problem Domain

The abstract problem domain of the generated editor is fixed. It is a graph structure as described in section 2.2. Our requirements analysis in 2.1.1 to 2.1.3 and 2.2 yielded, that the problem domain of the whole application on the other hand is in most cases not a graph itself, but does only contain graph-like parts instead.

As a consequence, the specification language must provide means to describe the mapping from the application domain to the graph domain. It must contain the partitioning of node and edge sets and a mapping from application data structures to node and edge partitions.

It is of advantage to simplify this mapping by assuming that each application class having node or edge properties maps to exactly one partition. This way, the mapping can easily be described by a simple enumeration.

Example 4.1

```
nodes Class, Note;
edges Inheritance, Association, Aggregation, Anchor;
```

The enumeration in example 4.1 divides the set of nodes into two partitions. It maps the application classes `Note` and `Class` one to one onto these partitions. The editor will

treat the application class `Note` as model for the one node partition and `Class` as model for the other one.[1]

This simple enumeration technique has the following properties:

- problem domains that are very similar to graphs in the sense that an implementation of the problem domain already provides all graph aspects are easiest to use in the editor specification. They will require the least additional integration work by the editor developer.
- problem domains that are hard to map to a graph structure, require more work from the editor developer. This work is on the level of coding in a programming language, not in the form of an abstract specification.

These properties meet the requirements of section 2.4, because they allow a simple integration of applications with graph-like problem domains. The less graph-like the problem domain is, the less appropriate a graph editor for this problem domain will be.

The concrete syntax of the specification language with regard to the mapping of the problem domain to the graph structure is then:

```
graphModule ::= "graph" ident "(" graphMapping* ")"
graphMapping ::= nodeMapping | edgeMapping
nodeMapping ::= "nodes" qualIdent* ";"
edgeMapping ::= "edges" qualIdent* ";"
```

where the terminal `qualIdent` is a qualified Java identifier, denoting an application class, and `ident` is an unqualified Java identifier, denoting the name of the graph.

From section 5.1 the following type constraints apply:

- each identifier in a `nodeMapping` is an application class implementing the interface `grace.model.Node` as described in section 5.1 or an subinterface thereof.
- each identifier in an `edgeMapping` is an application class implementing the interface `grace.model.Edge` as described in section 5.1 or an subinterface thereof.

Section 5.1 gives more insight into how these application classes are used inside the editor and how a tight application integration is possible.

[1] see also section 5.1 on the model role of application classes

4.2 Specifying the Dialog Control

4.2.1 Methods of Specifying the Dialog Control

In general model based user interface generation, a variety of specification techniques has been examined. Many of them also include a special technique for the specification of the dialog control part of the user interface.

Such specification techniques are for instance event automata, state transition systems, regular expressions, contextfree grammars, or context sensitive grammars. They differ in the level of abstraction they provide and in the application range they are aimed to support. Grammar specifications of interaction tend to be more abstract and are best suited for relatively long (i.e. in the number of interaction steps) and complicated dialogs. Event and state transition based specifications tend to be more concrete and seem best suited for simple dialogs.

The requirements of abstraction and detailedness in section 3.1 restrict us to techniques that can be used in both an abstract and more concrete specification style. We choose state transition systems as the basis of our further considerations. We prefer them to pure event based systems, because the state concept provides more ease of use for the editor developer (the communicatability requirement in section 3.1) and each purely event based system can easily be transformed into an equivalent specification of a state transition based system.

After introducing state transition system in a more formal way, we have in the following sections to address the problems, of how to provide different layers of abstractions, how to provide extendable interaction specifications and to how to keep the size of specifications as small as possible.

4.2.2 Dialog Control Automata

Definition 4.1
A dialog control automaton DCA D is a tuple $D = (E, S, T, s_0, A)$ consisting of a set E of events (also called tokens), a set S of control states, a set $T \in S \times E \times P \times Code \times S$ of transitions, a start state $s_0 \in S$ and a set of application states A. Transitions are tuples $(s, e, p(a), c, t)$ of a source state $s \in S$, an event $e \in E$, a predicate $p(a)$ over the state $a \in A$ of the application, an action $c \in Code$, and a target state $t \in S$.

In our context, the action is a piece of Java code to execute when the transition is fired. It may change the state of the application. *Code* is technically the set of all *block_statements* as defined by the Java Language Specification [14] in section 14.2.

A state of the application $a \in A$ is itself a complex semantical object including the values of all objects of the application etc. Since it is not the goal of this thesis to provide the formal semantics of the Java programming language, we refer the kind reader to projects like Bali [32] for further information on that topic. For our considerations about dialog control automata, we will treat the state of the application as an abstract value, that can be changed by action code of a transition, and that can determine the truth value of a predicate $p(a)$.

Definition 4.2
The *current state* of the DCA is the pair (s, a) of the current control state $s \in S$ and the current state of the application $a \in A$.

Dialogs

A DCA works very similar to a finite automaton. It accepts sequences of tokens and fires state transitions. Basically, a transition $t = (s, e, p(a), c, t)$ is fired, when the DCA is in state s, receives an event e and the predicate $p(a)$ yields true for the current state a of the application. When a transition is fired, the action c is executed and the DCA changes its state into the new state t.

More formally:

Definition 4.3
For input e in control state s of a DCA, a transition $t = (s_1, e_0, p(a), a, s_2)$ is called *applicable* when $e = e_0 \wedge s = s_1 \wedge p(a)$ for the current application state $a \in A$. $app(s, a, e) \subseteq T$ is the set of all applicable transitions for input e in state (s, a)

Definition 4.4
A *dialog step* d_i is a tuple $((s, a), e, (s', a'))$ with

$$app(s, a, e) = \emptyset \wedge (s, a) = (s', a') \ \vee \ (s, e, p, c, s') \in app(s, a) \wedge p(a) \wedge (c, a) \rightarrow a'$$

where "$\rightarrow$" $\subseteq Code \times A \times A$ is a relation describing the semantics of action code c, relating the application state a before execution of c with an application state a' after execution of c.

Definition 4.5
A *dialog* d is a series $d = (d_i)_{i \in \mathbb{N}}$ of dialog steps with

$$\forall d_i, d_{i+1} \in d.\ d_i = ((s, a), e_i, (s_i, a_i)) \ \wedge \ d_{i+1} = ((s_i, a_i), e_{i+1}, (s', a'))$$

The intention is, that each state (s_i, a_i) in a dialog d has a visualization, being presented to the user when the DCA and the application are currently in that state.

Since the transitions of a DCA determine the order and possible values of the (s_i, a_i), a DCA defines the set of all possible dialogs.

In general, there may be one, more than one or zero applicable transitions for each input e and state (s, a).

A DCA is called *deterministic* when for each sequence of input events $(e_i)_{i \in \mathbb{N}}$ there is exactly one dialog d. This implies that there may be at most one transition for each state (s, a) and input e. Since we are aiming to specify user interface dialogs for graph editors, we wish the DCA to be deterministic.

A deterministic dialog is not only much easier to understand for the the editor developer, but also for the end user of the editor. A deterministic DCA does not imply that the dialog has to be linear in the sense that the user has at each time exactly one valid path to choose from (or not to choose from that is). It only leads to a user interface being deterministic in its responses, i.e. the user gets the same output for the same input. Or more formally: For each sequence of input events $(e_i)_{i \in \mathbb{N}}$ there is exactly one dialog d.

Please note, that the DCA can only be deterministic in the sense defined above, when the semantics "$\rightarrow$" of the action code pieces is deterministic. For Java this is only easily seen to be true, when there is only one thread of control.

4.2.3 Applying State Transition Systems to Graph Editors

This section applies the dialog control automata discussed above to our task of specifying the interaction part of graph editors.

The properties requirements of section 3.1 demand for this part of the specification language are: short, simple, modular, reusable, extendable and providing multiple layers of abstraction.

The first basic idea of how to apply DCA to graph editors might be to use one DCA to describe the interaction properties of the editor. Although this is certainly possible, it does not really satisfy our requirements of reusability and modularity. The conceptual modules of a graph editor are the editor panel itself and the objects that can be modified on this panel, i.e. the nodes and edges of the graph. The user perceives these objects as having a separate interaction behavior. When repositioning a node, she perceives to interact with a node in the editor, not with the whole user interface of the application. This kind of modularity in interaction and also presentation aspects of parts of the user interface is mirrored in the software architecture of common current user interface toolkits as for instance Java Swing. They provide abstract interaction objects like pushbuttons or menus and encapsulate the interaction and presentation behavior in these objects.

For the specification of graph editor interaction aspects, we will follow a similar approach. The interaction of the editor will not be specified by one large DCA, but by multiple DCAs working together. Following the conceptual model of a graph editor, we will assign one DCA to each class of nodes, to each class of edges and to the editor panel itself.

This has the following consequences:

- The DCAs may have to communicate in some way.
- The single DCAs should each be much smaller and simpler than one large DCA.
- The expressiveness will not be greater than with one large DCA, because a set of DCAs can be simulated by one DCA.
- Since the interaction specification is divided into modules, it is easier to reuse parts of one graph editor specification in another.

So there are four problems for the specification language still to address:

1. In section 4.2.2 we stated, that a DCA should be deterministic. The specification language should provide a mechanism to either only allow deterministic DCAs or to transform a non deterministic specification of a DCA into a deterministic DCA.
2. How multiple DCA communicate
3. How DCAs can easily be extended
4. How to provide multiple layers of abstraction

To solve problem one, we will use a very pragmatic approach making deterministic DCA from a non deterministic specification. We assign distinct priorities to all transitions. When two transitions are applicable, the DCA always chooses the one with the higher priority. A simple, but effective priority assignment is for instance the line number in the specification. If transition a comes before transition b in the specification and both are applicable, then transition a will be fired. The prototype implementation of the graph editor generator uses a slightly different priority assignment arising out of problem three. It is described in detail in section 4.2.4

To solve problem two, we allow one DCA to send an input token to another DCA in the action part of a transition. In this way, for instance the DCA of a node can initiate a state transition in the editor.

Problems three and four are discussed in section 4.2.4 and 4.2.5 respectively.

Example 4.2
A textual, very simple, abstract interaction specification of an edge or node could then look somewhat like this:

```
automaton Selectable {
  tokens select, deselect;
  states not_selected, selected;

  >not_selected:
    (select) -> selected
  selected:
    (deselect) -> not_selected
}
```

This describes a DCA `Selectable` with two states, `selected` and the initial state `not_selected`, and two transitions between them. For e.g. token `select` in state `not_selected` the new state is `selected`. The example does not contain any preconditions or action code.

4.2.4 Extending Dialog Control Automata

Section 3.1 requires the specification to be extendable. For DCA there is a very simple and powerful technique to provide that extensibility. Existing DCAs can be extended in a way that it analogous to the concept of inheritance in object oriented programming languages.

In object oriented languages, classes, that are in some sense more general, are extended by classes, that are more special. For DCAs we adopt this notion and say that DCAs with fewer states and transitions are more general in their behavior than DCAs with additional states and transitions.

Definition 4.6
DCA $Y = (E_Y, S_Y, T_Y, s_0), A_Y)$ *extends* DCA $X = (E_X, S_X, T_X, s'_0, A_X)$ when $E_X \subseteq E_Y$, $S_X \subseteq S_Y$, $T_X \subseteq T_Y$ and $A_X = A_Y$.

So when Y extends X it has at least all the states, tokens and transitions of X and works on the same application. If we connect the DCAs not with the whole application, but with a particular application class instead, we can relax this further and let X also work on subclasses of the application class of Y. If Y extends X we also say Y is *derived* from X.

We can use this concept to create a notation analogous to object oriented languages that simplifies the construction of derived automata. This way, we are able to specify automata, that are small, simple and general as for instance the example in section 4.2.3, and then extend them to specialize it for a certain application.

Example 4.3

```
automaton Activatable extends Selectable {
  tokens activate, deactive;
  states deactive;

  >deactive:
    (activate) -> not_selected
  selected:
    (deactivate) -> deactive
}
```

The DCA `Activatable` is derived from the DCA `Selectable`, the example of section 4.2.3. It consists of three states: the states `selected` and `not_selected` from DCA `Selectable` and a new initial state `deactive`. The transitions describe an object that can be selected when it is not selected, can be deselected when it is selected (the behavior from `Selectable`) and additionally can be deactivated (for instance by another DCA) when it is selected and reactivated when it is deactive. The general behavior of `Selectable` is preserved, while it is at the same moment specialized.

The possibility to easily create derived DCAs leads to an equal possibility of introducing unwanted non determinism into a specification. Consider the following

Example 4.4

```
automaton A {
  tokens x; states a,b;
  a: (x) -> b
  b: (x) -> a
}

automaton B extends A {
  a: (x; some_condition) -> a
}
```

DCA `B` now has a non deterministic specification, provided condition `some_condition` $\neq$ `false`. The basic line number priority assignment described in section 4.2.3 would technically work, but could produce unreliable and counter intuitive results. The priorities of transition `a:(x) -> b` and `a:(x; some_condition) -> a` now depend on which

automaton is declared first and given the most natural order above, the new transition of DCA `B` would never be fired, because all transitions of `A` appear before it.

Intuitively one would expect the transition `a:(x) -> b` to be fired by default and under some special condition, i.e. when `some_condition` yields `true`, the more special transition of DCA `B` to be fired instead.

We can reach this behavior easily by modifying the above transition priority assignment. When a DCA B extends DCA A, we say, that all transitions of B have a higher priority than those of A.

When used properly by the editor developer, this turns the danger of unwanted non determinism in the interaction specification into a intuitive and powerful tool for specializing and extending existing specifications.

4.2.5 Providing multiple Layers of Abstraction

In the sections above we saw, how one can use dialog control automata to specify the interaction properties of graph editors in a relatively short, simple, modular, reusable and extendable way. We did not yet cover the issue of multiple degrees of abstraction we have required in 3.1.

The level of abstraction a DCA provides depends on several factors: The level of abstraction of the input events, the level of detail of the predicates of transitions, and the level of detail in the action code of transitions. The examples of DCAs provided in the sections above were relatively abstract. They did not involve any action code at all, one precondition, and abstract, logical input tokens.

In the examples we did not discuss, where these logical input tokens came from. When writing an usual user interface, the developer is working with a set of relatively low level input events provided by the underlying windowing toolkit – in our case the Java AWT. Examples of such physical input events are *mouse moved* or *mouse button pressed*. For an abstract specification of user interface behavior, we are interested in logical events like *button pressed*, *object selected* or *action x triggered*. The Java AWT for instance does support the use of abstract tokens in a class `java.awt.event.ActionEvent` that can be associated with one or more abstract interaction objects like buttons or menu items. These logical action events are then used by the higher level dialog control part of the user interface.

It is the purpose of those abstract interaction objects to decouple the rest of the user interface from low level physical events like *mouse moved*. An action or in our terminology a logical DCA token can be generated by multiple different abstract interaction objects,

that themselves could require a maybe complex sequence of physical events first. A logical token *action x triggered* could for instance be generated by a menu item showing the text for action x, by a button showing a similar text or by a drop down list where action x is one of the choices presented to the user. The sequence of physical events required to trigger the action x is different in each case, while the higher level dialog control of the user interface always receives the same logical token.

The `Selectable` DCA of example 4.2 for instance is a high level specification of the behavior of a node. It uses logical tokens `select` and `deselect`. It does not specify, how these tokens relate to a sequence of physical events.

In sections 2.1.1 to 2.1.3 we saw, that it might be necessary for the editor developer to describe interactions in the graph editor at even this level of detail to maintain a consistent look & feel for a whole application.

Since we are looking for a way to describe possible sequences of interaction events and their relation to application states, we can again conveniently use DCAs for this purpose. The input of the DCA is then not a logical token but a physical event instead. The most important side effect of a transition action is now not a change in state of the application, but the construction of a logical token for a more abstract dialog control automaton.

We introduce some new bits of concrete syntax in the specification language, to support the editor developer in designing a low level DCA. We could of course use the same description technique as for the abstract DCAs, and we could even add direct support for an abstract interaction object to not only have two DCAs controlling it (abstract and low level) but a whole hierarchy of DCAs instead. We do not do so here for the following reasons:

A hierarchy of DCAs can describe more complex dialogs in a more compact way. A hierarchy has on the other hand the disadvantage of being harder to understand, because the behavior is more spread out into different DCAs. We also claim, that the dialogs in graph editors remain relatively simple, especially when the abstract and low level interaction is decoupled as described above. We can of course not give a proof here for that claim, but our examples for the graph editor generator prototype do at least seem to support it.

The most important reason is, that a clear distinction between two levels gives the possibility to support the developer in each level with more specific specification aides. These specification aides may be of a very technical nature, but add significantly to the usability of the specification language.

One such aid for instance is rooted in the fact that the physical location of an input event like a mouse click is very important for the behavior of an graph object in the editor. If for instance the click originated inside a node, it may lead to a `select` token

being generated and sent to that node, while a click on the editor background may instead deselect all components or do nothing. Since this testing if a physical event originated in the inside of the presentation of an object is a very common thing in a low level interaction specification, it is useful to provide a convenient extension of the specification syntax for this purpose. For abstract DCAs on the other hand, this aid is completely meaningless and therefore most probably confusing for the developer.

Another example is, that low level DCAs are very tightly associated with the presentation aspects of the abstract interaction object they describe. Abstract DCAs do not use properties of the presentation, but instead determine by their control state, what the presentation should look like.[2]

The following DCA is an example of a low level, concrete interaction specification.

Example 4.5

```
interactor ClickInteractor {
  requires node;
  states start;

  >start:
    (clicked) sends Selectable.select to editor
}

interactor DragInteractor extends ClickInteractor {
  states dragging;

  classcode {
    private int x, y, dx, dy;

    private void update(MouseEvent e) {
      dx = e.getX()-x;  x = e.getX();
      dy = e.getY()-y;  y = e.getY();
    }
  }

 >start:
    (pressed) -> dragging do { update(event); }

  dragging:
    +(dragged) do { update(event); parent.translate(dx,dy); }
    +(released) -> start
}
```

[2] see also section 4.3.2

Examples 4.5 defines two low level DCAs: `ClickInteractor` and `DragInteractor`. The `requires node` defines, that these interactors can only be associated with the presentation of graph nodes. Instead of only requiring, that the DCA is associated with the presentation of a node, we could also specialize `ClickInteractor` further and require a specifically named presentation. The interactor would then be applicable to that presentation and all presentations extending it.

`ClickInteractor` consists of one state `start` and one transition that is fired when a `MouseClicked` event is received. When the transition is fired, an abstract token `select` for the DCA `Selectable` of example 4.2 is generated and sent to the automaton of the graph editor.

`DragInteractor` extends `ClickInteractor` and provides the additional user interface functionality of dragging a node presentation with the mouse. It defines one new state `dragging`. It includes a short piece of custom code into the DCA that can directly be used in the action part of transitions. If in state `start` a `MousePressed` event is received, the action `update(event);` is executed and the current state is changed to `dragging`. The action code uses one of the technical aides: the implicitly declared variable `event` containing the low level event that was actually received from the underlying windowing toolkit. The transitions of state `dragging` both demonstrate another two of those aides. First: they are preceded with the character `+`. This disables the automatically carried out check, if the event originated from a screen position inside the node presentation associated with the low level DCA. Second: the action code references an implicitly declared variable `parent` containing the presentation associated with the DCA.

Together with the concepts of modularity and extendability, we arrive at the following levels of abstractions which we can use for DCAs in the graph editor specification:

- assign existing high level DCAs to graph objects for most abstract description
- extend existing high level DCAs for more detail
- write own high level modules
- use existing modules of low level DCAs
- extend existing lowlevel modules
- write own low level DCAs

4.2.6 Syntax

This sections gives an overview for the syntax of the graph editor specification language with respect to interaction.

We use a BNF notation with regular right sides. Terminal symbols like `"automaton"` are enclosed in double quotes. The only terminal symbol apart from that is `ident`, which is a Java identifier as defined in [14].

The syntax of abstract dialog control automata descriptions:

```
automatonModule ::= "automaton" ident [extends] "{" automatonSpec* "}"

automatonSpec   ::= stateDecl | tokenDecl | import_statement |
                    classCode | stateDef

stateDecl       ::= "states" ident* ";"
tokenDecl       ::= "tokens" ident* ";"
classCode       ::= "classcode" block

stateDef        ::= [">"] ident ":" transition*
transition      ::= "(" ident [";" conditional_expression] ")"
                      ["->" ident] ["do" block]

extends         ::= "extends" qualIdent
qualIdent       ::= ident ("." ident)*
```

The non terminals `import_statement`, `conditional_expression` and `block` are Java code and defined by the Java grammar in the Java Language Specification [14].

The syntax of low level dialog control automata descriptions:

```
interactorModule ::= "interactor" ident [extends] "{" interactorSpec* "}"

interactorSpec   ::= stateDecl | classCode | import_statement |
                     interactorReq | interactorState

interactorReq    ::= "requires" ("node" [ident]|"edge" [ident]|"editor") ";"

interactorState  ::= [">"] ident ":" interactorTrans*
interactorTrans  ::= ["+"] "(" ident [";" conditional_expression] ")"
                       ["-> ident] [send] ["do" block]

send             ::= "sends" qualIdent ["to" ("editor" | qualIdent)]
```

4.3 Specifying the Presentation

4.3.1 Presentation objects

There is a variety of methods to specify the presentation aspects of a standard user interface. Many of these specification methods like e.g. the BOSS System [38] use abstract interaction objects like buttons as their basic building blocks. As we are dealing with a direct manipulation interface and are aiming to build new abstract interaction objects like presentations for specific node classes, these techniques are at least not directly applicable.

An abstract interaction object is formed by a presentation and a translation of a sequence of physical input events into a logical token. In our context that means, an abstract interaction object is a presentation together with a low level dialog control automaton.

The abstract interaction objects in our graph editor are the editor panel, the nodes, and the edges in this panel. Instead of applying a general specification technique, we again follow a very pragmatic approach as do many other graph editor related systems like e.g. GeneSys [18] or Graphlet [17]. We provide the editor developer with a set of graphics primitives. Additionally, we also provide the possibility to parameterize these graphics primitives. The parameterization can then for instance be used to combine existing primitives and to arrive at more complex structures.

We call these graphics primitives *figures*. They include figures for nodes e.g. boxes and circles, for edges, e.g. straight lines or orthogonally connected lines, and also figures for the editor to display for instance a selection frame. Additional primitives are figures to display and layout small amounts of text, or e.g. a parameterized figure that displays a selection box with resize handles around another node figure. A composite figure takes to other figures as parameter and e.g. paints the one over the other. See section A.2.4 for the complete set of figures that are supported by the generator prototype.

When providing a set of primitives a very important aspect is the ability to extend that set at a later time. Extending the set of figures should not leave the editor developer with the need of changing the graph editor generator. It should instead be possible to create an own, new graphics primitive and to use that figure like any other in future graph editor specifications. We achieve this by creating an open software architecture for figures[3] and by defining a clear interface that classes representing a figure should implement. It is then possible to use the name of a class implementing the interface `grace.figures.Figure` as a reference to that figure in graph editor specifications.

The concept of figures alone does not yet describe all presentation issues in a graph

[3] see also section 5.4

editor. There are still two questions open to consideration: how are nodes and edges connected and where are nodes laid out in the editor?

The first question is solved similarly to the concept used for figures by providing a set of algorithms that assign positions to the ends of edges from the properties of the node and the edge they are connecting. A node and an edge can for instance be connected at one of a set predefined positions at the node. Some tools and graph editor frameworks call those sites *ports*. Another connection algorithm calculates the intersection of the node border with a line between centers of source and target node of the edge. The set of algorithms can be extended in the same way and as easily as the set of figures. See section A.2.5 for the complete set of connection algorithms.

The second question is solved by an external layout algorithm, that assigns positions to nodes. The interface for these algorithms is defined in section 5.5.

4.3.2 Connecting presentation and dialog control

In section 4.2.2 we have defined a dialog as a series $(d_i)_{i\in\mathbb{N}}$ of dialog steps, where each dialog step d_i consists of a start state, an input event, and a next state. The intention was, that each state in the dialog should have an own visualization that is presented to the user. So the presentation depends on the current control state of the dialog and on the current state of the application.

From the section above we have, that an abstract interaction object consists of a figure and an interactor. From the considerations above, we have, that the whole presentation is a mapping from application state and control state to abstract interaction objects.

The dialog control state can easily be explicitly modeled in the specification, the current application state cannot. By defining the presentation as a mapping from dialog control state to abstract interaction objects that have access to the application state by ways of direct method calls, we can keep the specification language simple while not loosing any power of expression of the specification as a whole. Access to application methods from abstract interaction objects consisting of figures and interactors is controlled by the software architecture of the graph editor described in section 5.1. It could for instance consist of a Java interface that is implemented by a node class and that e.g. defines the node as having a text property that can be accessed by a text displaying figure.

For nodes, a presentation description could then look like example 4.6.

Example 4.6

```
node presentation BoxNode {
  requires automaton Selectable;
```

```
  not_selected:
    (Box, ClickInteractor, PerimeterConnection)

  selected:
    (SelectedNode(Box), DragInteractor, PerimeterConnection)
}
```

The presentation refers to the states of automaton `Selectable` from example 4.2 in section 4.2.3. It declates the state `not_selected` to be represented by an abstract interaction object that has a box like figure, an interaction described by `ClickInteractor` defined in example 4.5 and a connection policy, that paints edges at the border of the node. State `selected` is represented by an abstract interaction object consisting of a box like figure that is surrounded by a selection frame. It has the interaction capabilities of `DragInteractor` described in example 4.5.

For edges, the presentation description looks very similar. Because decoration at edge ends like different types of arrows is mostly independent of the edge presentation itself, we provide the possibility to specify those decorations separately and combine them freely with the main edge representation.

Example 4.7

```
edge presentation StraightEdge {
  requires automaton NoDialog;

  start:
    (StraightEdgeFigure, NullInteractor, DotDecorator, Arrowhead)
}
```

In example 4.7 we see a very simple edge presentation description. It uses the abstract dialog control automaton `NoDialog` that has one state `start` and no transitions. The one state is mapped to an abstract interaction object for an edge consisting of a straight line that has a small filled circle at the source and an arrowhead at the target end. The associated interaction description `NullInteractor` allows no interaction at all.

The editor panel itself does not need any special additional specifications, and is therefore simply a mapping from dialog control state to pairs of figure and low level DCA. Example 4.8 demonstrates a description of this mapping for a very basic editor panel, allowing no further interaction of its own.

Example 4.8

```
editor presentation BasicEditor {
  requires automaton EditorDialog;
```

```
  edit:
    (BlankEditor, NullInteractor)
}
```

In section 3.1 we have required the graph editor specifications to be extendable. We can apply the extendability concept of DCAs analogously to presentation specifications.

If presentation B extends presentation A, it contains all mappings that are listed in A together with all mappings listed in B. If there is a mapping for a DCA state listed in both presentation specifications, the mapping of the more special presentation B applies. B can only extend A when its associated DCA is the same as or extends the DCA of presentation A.

Example 4.9

```
editor presentation FullEditor extends BasicEditor {
  requires automaton FullEditorDialog;

  edit:
    (BlankEditor, ClickInteractor)

  selecting:
    (SelectionFrame, DragSelection)

  insert_node:
    (BlankEditor, ClickInteractor)
}
```

In example 4.9 the editor presentation `FullEditor` extends the `BasicEditor` mapping of example 4.8. Its DCA `FullEditorDialog` extends the DCA `EditorDialog` and contains two new states `selecting` and `insert_node`. The mappings for these two new states are added to the set of mappings of `BasicEditor`. The mapping for state `edit` in `FullEditor` overrides the one of `BasicEditor`.

4.3.3 Syntax

This section gives a brief summary of the syntax for presentation descriptions in BNF notation.

```
presentationModule ::= nodePresentation | edgePresentation | editorPresentation

nodePresentation   ::= "node" "presentation" ident [extends] "{" nodeSpec* "}"
```

```
edgePresentation   ::= "edge" "presentation" ident [extends] "{" edgeSpec* "}"
editorPresentation ::= "editor" "presentation" ident [extends] "{" editor* "}"

nodeSpec           ::= nodeState   | presentationReq
edgeSpec           ::= edgeState   | presentationReq
editor             ::= editorState | presentationReq

presentationReq    ::= "requires" ("automaton"|"type") [qualIdent] ";"

nodeState          ::= ident ":" "(" qualIdent ["(" parameters ")"] ","
                         qualIdent "," qualIdent ")"
edgeState          ::= ident ":" "(  qualIdent ["(" parameters ")"] ","
                         qualIdent "," qualIdent "," qualIdent ")"
editorState        ::= ident ":" "(" qualIdent ["(" parameters ")"] ","
                         qualIdent ")"

parameters         ::= parameter ("," parameter)*
parameter          ::= qualIdent ["(" parameters ")"]
```

4.4 Styles

In the sections above we have discussed specification techniques for describing the problem domain, the dialog control, and the presentation of graph editors. To arrive at a working graph editor we still have to describe how these single descriptions relate to one another.

We do this by defining a *style* for the graph editor. A style is a mapping from problem domain classes to abstract dialog control and presentation descriptions. It defines, which problem domain classes use which presentation and how they interact with the user. Since our problem domain is described by an enumeration of node and edge classes, this mapping can also be described by a simple enumeration:

Example 4.10

```
style BasicStyle {
  Editor    : (BasicEditor, EditorDialog)
  Channel   : (StraightEdge, NoDialog)
  Component : (Box, Selectable)
}
```

Example 4.10 defines a style `BasicStyle` for a problem domain that consists of a single node class `Component` and a single edge class `Channel` in analogy to the SSD editor in

section 2.1.2. It maps the editor panel to the presentation `BasicEditor` of example 4.8 and the abstract dialog control `EditorDialog`. The problem domain edge class `Channel` is mapped to the presentation `StraightEdge` of example 4.7 and the `NoDialog` DCA. Example presentation `Box` and DCA `Selectable` of example 4.2 represent the problem domain class `Component`.

This mapping is directly applicable to a problem domain as described above, that consists of one node class `Component` and one edge class `Channel`. Using the object oriented concept of inheritance, this style is also applicable to other problem domains. The problem domains using this style must satisfy the following constraints: Each node class must be `Component` or be derived from `Component`. Each edge class must be `Channel` or be derived from `Channel`.

In this way, we can write styles that do not yet contain the concrete problem domain classes of a particular graph editor, but that use abstract classes or Java interfaces to divide the problem domain into an abstract set of edge and node partitions. The style is then applicable to all problem domains that implement those abstract classes.

Like all other specifications presented above, style specifications can also be extended. If style B extends style A it contains all mappings contained in A and additionally those that are listed in B. If a mapping for the same problem domain class is listed in both styles, the mapping of the more specific style B is applied.

Example 4.11

```
style SSDStyle extends BasicStyle {
  Editor : (FullEditor, FullEditorDialog)
  Port   : (CircleNode, Selectable)
}
```

The style `SSDStyle` of example 4.11 extends `BasicStyle` of example 4.10. It overrides the editor mapping and now maps the editor panel to a presentation `FullEditor` with the DCA `FullEditorDialog`. The mapping for problem domain class `Port` is added to the set of mappings. It connects ports with a presentation that visualizes nodes with circles and a dialog control `Selectable`. The other two mappings of `BasicStyle` remain as they are.

The syntax of style specifications in BNF notation is very short:

```
styleModule ::= "style" ident [extends] "{" mapping* "}"
mapping     ::= qualIdent ":" "(" ident "," ident ")"
```

4.5 Conclusion

We have presented a specification language for graph editors. We claim, that it satisfies the requirements of section 3.1.

Concepts The specification language clearly separates the aspects of application, dialog control and presentation.

Representation The language has a textual representation.

Communication Dialog control automata and mappings are well defined and easily understood concepts. The specification language encourages the editor developer to use expressive names for each of the specified modules. These properties provide a more efficient means for human communication than pure programming language.

Abstraction The possibility to build and use specification libraries can produce very short minimal and abstract specifications. A 10 line problem domain description can result in 1000+ lines of Java code for a graph editor.

Detail The specification techniques for dialog control and presentation allow a developer controlled level of detail from very abstract (problem domain description only) down to very concrete (at the level of programming language code).

Reuse All specification modules can be extended without being changed and reused in future specifications.

Modularity The language is divided into a set of of multiple named modules.

Integration A generated editor can be integrated into existing applications either directly or by using the adapter design pattern[4].

Usability Specification of working graph editors as defined in sections 2.4 and 2.5 is possible.[5]

Because the editor developer does not need to know the particular problem domain beforehand when creating an editor style, it is possible to write a library of specifications that can be directly used on different problem domains for rapidly generating a first

[4] see also chapter 5

[5] see for instance the example specification in appendix B

editor prototype for that problem domain. No further specification or tailoring is needed for this first step.

At later stages of development, the style specifications, the dialog specifications or the presentation specifications can be extended or created step by step at the precise level of detail needed in each development step.

The complete syntax of the specification language is compiled in appendix A.1.1.

5 A Software Architecture for Graph Editors

Chapter 5 demonstrates a flexible and extendable software architecture for graph editors. It is applied by the graph editor generator prototype *grace*. We explore the basic building blocks of the architecture, an interface to automatic layout algorithms, and the integration of graph editor and application.

5.1 Overview

This section gives an high level introduction to the software architecture of graph editors generated by *grace*.

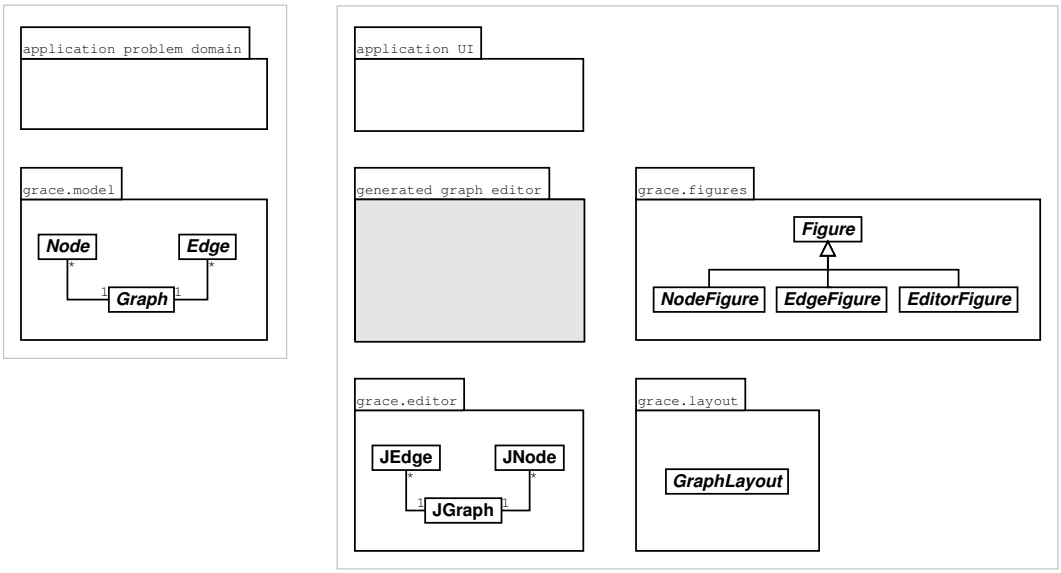


Figure 5.1: A software architecture for graph editors

Figure 5.1 shows that the generated code is only a small part of the resulting graph editor. Large parts of it are implemented in base classes of various *grace* Java packages.

The architecture as a whole is based on the Model/View/Controller design pattern[1]. The design pattern allows to decouple model, i.e. the problem domain, and presentation. It also allows to have multiple synchronized views on one model. Some of these views can be generated graph editors.

The left part of figure 5.1 shows the packages that implement the model role of the design pattern, while the view and controller roles are contained in the box to the right.

The model consists of the application problem domain classes together with the interfaces of `grace.model` that define the graph, node, and edge roles of the problem domain. Section 5.2 discusses, how the classes of application problem domain and `grace.model` are connected, and how existing applications are integrated.

The right part of the architecture diagram consists of the existing application user interface, the generated graph editor code, the base view classes in `grace.editor`, the layout algorithm interface in `grace.layout` and a set of figure implementations in `grace.figure`. The generated graph editor code is the mediator between the application classes, i.e. problem domain and user interface, and the three packages `grace.editor`, `grace.figures` and `grace.layout`. These three *grace* packages define basic properties and the fixed, standard part of the graph editor. The generated classes contain the specialized behavior that was described in the graph editor specification.

The dialog control, i.e. the controller role of the Model/View/Controller pattern, is implemented by generated classes and parts of `grace.editor`. Section 5.3 provides more detail on that.

Section 5.4 discusses the presentation aspects of the software architecture, i.e. the view role of the Model/View/Controller pattern.

The layout interface and integration of the application user interface with the graph editor are covered by sections 5.5 and 5.6 respectively while section 5.7 deals with persistence issues.

5.2 Application

In this section we will investigate, how the application problem domain can be connected to the graph editor.

The interfaces `Graph`, `GraphObject`, `Node` and `Edge` define for the rest of the graph editor the roles of the components of a graph, i.e. nodes and edges, and the graph structure as

[1] see also [7, p. 125]

a whole. The class diagram in figure 5.2 shows the relationships between them, forming an implementation of an abstract graph structure as defined in section 2.2.

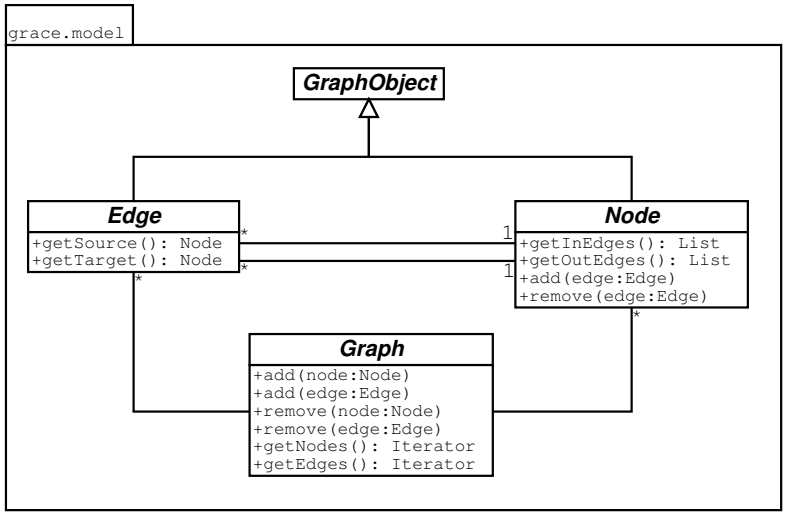


Figure 5.2: The model

Application classes that implement these interfaces are recognized by the remaining editor classes as implementing the role of a node, an edge, or of the whole graph structure. Graph editor specifications, that do not refer to any special properties of the application problem domain and only use the basic concept of nodes, edges and a graph structure can be directly applied to all graph-like problem domains. While it normally is not sufficient to only refer to these basic properties in a specification, it is possible to write general specification modules that only define basic behavior and do not need any specific problem domain properties. These general specification modules can later be reused and extended.

The simplest way of integrating the application problem domain with the graph editor is to let application classes of the problem domain directly implement these interfaces as in figure 5.3.

Figure 5.4 gives a concrete example for the SSD problem domain of AutoFocus described in section 2.1.2. It also demonstrates a possibility to provide the graph editor with more information about the problem domain without having to refer to a concrete application class. The application problem domain class `Component` representing a named component in an AutoFocus specification does not directly implement the interface `Node` but a more special interface `LabeledNode` instead. The interface `LabeledNode` is derived from `Node` and contains an additional method `String getLabel()` that can be used in

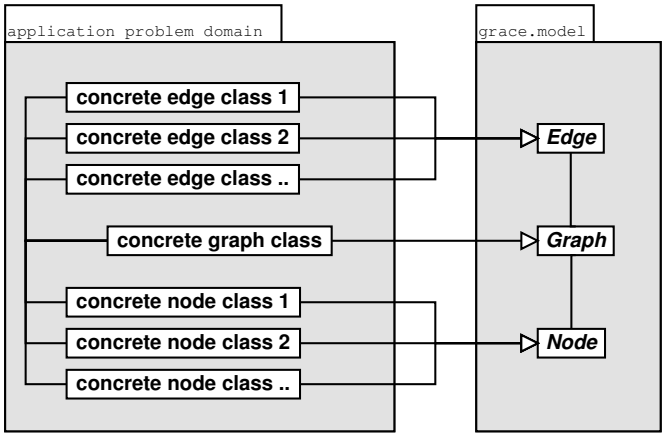


Figure 5.3: Direct problem domain integration

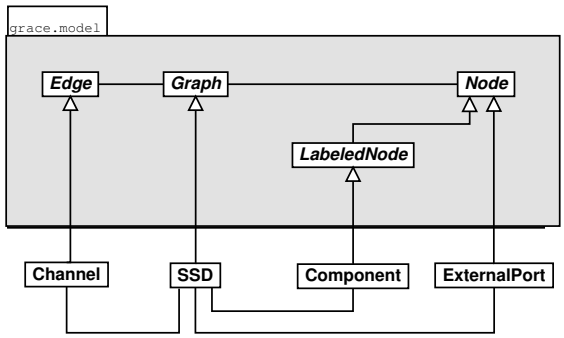


Figure 5.4: Example of direct problem domain integration

a graph editor specification to get a label to display in the editor from the application class. If the graph editor specification only uses the interface `LabeledNode`, it does not directly depend on an application problem domain class. The specification module referring to `LabeledNode` can be reused for any other problem domain containing nodes with labels. Most importantly, it can be reused, without needing to be changed. Together with the extension mechanisms described in chapter 4 it is possible to build libraries of specification modules to be reused in other, more specialized specifications.

Unfortunately, it is not always possible to let the application problem domain classes directly implement the `grace.model` interfaces. There are two common reasons for this: First, the application problem domain classes may not map easily one to one on a graph structure; some more or less complex computations may be necessary for that. Second,

the application problem domain classes already exist and cannot be changed for some reason.

In these cases, the adapter design pattern described e.g. in [13] opens a solution to still cleanly integrate the application problem domain with the graph editor. It is used to *convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces* [13, p. 139].

The adapter design pattern does not require the application classes to be changed and it allows for instance to convert complex relationships between application classes into an edge between two nodes of a graph.

Figure 5.5 shows the basic structure of the adapter design pattern applied to the graph editor architecture. We refer the interested reader to [13] for a more detailed and thorough discussion of the design pattern.

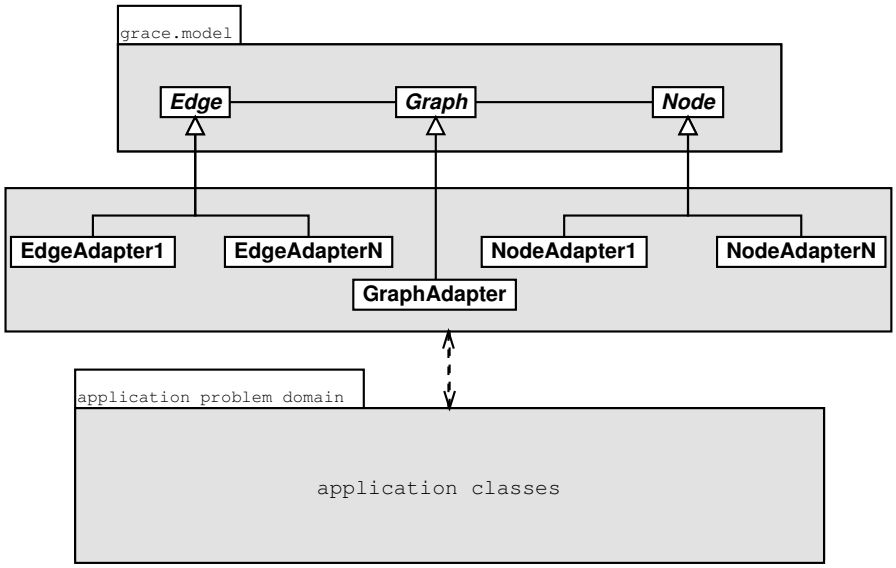


Figure 5.5: Application integration with the adapter design pattern

5.3 Dialog Control

The specification language uses two very similar techniques to describe interaction properties of graph editors: abstract dialog control automata for high level dialog control and low level dialog control automata for concrete interaction on the system event level.

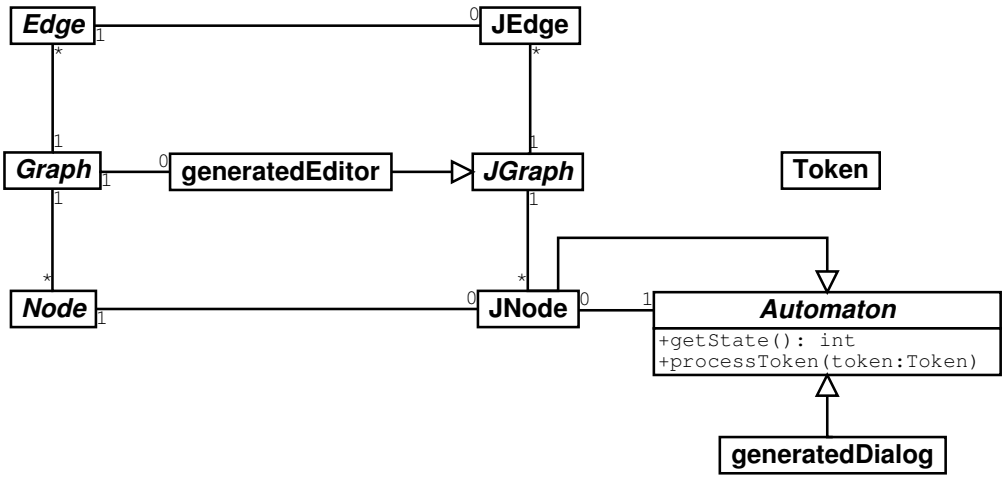


Figure 5.6: Abstract dialog control via delegation

Abstract DCA, defined in graph editor specifications by the keyword `automaton`, are represented in the software architecture by the interface `Automaton`, while low level DCA, introduced as `interactor` in the specification language, are represented by the interface `Interactor`. Both interfaces are located in package `grace.editor`.

Figure 5.6 shows how these generated classes are integrated into the software architecture by delegation[2]. To keep it simple, the diagram only contains an exemplary abstract dialog for a node.

Each object providing abstract interaction services implements the `Automaton` interface; each object providing event level interaction implements `Interactor`. Interactors together with presentations form abstract interaction objects. Generated interactors implement one or more of the event listener interfaces of the `java.awt.event` package. Their relationship with presentation classes is discussed in section 5.4. In this section we concentrate on the abstract dialog control, i.e. the controller part in the Model/View/Controller pattern that is independent of a particular presentation.

The `Automaton` interface contains two methods: `int getState()` and `void processToken(Token)`. As the name suggests, the `getState` method is expected to return the current control state of the automaton, while `processToken(Token token)` should fire the appropriate transition for input `token` in the current control state of the dialog control automaton.

The classes `JGraph`, `JNode` and `JEdge` serve as coordination points for the view and

[2] see also [13, p. 20]

controller parts of the Model/View/Controller architecture. They correspond directly to the model interfaces `Graph`, `Node` and `Edge`.

The generated main editor class, represented in figure 5.6 by an exemplary class `generatedEditor`, connects instances of problem domain classes implementing one of the model interfaces with instances of `JNode` or `JEdge`. It also connects these `JNode` and `JEdge` instances with objects of the appropriate dialog control class for each problem domain class. Because of the delegation architecture, the interaction behavior can not only be determined at generation time but also changed later at runtime.

Figure 5.6 contains an exemplary dialog class `generatedDialog` to be connected with a `JNode` instance. This `JNode` instance will delegate its abstract interaction interface, i.e. the `getState` and `processToken` methods, to this generated class.

Section 6.1 discusses how the generated automaton and interactor classes are implemented.

5.4 Presentation

This section describes the View part of the Model/View/Controller design pattern together with the low level controller parts that depend heavily on presentation properties such as the shape of a figure.

The specification language uses a set of graphics primitives, called figures in section 4.3.1. As for problem domain and dialog control, the software architecture of the generated graph editors contains a set of interfaces and abstract classes that represent the specification language concept. Figures in the presentation specification are represented by classes implementing the `Figure` interface located in package `grace.figures`.

In section 4.2.5 we defined abstract interaction objects as consisting of a presentation and a transformation of a sequence of physical input events into logical tokens. Event translation is provided in this archicecture by generated classes implementing `Interactor`. Low level presentation is provided by existing classes implementing `Figure`. Actions in interactors may modify properties of associated figures to affect the presentation according to their current state. So we have, that a figure connected with an interactor forms an abstract interaction object.

In figure 5.7 we see three kinds of abstract interaction objects. They are composed of `EditorFigure` and `EditorInteractor`, of `EdgeFigure` and `EdgeInteractor`, and of `NodeFigure` and `NodeInteractor`.

Again, the coordination classes `JGraph`, `JNode`, and `JEdge` use delegation to provide a particular service; this time presentation. Differently to abstract dialog control, the

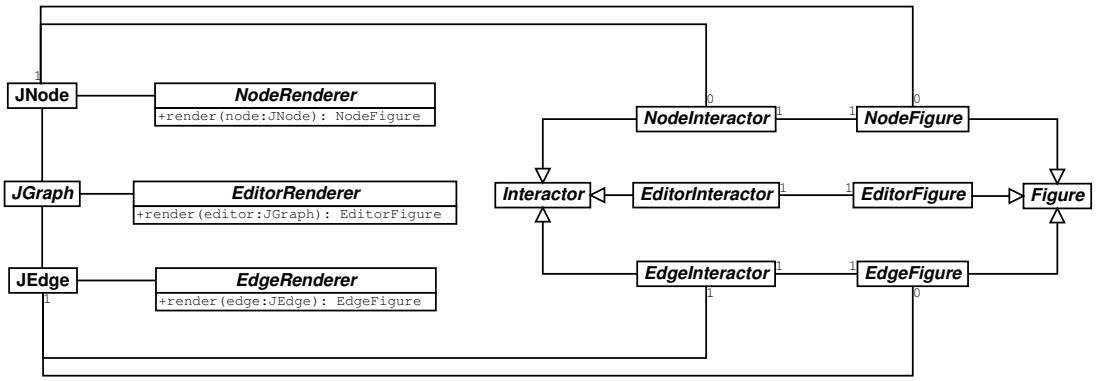


Figure 5.7: The renderer concept

classes now explicitly exploit the ability of the delegation technique to change a particular behavior at runtime.

Since it is anticipated by specification language concepts, that each control state of the abstract dialog control automaton has a different presentation, i.e. a different figure, the presentation is recalculated at each control state and application state change[3].

In analogy to the renderer concept of Java Swing, used to describe how model properties are presented in a view, we use three kinds of renderers in the graph editor. `EditorRenderer`, `NodeRenderer` and `EdgeRenderer` define their interface.

Again using delegation, generated classes implement these renderer interfaces and are connected with particular editor, edge, and node view instances by the generated editor main class. Renderers directly correspond to and implement the presentation mappings of the graph editor specifications defined in section 4.3.2. They map abstract dialog control states to figures and they connect these figures with the appropriate generated interactors.

Figure 5.8 takes a closer look at the `Figure` interface and at how the different kinds of figures are related. The `Figure` interface contains among others three important methods:

`getShape` is expected to return the `java.awt.Shape` that describes how the figure is presented to the user on screen. `contains` is a service for interactors to determine whether a particular point is located inside the screen representation of the figure or not. `paint` is the main delegate method used by the view coordination classes to create the actual screen representation.

[3] see section 5.6 on how these changes are propagated

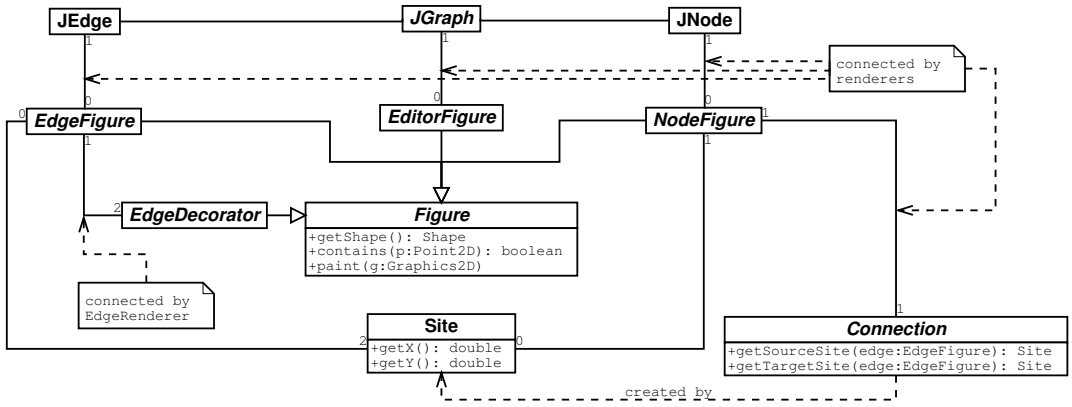


Figure 5.8: Figure relationships

All methods of `Figure` are expected to use model properties, i.e. the application problem domain, to influence and determine the screen representation. Since there are not only application problem domain specific attributes that can influence the appearance of an user interface component but also user interface specific properties like color preferences etc. the view coordination classes `JGraph`, `JNode`, and `JEdge` support a set of named properties, that figures may take into account when creating a screen representation.

The property mechanism uses two methods `void setProperty(String name, Object value)` and `Object getProperty(String name)` to store arbitrary property values under certain standardized names. Please see section A.2.4 for a list of properties supported by the generator prototype *grace*. Properties can be set and modified from inside the graph editor specification, from the rest of the user interface as well as directly from within the application. The `setProperty(GraphObject object, String name, Object value)` and `getProperty` methods of `JGraph` provide convenient access for application functions to properties that are mapped automatically onto the respective view instances.

In section 4.3.2 we mentioned the concepts of edge decorators and connection algorithms. They are represented by the interfaces `EdgeDecorator` and `Connection`. While `EdgeDecorator` is part of the `Figure` hierarchy, and therefore is itself a `Figure`, the interface `Connection` is not a figure but only associated with one.

Generated classes implementing `Connection` are associated with concrete `NodeFigure` instances by `NodeRenderers`. Depending on whether an edge leads into or out of the node, they provide for each edge connected with a node a source and target `Site` respectively. A `Site` stores a screen location. Using the `Connection` interface of its source

and target node, each `EdgeFigure` has access to exactly two sites: the end points of the edge. At these two locations also the edge decorators are painted on screen.

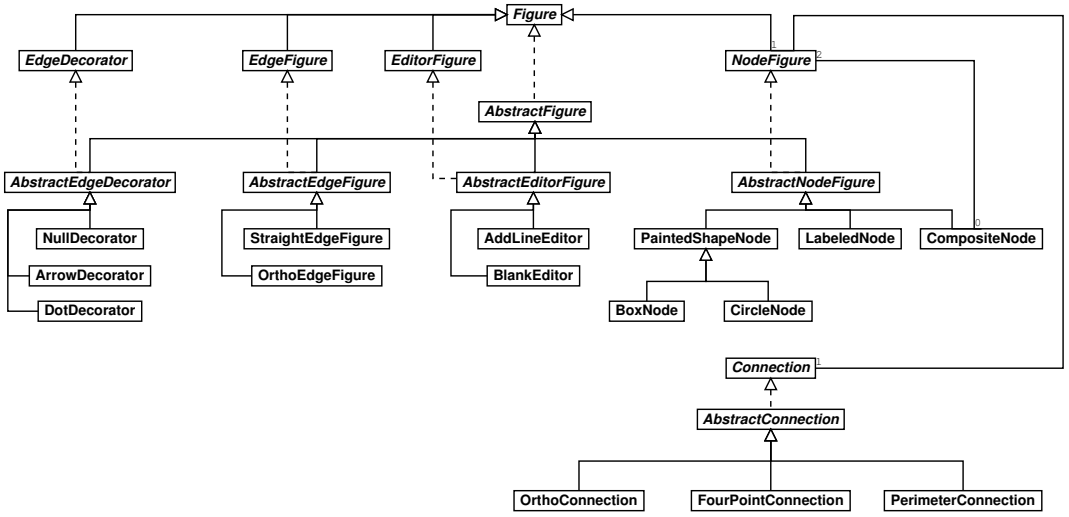


Figure 5.9: Figure class and interface hierarchy

Figure 5.9 while neither exhaustive in the number of classes nor in the relationships between them gives an overview of the interface and class hierarchy induced by `Figure`. It contains on the one hand the already described hierarchy of `Figure` interfaces, and on the other hand a hierarchy of abstract base classes starting with `AbstractFigure` that provide default implementations of the `Figure` hierarchy interfaces.

This architecture provides the possibility to extend the existing set of figures with very little additional overhead while it also permits a completely independent class to be used in the specification language. The generator only uses and expects the particular `Figure` interfaces to be implemented. So it is convenient but not necessary to extend the figure set by subclassing either an existing concrete figure or one of the abstract base classes.

5.5 Layout Algorithm interface

The interface to third party automatic layout algorithms is intentionally kept very small.

The Java interface `grace.layout.GraphLayout` that an automatic graph layout algorithm is expected to implement, demands only one method `void layoutGraph(Graph model, JGraph view)`. It is expected to use the `setBounds` methods of `JGraph` to set or change the geometric properties of the model component's representations.

While there is only a minimal API to be implemented by the third party module, it can and is expected to use the relatively rich interface of the view classes `JGraph`, `JEdge`, and `JNode`. Their interface is described in full detail in the API class documentation on disk.

The layout algorithm can be connected with an editor instance by using the method `setGraphLayout` of `JGraph`. The view is then laid out whenever the `layoutGraph` method of `JGraph` is executed again delegating the actual work to the layout algorithm.

This method is not called automatically, but has to be invoked explicitly for instance in the dialog control specification, because most layout algorithms are either time consuming or may lead to large changes in diagram appearance irritating most users or both. It is also possible to explicitly invoke a new diagram layout from outside the graph editor or to use the listener concept of section 5.6 for an automatic relayout at particular application or editor related events.

5.6 Integration

In the sections above we surveyed, how the specification language concepts can be represented in a flexible and extendable way. We touched the topic of application integration at several points, as in the ways the problem domain implementation can be connected to the editor, the ways application properties can influence figure screen representations, how the application user interface can directly modify view properties like color and geometry, and how automatic layout is initiated.

We did not yet cover, how application problem domain, application user interface, and graph editor remain consistent and how changes are propagated between them.

For this purpose the graph editor design uses specialized versions of the observer design pattern presented in e.g. [13]. Gamma et al give the following motivation for the observer pattern: *A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability* [13, p. 293].

One of the liabilities of the observer pattern is, that observers are only notified *that* a change took place. Without further protocol, they are not notified of *what* changed. Therefore we incorporate this necessary further protocol into the observer pattern and arrive at a listener concept that is also used in the Java AWT.

Figure 5.10 demonstrates the listener concept for propagating changes in the application problem domain. The graph listener architecture consists of the `GraphListener` interface for objects to be notified of changes, the `GraphEvent` class for storing information about

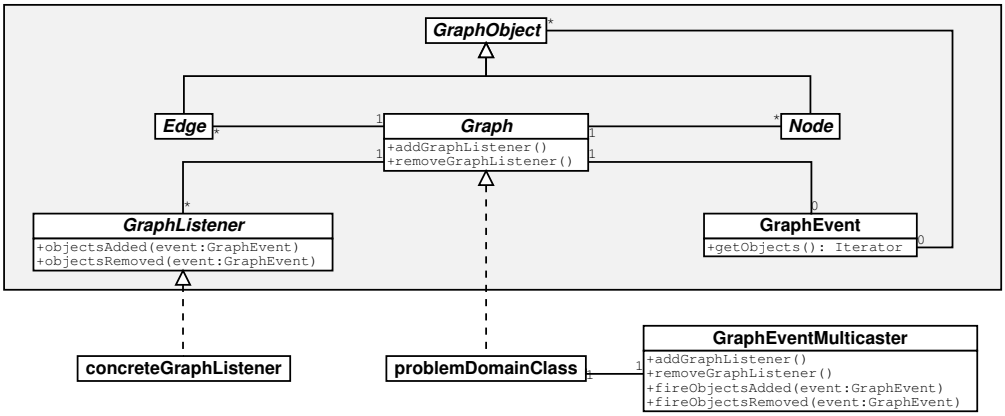


Figure 5.10: Graph event listeners

the change, the event source, in this case an instance of the `Graph` interface, and a convenience class `GraphEventMulticaster`.

`GraphListener` takes the role of the observer in the observer pattern, but instead of only providing one method `update`, it provides specialized methods `objectsAdded` and `objectsRemoved` that are expected to be called when objects are added to or removed from the graph respectively. The `Graph` interface plays the role of the observable in the observer pattern. It provides methods to add and remove graph listeners to be notified of subsequent changes. Additionally to the observer pattern, the update methods of `GraphListeners` have a parameter `GraphEvent` containing further information about the propagated change – in this case which objects were added or removed.

Because management of listeners is a recurring task for each problem domain class or problem domain adapter, this work can be delegated to methods of class `GraphEvent-Multicaster`. Additionally to the event methods of `Graph`, it provides two methods `fireObjectsAdded` and `fireObjectsRemoved` that propagate a particular `GraphEvent` to all listeners and that call the appropriate listener method.

The concrete listener – in our case e.g. the generated editor – attaches itself to the event source and can then react to changes in the graph structure without the graph structure itself depending on any further properties of the listener.

Graph event listeners are used by the editor to keep it consistent with the current state of the application problem domain.

The rest of the user interface controls the editor by sending logical tokens to one or more of the abstract dialog control automata of the editor, and by directly modifying view properties as described in section 5.4. To notify the rest of the application

user interface of state changes in the editor, another instance of the listener template, the `StateListener`, may be used. `StateListener` can be connected to any `Automaton` event source. Similarly the standard classes `PropertyChangeEvent` and `PropertyChangeListener` of package `java.beans` may be used to keep track of individual view properties. We refer the interested reader to the API class documentation of the package `grace.editor` and `java.beans` for further information on these listeners.

Because the generated editors are subclasses of `javax.swing.JPanel`, and because of the listener concept known from the Java AWT, a seamless integration of the visual and interactive properties of the editor with the application user interface is possible.

5.7 Persistence

Section 2.4 requires the diagrams created with the graph editor to have the ability to be made persistent.

As we already observed in section 2.4 there are multiple choices of which parts of the diagram are to be made persistence.

For some applications only the logical structure of the diagram is to be stored. In our software architecture this is the model, i.e. the application domain classes. This technique is usually combined with an automatic layout algorithm that is able to recreate the appearance of the diagram from the logical structure of the graph domain only. Upon retrieval of the problem domain classes, a new generated editor can be created and attached to the retrieved model. After automatic layout diagram appearance is the same as before committing it to storage.

In other cases not only the logical structure, but also diagram appearance including graph object positions, colors, etc. has to be made persistent, while more transient information as the current dialog state is lost. Especially if no automatic layout algorithm is used, this is the most common form of storing diagrams. This is also the form that the *grace* software architecture supports by default.

Diagram appearance in *grace* editors is determined by the model, by classes generated from the specification, and by figure properties that have been set by application or editor. The code of the generated classes does not have to be stored for each diagram. It remains fixed for each editor specification. The information that has to be stored to recreate a diagram is the value of all properties and the information which of these properties are associated with the view of which model instances.

One possibility to achieve this in a way that is comfortable for the editor developer is to use Java serialization. All classes of the graph editor including all generated classes support serialization. For the diagram to be made persistent the model classes and all

property values have to support serialization, too. All standard *grace* figures allow to only use properties that comply with this condition. So for the editor developer there is only the responsibility left to make sure that the model is serializable. In most cases this only means to let the application problem domain classes implement the empty interface `java.io.Serializable`.

This is the standard way to make a diagram of a *grace* editor persistent.

If it is not possible to use Java serialization for some reason, or if full control over all detail of what is to be stored is desired, there is also another possibility. We established above that all that is needed to recreate a diagram are the property values and the association of model instances with view instances containing the right property values. It is possible, to use the `getProperty` methodes of `JNode`, `JEdge`, and `JGraph` to retrieve all these values and store them in a custom data format together with custom information about the model. Upon retrieval for each model instance a new view instance can be created and the properties be restored by using the various `setProperty` methods.

This method is not as comfortable as serialization but offers more flexibility in return.

The flexibility includes also the freedom to store more than property values only. It could for instance be used to also store and later retrieve the current dialog state or other custom data that may be important for specific applications.

5.8 Conclusion

We have presented a flexible and extendable software architecture for graph editors. We claim that it reflects the specification language concepts of section 4 and that it enables generated graph editors to satisfy the requirements of sections 2.4 and 2.5.

The specification language concepts are:

Problem Domain The notion of a graph structure, edges, and nodes and their direct mapping to application domain classes are directly represented by the respective interfaces. A technique for using existing problem domains without change has been introduced.

Dialog Control The concepts of abstract and concrete dialog control automata are directly represented by `Automaton` and `Interactor`.

Presentation The figure concept, and the mapping from dialog control state and application state to presentation are directly represented by `Figure` and the renderer interfaces.

Styles The mapping of application domain classes to abstract dialog control automata and presentations is directly represented by the generated main editor class.

Functional and non functional requirements of sections 2.4 and 2.5:

Component The generated editors are subclasses of `javax.swing.JPanel` and may be used in the application user interface like any other component.

Integration Figure properties and the listener concept presented in section 5.6 enable a seamless integration of visual and interactive user interface properties.

Interaction The possibility to place arbitrary code into the action part of dialog specifications and to directly reference the application problem domain classes enables all four interaction classes required in section 2.4.

Persistence Persistence is achieved by Java serialization or the figure property mechanism.

Layout The architecture provides an easy to use interface to third party layout algorithms.

Restrictions Interaction restrictions may be used as preconditions in dialog specifications and are therefore inherently supported by the editor architecture.

Platforms The implementation language Java is widely accepted as being platform independent.

Response Time Measurements on an otherwise idle Sun Ultra 60 workstation for the example in appendix B lie consistently well below the specified limit.

Part III

Implementation

6 The Generator

This chapter briefly describes how some of the specification language concepts can be implemented, and how one can arrive at these implementations automatically. We will also take a look at the parallels that exist between a graph editor generator and a programming language compiler.

6.1 Generating Dialog Control

The graph editor specification language uses dialog control automata as a concept to define and specify dialog control. Dialog control automata are very similar to finite automata and the implementation of finite automata is both relatively simple and well known from formal languages or e.g. lexical analysis in compiler construction.

The task is to arrive from a DCA description at a piece of Java code that has the same semantics. We will briefly describe a method to automatically implement a DCA and we will also identify which pieces of Java code correspond to which DCA concepts. Since the necessary amount of formalization does not result in much further insight, we will not give a formal proof of semantical equivalence and of correctness of the method. The method should however be simple enough to be trusted without formal proof.

6.1.1 Generating Abstract Dialog Control Automata

Implementations of abstract dialog control automata are expected by the graph editor architecture to implement the interface `Automaton`.

From finite automata in lexical analysis and implementations of state transition systems as e.g. in [19], we arrive at the following basic structure:

- The current control state of the automaton is encoded as an integer value.
- Possible control states of the automaton are implemented as an enumeration of integer constants.

- A transition is fired by checking if it is applicable, executing the associated action code and by changing the current control state to the target state of the transition.

The implementation of the `int getState()` method of `Automaton` is therefore simple and fixed. It returns the integer value encoding the current control state.

The implementation of the `void processToken(Token token)` method consists of finding the applicable transition and firing it. Section 4.2.3 requires the automaton to be deterministic. Because the specification may not by itself describe a deterministic automaton, the implementation has to reflect a kind of priority assignment as described in sections 4.2.3 and 4.2.4.

If we encode the tokens also as integer values stored in the `kind` field of class `Token`, we can for simple cases without automaton inheritance use a method very similar to the one presented in [19]. For a specification like

```
automaton A {

  states state_1, .., state_n;
  tokens token_1_1, ..., token_n_k;

  >state_1:
    (token_1_1; prec_1_1) -> target_state_1_1 do { action_1_1; }
    ..
    (token_1_m; prec_1_m) -> target_state_1_m ..

  ..

  state_n:
    (token_n_1; prec_n_1) -> target_state_n_1
    ..
    (token_n_k; prec_n_k) -> target_state_n_k
}
```

we would arrive at a implementation looking like

```
public void processToken(Token token) {

  switch (state) {

  case state_1:
    if (token.kind == token_1_1 && prec_1_1) {
      action_1_1;
      state = target_state_1_1;
```

```
      break;
    }

    ..

    if (token.kind == token_1_m && prec_1_m) {
      action_1_m;
      state = target_state_1_m;
      break;
    }

    break;

  ..

  case state_n:
    if (token.kind == token_n_1 && prec_n_1) {
      action_n_1;
      state = target_state_n_1;
      break;
    }

    ..

    if (token.kind == token_n_k && prec_n_k) {
      action_n_k;
      state = target_state_n_k;
      break;
    }

    break;
  }
}
```

In each `case state_i` branch, all transitions with start state `state_i` are listed. A transition on the implementation side is represented by an `if` statement testing for the input token to match and the precondition to be true. If both comparisons yield true, the action code is executed and the control state is changed to the appropriate target state.

We claim that, when program execution reaches any `action_x_y` code, the transition containing that piece of action code is the one that is applicable.

It is obvious, that the current control state must match the start state of the transition, that the input token must match the token of the transition, and that the precondition of

the transition must yield true to arrive at the action code part. It remains the question if, when the specification was non deterministic, the priority assignment has been satisfied. If the order of the `if` statements matches the order of the transitions in the specification, the uppermost `if` corresponding to the first applicable transition is executed first. It is the only transition that is fired, because each `if` statement closes with a `break`.

We also claim, that if a transition is applicable, it is fired.

If a transition is applicable, its source state `state_i` matches the current control state. Since it is listed in the `switch` statement in branch `case state_i`, the `if` statement of the transition will be reached eventually if no other transition is applicable. In an implementation resulting from a deterministic specification this is obvious. For a non deterministic specification, the uppermost applicable transition is listed first in the implementation. By the priority assignment this is to be treated as the only applicable transition.

If no transition is applicable, dialog control state and application state do not change as required in section 4.2.3.

If the control state is initialized with the state marked as initial in the specification, we have that the semantics of the automaton implementation exactly match the semantics of the DCA specification. [1]

Automata constructed using inheritance can be generated in almost the same way. We keep the implementation method as it is, but convert the DCA with inheritance to a DCA without inheritance first.

The set of states is then the union of all sets of states of the ancestor automata as described in section 4.2.4. If DCA B extends DCA A, we add all transitions of A starting in state s to the state s of DCA B below all other transitions of B starting in s. If A is a DCA using inheritance we use the same method recursively on A first. If all transitions of A have been added to B, the new B without inheritance describes the same DCA as the old B with inheritance.

Example 6.1
The specification

```
automaton A {
states a;
tokens x;
  >a: (x) do { action_1; }
}
```

[1] without a formal definition of the semantics of a DCA, the formal semantics of Java and a formal definition of the term “matches” this statement necessarily is not very satisfying. We rely on the kind interpretation of the reader.

```
automaton B extends A {
states b;
  >a: (x; some_cond) -> b do { action_2; }
  b: (x) -> a;
}
```

is converted into an implementation

```
public void processToken(Token token) {
  switch (state) {
  case a:
    if (token.kind == x && some_cond) {
      action_2;
      state = b;
      break;
    }
    if (token.kind == x) {
      action_1;
      break;
    }
    break;
  case b:
    if (token.kind == x) {
      state = a;
      break;
    }
    break;
  }
}
```

The method for eliminating inheritance presented above yields exactly the priority assignment required in section 4.2.4.

6.1.2 Generating Interactors

From an abstract point of view, interactors and abstract dialog control automata are the same thing. Therefore we can in principle use the same method to implement them.

The differences are merely technical. Interactors implement one or more event listeners of the package `java.awt.event` to be supplied with input, leading to the following consequences:

1. Event listeners have to be attached to a `java.awt.Component` to be supplied with input. We solve this by attaching them to the main editor class that is derived from `javax.swing.JPanel`.

2. It must be determined from the specification, which Java event listener interfaces have to be implemented.

3. Each event listener interface method has to be present in the implementation, even if a particular event from this listener is not used in the specification.

4. Requirements of section 3.1 demand, that it should be easy to add new low level input events, i.e. new AWT event listeners to the specification language.

5. Since Java event listeners are callbacks ordered by input event, transitions have to be mapped onto these callback methods.

Items 2, 3 and 4 are solved by a configuration file that includes a mapping from event name used in the specification to the AWT event listener providing that event. The configuration file also includes which method each event listener interface contains.

Item 5 is solved by partitioning the one large `switch` statement presented above into several small `switch` statements: one in each event listener method. Each event listener method contains all those transitions that are triggered by the corresponding input event. Otherwise the method presented in section 6.1.1 above is used without change.

Example 6.2
The following specification from example 4.5

```
interactor ClickInteractor {
  states start;

  >start:
    (clicked) sends Selectable.select to editor
}

interactor DragInteractor extends ClickInteractor {
  states dragging;

  classcode {
    private int x, y, dx, dy;

    private void update(MouseEvent e) {
      dx = e.getX()-x;  x = e.getX();
      dy = e.getY()-y;  y = e.getY();
```

```
    }
  }

 >start:
    (pressed) -> dragging do { update(event); }

  dragging:
    +(dragged) do { update(event); parent.translate(dx,dy); }
    +(released) -> start
}
```

results in the following implementation (only relevant pieces are listed)

```
public class DragInteractor implements Interactor, MouseMotionListener, MouseListener {

  final public static int dragging = 0;
  final public static int start = 1;
  ...
  private int x, y, dx, dy;

  private void update(MouseEvent e) {
    dx = e.getX()-x;  x = e.getX();
    dy = e.getY()-y;  y = e.getY();
  }

  public DragInteractor(JGraphObject parent) {
    this.state = start;
    setParent(parent);
  }

  ...

  public int getState() {
    return state;
  }

  public void mouseDragged(MouseEvent event) {
    switch (state) {
    case dragging:
      if (true) {
        update(event); parent.translate(dx,dy);
        break;
      }
      break;
    }
  }

  public void mouseClicked(MouseEvent event) {
```

```
    switch (state) {
    case start:
      if (contains(event)) {
        editor.processToken(new Token(Selectable.select,parent));
        break;
      }
      break;
    }
  }

  public void mousePressed(MouseEvent event) {
    switch (state) {
    case start:
      if (contains(event)) {
        update(event);
        state = dragging;
        break;
      }
      break;
    }
  }

  public void mouseReleased(MouseEvent event) {
    switch (state) {
    case dragging:
      if (true) {
        state = start;
        break;
      }
      break;
    }
  }

  public void mouseMoved(MouseEvent event) { }
  public void mouseEntered(MouseEvent event) {  }
  public void mouseExited(MouseEvent event) {  }
}
```

6.2 Generating Presentation and Styles

The presentation of the editor and graph objects are represented by the renderer interfaces in the software architecture. In the specification language, the presentation is mainly a mapping from control state to figures.

This mapping is very simple to implement and we will not bore the reader with further source code, but will instead briefly list the basic tasks for each kind of renderer:

- Node renderers have according to the current dialog control state to construct the appropriate figure instance for the node, to construct an interactor, to construct an instance of a connection algorithm, and to connect these three objects.
- Edge renderers have according to the current dialog control state to construct a figure for the edge, to construct an interactor, to construct a decorator for source and target end of the edge resp., and to connect these objects.
- Editor renderers just construct the appropriate figure and interactor instance for the current dialog control state and connect both.

On the technical side, it is important for performance to avoid the construction of new objects, in this case figures, as far as possible. If for instance the automaton control state did not change between two calls of the `render` method, the figure already present on screen should be reused.

Inheritance for renderers can be resolved by including all mappings from the direct ancestor that are not overwritten in the derived presentation. Again, if the ancestor also uses inheritance, the method recursively resolves the ancestor first.

Styles in the specification are mappings from problem domain classes to presentation and abstract dialog control automata. On the implementation side, this corresponds to creating a view instance for each problem domain object and to connect this view instance with the appropriate generated automata and renderers.

New view instances are only created, when a new editor instance is created or when the model, i.e. the graph structure changes. We can therefore implement a style in the `add` methods of the generated graph editor.

When a new object is added to the graph, and therefore `add` of the graph editor is called subsequently, we can determine the runtime type of the object using the Java `instanceof` operator. A series of `if` statement in the correct order will find the most special runtime type of the object that was provided in the specification.

Together with the same method to resolve inheritance that we also used for renderers, we arrive at an implementation that satisfies the requirements of the specification language described in section 4.4.

6.3 Conclusion

We have given a brief overview of how to automatically arrive at an implementation of a graph editor specification.

Using the software architecture of chapter 5, we have direct implementations of abstract dialog control automata, of event level interactors, of presentation specifications, and of style specifications.

7 A Prototype

This chapter introduces the graph editor generator prototype *grace*.

7.1 Parallels to Compiler Construction

Being a generator *grace* is very similar to a compiler. *grace* transforms an human readable specification into an efficient implementation. A compiler transforms a program in a higher level programming language into efficient machine code.

The phases in generating a graph editor from a specification have direct parallels to compiler construction.

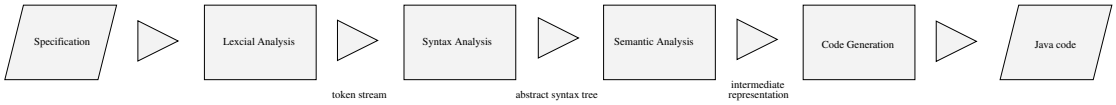


Figure 7.1: Phases of graph editor generation

Figure 7.1 presents the phases of graph editor generation with *grace*. As it is the case in an usual compiler, *grace* reads the input by first performing lexical and syntactical analysis resulting in an abstract syntax tree. During semantic analysis of this abstract syntax tree, the generator performs basic type checks, tests for some consistency properties of the specification, and transforms the abstract syntax tree into an intermediate representation. This intermediate representation is then transformed into the Java source code of a graph editor.

As in a compiler, the abstract syntax tree and the intermediate representation serve to decouple the front end and back end of the generator from the actual transformation. This enables the specification language and the target language to be changed independently of one another.

The lexical and syntactical analysis components of *grace* are themselves generated automatically from a specification by the tool *pgen* included in the *grace* distribution. *pgen* is

an integration tool for the lexical analyzer generator *JFlex* [24] and the parser generator *CUP* [23]. Classes storing the abstract syntax tree are generated by *classgen*[1].

Semantical analysis in *grace* is partially tool supported by the integrated visitor design pattern [13, p. 331] support of *classgen*. There is not yet a generator the author knows of for automatic attribute evaluation of attribute grammars in Java.

7.2 Implemented Features and further research

The *grace* prototype implements the full spectrum of the specification language as presented in this thesis.

Obvious extensions to this first prototype are of course a larger set of graphical figures and a more comprehensive library of graph editor specification modules for different well designed interaction and presentation styles. The generator is explicitly designed to accommodate both these kinds of extensions.

Further research for graph editor generation could include the following items:

- The specification language does at the moment only have a purely textual representation. A graphical representation of the same concepts could be more intuitive and easier to use. Especially dialog control automata can be described very intuitively by state transition diagrams.
- A graphical representation of graph editor specifications could be supported by an integrated development environment for graph editors. The IDE could for instance include graphical editors for all specification modules, browsing support for specification libraries, an integrated generator, etc. This interface builder for graph editors would itself contain editors for graph-like data structures, e.g. for dialog control. They can themselves be generated by *grace*.
- If taken one step further, the graph editor IDE could also include extensive debugging support for graph editors, such as step by step simulation of specifications and simultaneous execution of the specification and the generated editor. In this way one could watch a dialog evolve in the abstract graphical dialog specification, while it is at the same time executed in the real, maybe even already application integrated editor.
- In the current version of *grace*, new figures have to be coded in Java when the set of existing figures is not sufficient. While this is not necessarily a complicated task,

[1] see also [5] for further information about *classgen*

because the relatively rich Java 2D API can directly be used and existing figures can be extended by inheritance, it would nevertheless be nice to have a graphical editor, that allows the creation of new figure shapes in a drawing tool like fashion. This editor could be another part of a graph editor IDE.

The items above concentrated on the front end of the generator and on the usability of the specification language. There is also opportunity for further research in enhancing the expressive power of the language.

- Related work described in section 1.3 includes specification techniques for the syntactical correctness or consistency of the problem domain structure. It would be interesting to investigate how these specification techniques can be integrated as another module into the current graph editor specification language. The approach of DiaGen [43] for instance combined with the methods for extendable specifications presented here could prove fruitful.

- A specification technique of static and incremental graph layout algorithms would be a valuable addition to the specification language. Further research is needed on how e.g. constraint based techniques like the work of Chok and Marriott [9] can be directly used for that purpose and be integrated into a graph editor generator like *grace*.

- In the current version, *grace* performs only very basic checks, whether a given graph editor specification is consistent. Consistent can in this case mean many things. Consistency properties can be very general such as that every dialog control state should be mentioned in the associated presentation mapping, or they can also be specialized for a certain class of graph editors such as a basic completeness condition, that e.g. all interactors have to react to a certain input event. AutoFocus includes an interesting approach in [21], that lets the specification developer define, when a specification is to be called consistent. Such basic, developer defined first order predicates over graph editor specification properties could be integrated into the semantical analysis of the generator.

- Taking the idea above one step further, graph editor specifications could be checked for dialog properties specified in temporal logic. For that to be possible, the precondition and action part of dialog control transitions would have to be restricted, so that temporal dialog properties can automatically be proven by a model checker.

8 Conclusion

In this thesis we have discussed the problem of rapidly developing graph editors to be integrated into a wide range of applications.

We collected, structured, and analyzed classes of requirements for integrated graph editors.

Based on these requirements, we have developed a specification language, that allows to specify graph editors conveniently.

The graph editor specification language provides a clear distinction between problem domain, dialog control and presentation. Problem domain, dialog control, presentation and style specifications use an inheritance concept to provide modular, reusable and extendable descriptions.

The concept of styles allows the construction of graph editor specification libraries to be used for rapid prototyping on new problem domains. Specifications can be changed and extended step by step in the development process at each desired level of detail.

Compared to a full natural language description or a complete implementation, our graph editor specifications are compact, extendable, reusable, modular, easy to use, easy to change, easy to read, and easy to communicate.

Together with a flexible and extendable software architecture for graph editors that reflects the concepts of the specification language, we have developed the graph editor generator prototype *grace*.

grace transforms a graph editor specification into a full Java implementation of the editor described. The generated graph editor can be seamlessly integrated into the application user interface.

The main goal of this diploma thesis was to reduce the development and maintenance cost for graph editors.

We claim to have reached this goal.

Part IV

Appendix

A Grace

A.1 Specification Language Overview

A.1.1 Syntax

The following EBNF grammar describes the syntax of a *grace* specification file.

```
specification      ::= [package] useSpec* [editorStyle] module*

package            ::= "package" qualIdent ";"
editorStyle        ::= "editor" "style" ident ";"
useSpec            ::= "uses" qualIdent ";"

module             ::= graphModule
                     | automatonModule
                     | interactorModule
                     | presentationModule
                     | styleModule

graphModule        ::= "graph" ident "(" graphMapping* ")"
graphMapping       ::= nodeMapping | edgeMapping
nodeMapping        ::= "nodes" qualIdent* ";"
edgeMapping        ::= "edges" qualIdent* ";"

automatonModule    ::= "automaton" ident [extends] "{" automatonSpec* "}"

automatonSpec      ::= stateDecl | tokenDecl | import_statement |
                       classCode | stateDef

stateDecl          ::= "states" ident* ";"
tokenDecl          ::= "tokens" ident* ";"
classCode          ::= "classcode" block

stateDef           ::= [">"] ident ":" transition*
transition         ::= "(" ident [";" conditional_expression] ")"
                         ["->" ident] ["do" block]

extends            ::= "extends" qualIdent
qualIdent          ::= ident ("." ident)*
```

```
interactorModule   ::= "interactor" ident [extends] "{" interactorSpec* "}"

interactorSpec     ::= stateDecl | classCode | import_statement |
                       interactorReq | interactorState

interactorReq      ::= "requires" ("node" [ident]|"edge" [ident]|"editor") ";"

interactorState    ::= [">"] ident ":" interactorTrans*
interactorTrans    ::= ["+"] "(" ident [";" conditional_expression] ")"
                       ["-> ident] [send] ["do" block]

send               ::= "sends" qualIdent ["to" ("editor" | qualIdent)]


presentationModule ::= nodePresentation | edgePresentation | editorPresentation

nodePresentation   ::= "node" "presentation" ident [extends] "{" nodeSpec* "}"
edgePresentation   ::= "edge" "presentation" ident [extends] "{" edgeSpec* "}"
editorPresentation ::= "editor" "presentation" ident [extends] "{" editor* "}"

nodeSpec           ::= nodeState   | presentationReq
edgeSpec           ::= edgeState   | presentationReq
editor             ::= editorState | presentationReq

presentationReq    ::= "requires" ("automaton"|"type") qualIdent ";"

nodeState          ::= ident ":" "(" qualIdent ["(" parameters ")"] ","
                         qualIdent "," qualIdent ")"
edgeState          ::= ident ":" "(  qualIdent ["(" parameters ")"] ","
                         qualIdent "," qualIdent "," qualIdent ")"
editorState        ::= ident ":" "(" qualIdent ["(" parameters ")"] ","
                         qualIdent ")"

parameters         ::= parameter ("," parameter)*
parameter          ::= qualIdent ["(" parameters ")"]


styleModule        ::= "style" ident [extends] "{" mapping* "}"
mapping            ::= qualIdent ":" "(" ident "," ident ")"
```

A.1.2 Semantics

This section gives a brief and informal overview of the semantics of the *grace* specification language.

A graph editor is described by a set of specifications, where each specification is located in its own file with extension `".spec"`.

A specification file consists of an optional package declaration, a list of specification file references, an optional main editor style declaration, and a list of modules.

```
specification ::= [package] useSpec* [editorStyle] module*
```

A package declaration `"package" qualIdent ";"` causes the generated code to be placed in the Java package denoted by `qualIdent`.

A specification file reference `useSpec ::= "uses" qualIdent ";"` causes the contents of the specification file denoted by `qualIdent` to be read and included in the set of specifications. *grace* follows the same naming conventions as the Java package mechanism. A `use grace.presentations` will import the file `grace/presentations.spec` applying the correct platform dependent path conventions. *grace* will read the first matching file in the search path[1].

A main editor style declaration causes the specified style to be used for the generated graph editor. There must be exactly one main editor style declaration in the set of specification files.

A specification module is one of the modules described in chapter 4. The following sections provide a brief overview for each.

Graph Modules

Graph modules define the problem domain of graph editors as described in section 4.1.

```
graphModule ::= "graph" ident "(" graphMapping* ")"
```

It has a name, denoted by `ident`, and contains a list of node and edge mappings.

The graph module identifies the main graph structure of the problem domain as an application class with the same name `ident` as the graph module, implementing the interface `grace.model.Graph`, and the partitions of the graph. Each application class listed in an edge or node mapping, implementing `grace.model.Edge` or `grace.model.node` respectively forms a partition class of the graph structure.

[1] see also section B on how to modify the search path

Automaton Modules

An automaton module defines an abstract dialog control automaton as described in section 4.2.2.

```
automatonModule ::= "automaton" ident [extends] "{" automatonSpec* "}"
```

Automaton modules have a name denoted by `ident`, can extend[2]other automata described in the specification and contain a list of automaton specification parts defining the content of the automaton.

```
automatonSpec   ::= stateDecl | tokenDecl | import_statement |
                    classCode | stateDef
```

An automaton specification part is a state or token declaration, an import statement, code to be included verbatim into the generated automaton or a state definition.

State and token declarations are lists of identifiers defining the set of states and tokens of the automaton. The state name `all` is reserved.

Import statements are Java import statements to be included in the generated automaton and to be used in the class code parts of the automaton or in the action code of transitions.

If multiple class code pieces are present in a specification, all of them are included in the generated automaton.

A state definition

```
stateDef ::= [">"] ident ":" transition*
```

consists of an optional indicator whether the state is initial, the name of the state, and a list of transitions originating in that state. There must be exactly one initial state in each automaton module. If more than one state definition with the same state name is present in an automaton module, the transitions of all these state definitions are included in the DCA described by the module. If the name of the state is `all`, the transitions listed under this state originate in all states of the automaton and in all states of derived automata. The priority of these transitions is lower than the priority of all other transitions, so that they can be used to define default behavior. Among the transitions originating in the pseudo state `all`, the same priority assignment as usual transitions applies[3].

A transition description

[2]The current version of *grace* supports single inheritance only. See section 4.2.4 for more information about automaton inheritance

[3]see also section 4.2.3 for transition priority assignments

```
transition ::= "(" ident [";" conditional_expression] ")"
               ["->" ident] ["do" block]
```

consists of an input token, an optional precondition, an optional target state, and optional action code. If there is no precondition specified, it defaults to `true`. If there is no target state specified, it defaults to the state the transition originates in. If no action code is specified, no action is executed.

Action code of a transition may use the following implicitly declared fields:

`int state` the current automaton control state

`JGraph editor` the graph editor instance of the graph object using the automaton.

`Token token` the abstract input token currently being processed.

Interactor Modules

An interactor module defines a low level dialog control automaton as described in section 4.2.5.

Since they are very similar to automata described above, we will only list the differences.

An interactor specification part

```
interactorSpec ::= stateDecl | classCode | import_statement |
                   interactorReq | interactorState
```

can be a state declaration, a piece of class code, or an import statement as with automata; it can also be an interactor requirement or an interactor state definition.

An interactor requirement statement

```
interactorReq ::= "requires" ("node"|"edge"|"editor") [ident] ";"
```

requires the figures to be associated with the interactor to be of the specified type. The statement may require it to be a class implementing `grace.figures.NodeFigure`, `grace.figures.EdgeFigure`, or `grace.figures.EditorFigure`, and optionally to be a subclass of `ident`.

There may be at most one requirement statement in each interactor module. If there is no interactor requirement in a module, the interactor is general and can be used for any class implementing `grace.figures.Figure`. If a interactor `B` extends interactor `A`, the required figure of `B` must be the same as or a subclass of the one in `A`.

An interactor transition

```
interactorTrans ::= ["+"] "(" ident [";" conditional_expression] ")"
                    ["-> ident] [send] ["do" block]
```

consists of an optional test indicator, an input event, an optional precondition, an optional target state, an optional token creation, and optional action code.

Input events are instances of subclasses of `java.awt.AWTEvent`. The relationship between event names used in the specification and Java AWT event listener methods is defined in the *grace* configuration file. The standard *grace* configuration file is located in `lib/grace.config`.

Generated interactors test by default, whether an event is not already consumed and whether it originated inside the figure associated with the interactor. The event is only processed, if both tests yield true. If the test indicator `+` is present in a transition description, these tests are skipped.

For precondition, target state, and action code, the same rules as described above for automata apply.

A token creation

```
send ::= "sends" qualIdent ["to" ("editor" | qualIdent)]
```

causes a token of kind `qualIdent` to be created and to be sent to the automaton specified in the target description. Token kind identifiers are qualified and consist of the name of the automaton the token is declared in and the name of the token itself. If no target description is present, it is sent to the graph object associated with this interactor. It the target description is `"to" "editor"`, the token is sent to the editor. If the target description is `"to" qualIdent`, it is sent to the automaton with name `qualIdent`. In this case `qualIdent` is assumed to be a field either contained in the interactor (declared by class code) or otherwise accessible to the interactor.

Preconditions and action code of transitions may use the following implicitly declared fields:

`int state` the current state of the interactor

`JGraph editor` the graph editor containing the object this interactor is associated with

`figure` the figure this interactor is associated with. If no requirement statement was present in the specification, it is of type `Figure`; otherwise it is of the specified type.

`parent` the graph object the figure of this interactor is associated with. If no requirement statement was present, it is of type `JGraphObject`; otherwise it is of type `JGraph`, `JNode` or `JEdge`.

`event` the input event currently being processed. It is of the type the associated event listener requires.

Presentation Modules

Presentation modules define presentation mappings as described in section 4.3.2.

A presentation module is a node, edge, or editor presentation module. All three of them consist of a name, may extend another presentation module of the same kind as described in section 4.3.2, and may contain a list of presentation module contents.

A presentation module content description is either a presentation requirement or a presentation state description.

A presentation requirement

```
presentationReq ::= "requires" ("automaton"|"type") qualIdent ";"
```

is an automaton requirement or a problem domain requirement. The `qualIdent` specifies which automaton or problem domain class this presentation requires. As with interactor requirements, the requirements of derived presentations must be the same as or subclasses of the requirements in the base presentation.

A presentation state description is a node, edge, or editor presentation state description.

A node presentation state description

```
nodeState   ::= ident ":" "(" qualIdent ["(" parameters ")"] ","
                qualIdent "," qualIdent ")"
```

contains (in this order) the name of the automaton control state this mapping applies to, the figure with other figures as optional parameters for screen presentation, the name of the interactor to be associated with the figure and the name of the edge connection algorithm.

An edge presentation state description

```
edgeState   ::= ident ":" "(  qualIdent ["(" parameters ")"] ","
                qualIdent "," qualIdent "," qualIdent ")"
```

contains the name of the automaton control state this mapping applies to, the figure with other figures as optional parameters for screen presentation, the name of the interactor to be associated with the figure, the name of the edge decorator figure for the source end, and the name of the edge decorator figure for the target end.

An editor presentation state description

```
editorState ::= ident ":" "(" qualIdent ["(" parameters ")"] ","
                qualIdent ")"
```

contains the name of the automaton control state this mapping applies to, the figure with other figures as optional parameters for screen presentation, and the name of the interactor to be associated with the figure.

Style Modules

Style modules define editor styles as described in section 4.4.

A style module description

```
styleModule ::= "style" ident [extends] "{" mapping* "}"
```

contains the style's name, may extend another style as described in section 4.4, and contains a list of style mappings.

A style mapping

```
mapping ::= qualIdent ":" "(" ident "," ident ")"
```

connects the problem domain class `qualIdent` with an automaton and presentation module.

A.2 Standard Library Reference

Sections A.2.1 through A.2.5 give an overview of the standard graph editor specification libraries that are included with the *grace* prototype.

The standard libraries are located in the `lib/specs/` directory of the *grace* distribution. They can be used in a custom specification by requesting the module `grace.standards`.

The figures described in this section are part of the package `grace.figures` and are contained in the file `lib/grace.jar` to be distributed with applications using *grace* editors.

A.2.1 Automata

Name	States	Description
`NullDialog`	`start`	provides no interaction
`Selectable`	`not_selected`, `selected`	basic dialog for selectable objects
`BasicEditorDialog`	`edit`, `remove`	base editor dialog, provides basic selection and removal of graph objects
`EditorDialog`	`edit`, `remove`	extends `BasicEditorDialog`, declares tokens `pressed`, `released`, and `click` to be used in derived editor dialogs and the interactors `ClickEditorInteractor` and `AddEdgeEditorInteractor`

A.2.2 Interactors

General Interactors

Name	Required Figure	Description
`NullInteractor`	none	does nothing
`ClickInteractor`	none	sends `select` tokens to the editor

Node Interactors

Name	Required Figure	Description
`DragInteractor`	node	extends `ClickInteractor` by basic node dragging behavior.
`ResizeInteractor`	`SelectedNodeFigure`	extends `DragInteractor` by resize management using the handles of `SelectedNodeFigure`.

Edge Interactors

Name	Required Figure	Description
`CurveInteractor`	`CurveControl`	extends `ClickInteractor` by control tangent handling for cubic curves.
`TextInteractor`	`TextControl`	extends `ClickInteractor` by handling of text label replacements.
`TextCurveInteractor`	`CurveControl`	extends `CurveInteractor` by handling of text label replacements. The parameter of the `CurveControl` figure needs to be a `TextControl` figure.

Editor Interactors

Name	Required Figure	Description
`BasicEditorInteractor`	editor	sends a `deselect` token for a mouse click on the editor background
`ClickEditorInteractor`	editor	sends a `click` token for a mouse click an the editor background. The token info contains a `java.awt.Point` instance with the mouse click position
`AddEdgeEditorInteractor`	`LineEditor`	extends `ClickEditorInteractor` and also sends `pressed` and `released` tokens with the `JNode` under the mouse in the token info field. Updates the `LineEditor` when the mouse is dragged.

A.2.3 Presentations

Node Presentations

Name	Required Automaton	Description
`BoxNode`	`Selectable`	a simple selectable box with perimeter connection
`CircleNode`	`Selectable`	a simple selectable ellipse with four point connection
`TextBox`	`Selectable`	box presentation for labeled nodes
`TextCircle`	`Selectable`	ellipse presentation for labeled nodes

Edge Presentations

Name	Required Automaton	Description
`StraightEdge`	`NullDialog`	straight line figure with dot decoration at the source and arrow head decoration at the target side.
`TextLine`	`Selectable`	same as above, but with movable text label.
`DotEdge`	`NullDialog`	straight line figure with dot decoration at both ends.
`OrthoEdge`	`NullDialog`	an orthogonal edge with dot and arrow head decoration, needs `OrthoConnection` on the node to work.
`Curve`	`Selectable`	cubic curve with movable control tangents.
`TextCurve`	`Selectable`	same as above with movable text lable.

Editor Presentations

Name	Required Automaton	Description
`BasicEditor`	`EditorDialog`	blank editor canvas with `BasicEditor-Interactor`

A.2.4 Figures

Node Figures

Name	Parameters	Properties	Description
`BoxFigure`	none	`fill.paint` `outline.paint`	draws a filled, outlined box
`CircleFigure`	none	`fill.paint` `outline.paint`	draws a filled, outlined ellipse
`TextFigure`	none	`font`	draws a text string, requires a `LabeledGraphObject` as model
`SelectedNodeFigure`	one node figure	none	draws selection handles around a rectangular shape
`CompositeNode`	two node figures	none	draws one node figure on top of the other

Edge Figures

Name	Parameters	Properties	Description
`StraightEdgeFigure`	none	`edge.paint`	draws a straight line from source to target site
`CurveFigure`	none	`edge.paint`	draws a cubic curve
`CurveControl`	one edge	`select.paint`	draws control handles for cubic courves
`TextEdge`	none	`font`, `text.paint`	draws an edge label
`TextControl`	one edge	`select.paint`	draws a control handle for text replacement
`OrthoEdgeFigure`	none	`edge.paint`	draws a line consisting of two orthogonal pieces, requires `OrthoConnection`
`CompositeEdge`	two edges	none	draws one edge figure on top of the other

Edge Decorator Figures

Currently there are only three edge decorators. `NullDecorator` draws no decoration at all, `DotDecorator` and `ArrowHead` draw a small dot and an arrow head respectively.

Editor Figures

Name	Parameters	Properties	Description
BlankEditor	none	none	blank editor canvas
LineEditor	none	`select.paint`	a straight line in the editor canvas foreground, provides two methods `setStart` and `setEnd` for interactors to update end positions of the line

A.2.5 Connections

Name	Description
`PerimeterConnection`	calculates the intersection of a straight line between centers of source and target node with the node border, suitable for rectangular shapes and straight line edges
`CirclePerimeter`	same as above for ellipsiod shapes.
`CurveCircePerimeter`	same as above, but uses the control tangents of cubic curves as direction vector.
`FourPointConnection`	uses one point on each side of a node, suitable for all rectangular shapes and `CircleNode`
`OrthoConnection`	uses one point on each side of a node, suitable for all rectangular shapes and `CircleNode`

B Generating Editors with Grace

This tutorial gives a brief hands on introduction to the graph editor generator *grace*.

It first takes a look at how to install and run the generator and then proceeds to a small example graph editor. Using this example we will explore the basic concepts and features of *grace*.

B.1 Installing Grace

B.1.1 System Requirements

The graph editor generator *grace* needs a computing platform that supports a Java JDK 1.2 or above. It has been tested on Windows 95, Windows NT, Linux 2.2.12 i386 and Solaris 7.

For the generated editors something comparable in computing power to a Sun Ultra 60 Workstation is recommended.

B.1.2 Getting Grace

You can download the latest version of *grace* for free from the *grace* homepage at `http://www.doclsf.de/grace/`.

B.1.3 Installation

grace ships as compressed zip or tar archive.

On a Unix system, the archive is installed in your preferred directory (we will call it `$GRACE_HOME`) by `cd $GRACE_HOME` followed by a `tar -xzf grace-1.0.tar.gz` (or similar depending on your distribution).

On a MS Windows system, tools like `WinZip` should do the same job.

After successful installation, you should end up with the following directories:

```
$GRACE_HOME/
        +--bin/                     (start scripts)
        +--doc/                     (documentation)
        +--examples/ ..             (example specifications)
        +--lib/                     (precompiled classes and configuration file)
            +--specs/ ..            (specification libraries)
        +--src/
            +--grace/ ..            (source code of grace)
            +--java_cup/runtime/    (source code of cup runtime classes)
```

After that, you need to tell *grace* where the Java interpreter can be found if you don't have it in your binary path.

You tell *grace* about the java interpreter by editing the script `$GRACE_HOME/bin/grace` for Unix or `$GRACE_HOME\bin\grace.bat` for Windows respectively. It is defined in the first lines of the script by a variable `JAVA` and contains just `java` by default. Change it to suit your needs. For Windows you also have to supply the installation location of *grace* – it is by default on `c:\grace`.

If you want to be able to run *grace* from anywhere, you should include the start script in your binary path.

B.1.4 Running Grace

You can test if your installation was successful by just typing `grace` at the command prompt. It should print out a version number, read in its configuration file and complain about missing input.

The correct syntax for starting the generator is

```
grace <switches> <input-files>
```

where `<input-files>` are one or more files containing graph editor specification modules and `<switches>` are zero or more of the following options:

`-s <directory-list>`
: search for specification modules in the specified directory list. `<directory-list>` is a list of directories as for instance the classpath in Java. On Unix systems, it is a list of directories separated by `:`. On Windows, they are separated by `;`.

`-d <directory>`
: write the generated file to the directory `<directory>`

`-verbose` or `-v`
: display generation progress messages (enabled by default)

`-quiet` or `-q`
: display error messages only (no chatter about what *grace* is currently doing)

`-help` or `-h`
: print a help message explaining options and usage of *grace*.

Applications using *grace* editors should inlcude the file `$GRACE_HOME/lib/grace.jar` in their classpath. The `jar` file can be distributed with your application.

B.2 Specifying a Graph Editor

We will now, at last, come to the example we promised and to the basic concepts and features of *grace*.

Our example is a task editor for a workflow application. Tasks can have transitions between them, and they can be associated with data objects. What we want, is a graphical direct manipulation editor in which the user can enter, browse, and modify graphical task descriptions. The descriptions are to be used by other parts of the application.

Other members of our software project have been working hard, and have already implemented, what we call the problem domain of the editor. They also already did most of the user interface. Management has decided that we do not already have enough work at hand, and that we should deliver that direct manipulation editor for our small application as of tomorrow.

Instead of looking for a new job, we start by taking a look at *grace*.

B.2.1 Describing the Problem Domain

The first step is to think about the problem domain at hand. If we want to use *grace* as a tool for generating an editor, the domain has to have graph-like properties. Incidentally, our problem domain does have exactly that. We can easily imagine the tasks and data objects as classes of nodes, task transitions as one class of edges, and task/data associations as another class of edges.

We have a graph-like problem domain and we want a graphical, direct manipulation editor for that domain, so *grace* is perfectly suitable.

To get an editor for a new problem domain from *grace*, we have to specify this problem domain. We take a look at the already implemented classes and find them in package `model` with names `Task`, `DataObject`, `Transition`, and `TaskDataObjectCouple`. We just have to tell *grace* about it:

```
graph TaskModel {
  import model.*;

  nodes Task, DataObject;
  edges Transition, TaskDataObjectCouple;
}
```

For *grace* to understand and access the classes correctly, they need to implement the interfaces `Graph`, `Node` and `Edge` of package `grace.model`. Out of pure coincidence, the API of the application classes written by our colleagues fits exactly. We just have to add the interfaces to the class declarations and are done. If our colleagues did not have had that unidentified hunch, that caused them to use exactly these method names and signatures, we would either have had to change the API[1], or we could have used the adapter design pattern to simulate the *grace* API on the existing classes.

Either way, the first and most important step is done.

B.2.2 Choosing a Style

Well, we have told *grace* about the problem domain. Now where is my editor?

We are almost finished. We have not yet said, how the editor should look like and where to put it.

Three lines do the job:

```
package ui.editor;

uses grace.standards;

editor style Basic;
```

They tell *grace* to put the generated classes in package `ui.editor`, to use the standard graph editor specification library and the main editor style `Basic` that is defined in that standard library.

[1] which might be a bad idea, because other classes may already depend on it

Assuming we put the specification in a file with name `spec/graph.spec` in directory `examples/tutorial/step1/` we can get a first very prototypy version of the editor by issuing

```
grace -s spec graph.spec
```

on the command line.

grace in turn says something like

```
Welcome to the Java graph editor generator grace (version 1.0).

Reading configuration file [/usr/lib/grace-1.0/lib/grace.config]
Reading [./spec/grace.spec]
Reading [/usr/lib/grace-1.0/lib/specs/grace/standards.spec]
Reading [/usr/lib/grace-1.0/lib/specs/grace/styles.spec]
Reading [/usr/lib/grace-1.0/lib/specs/grace/presentation.spec]
Reading [/usr/lib/grace-1.0/lib/specs/grace/automaton.spec]
Reading [/usr/lib/grace-1.0/lib/specs/grace/lowlevel.spec]
Writing [gui/editor/BasicEditorDialogTokens.java]
Writing [gui/editor/BasicEditorDialog.java]
...
Writing [gui/editor/StraightEdge.java]
Writing [gui/editor/ListenerZList.java]
Writing [gui/editor/TaskModelEditor.java]
```

The `-s spec` switch tells *grace* to search for specification files in the standard library path and in the directory `spec/` for specification files. We could have used a simple `grace spec/graph.spec` as well, but that would not have demonstrated how to use the search path switch.

Among the generated classes is one that is called `TaskModelEditor`. That is what we wanted in the first place – the editor. It is a subclass of `javax.swing.JPanel` and we use like we would use any other component.

In our main application window class `gui/MainWindow.java` we include the new editor by adding the following few lines to the class and constructor:

```
private TaskModelEditor view;

public MainWindow(TaskModel model) {
  ...
  view = new TaskModelEditor(model);
  view.setSize(800,600);
  ...
  content.add(view, BorderLayout.CENTER);
  ...
}
```

By writing a small test method `makeGraph`, we construct an example model and add some tasks, data objects and transitions. We use the `setProperty` method of `JGraph` to assign predefined positions and sizes to the nodes.

```
private void makeGraph() {
  Task authentificate    = new Task("authentificate");
  Task givesubjects      = new Task("givesubjects");
  DataObject authentdata = new DataObject("authentdata");
  DataObject subjects    = new DataObject("subjects");

  model.add(authentificate);
  model.add(givesubjects);
  model.add(authentdata);
  model.add(subjects);
  model.add(new Transition(authentificate,givesubjects));

  view.setProperty(authentificate, "bounds", new Rectangle(200,200,100,50));
  view.setProperty(givesubjects,   "bounds", new Rectangle(400,200,100,50));
  view.setProperty(authentdata,    "bounds", new Rectangle(200,300,100,50));
  view.setProperty(subjects,       "bounds", new Rectangle(400,300,100,50));
}
```

We can now compile the whole thing and take a look at the editor running. We see something like in figure B.1.

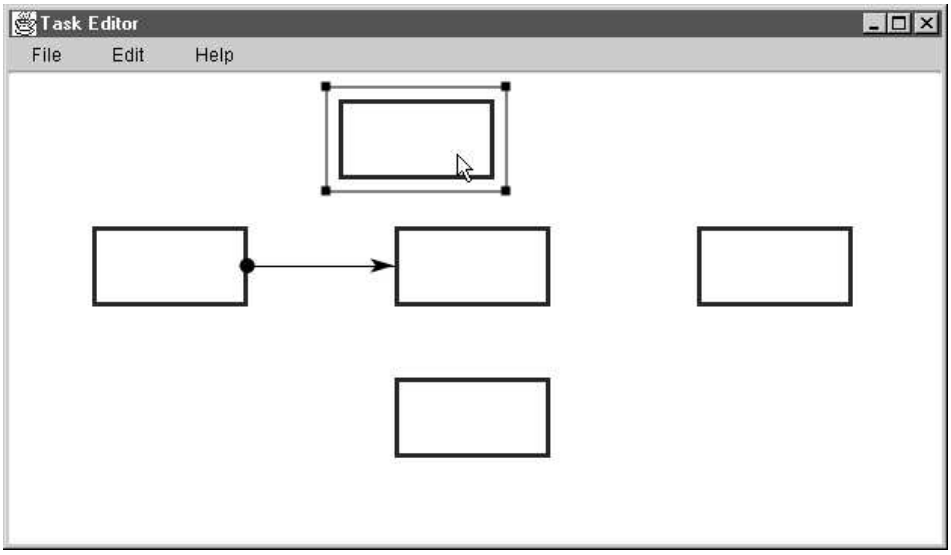


Figure B.1: The first generated prototype editor

We are disappointed. No way our boss will be satisfied with that. The editor is graphical and direct manipulation and all that, but there is no text in the nodes, and they all look the same, and we cannot create new ones, and it's basically not what we wanted.

Since this is a tutorial and we are using *grace*, this is not a real problem. If we want the nodes to contain text and if we want to have different presentations of different classes of nodes and edges, we just tell *grace* about it, hoping she will fix it.

We do this by creating an own style, extending the existing `Basic`:

```
style Task extends Basic {
  Task                : (TextCircle, Selectable);
  DataObject          : (TextBox, Selectable);
  Transition          : (StraightEdge, Selectable);
  TaskDataObjectCouple : (DotEdge, Selectable);
}
```

A *grace* style is a mapping from problem domain class to a dialog description and a presentation description. All dialog and presentation descriptions in our style above are contained in the *grace* standard libraries.

We change the `editor style` to `Task` and start *grace* on `spec/graph.spec` in directory `examples/tutorial/step2/`.

The editor now results in something like figure B.2.

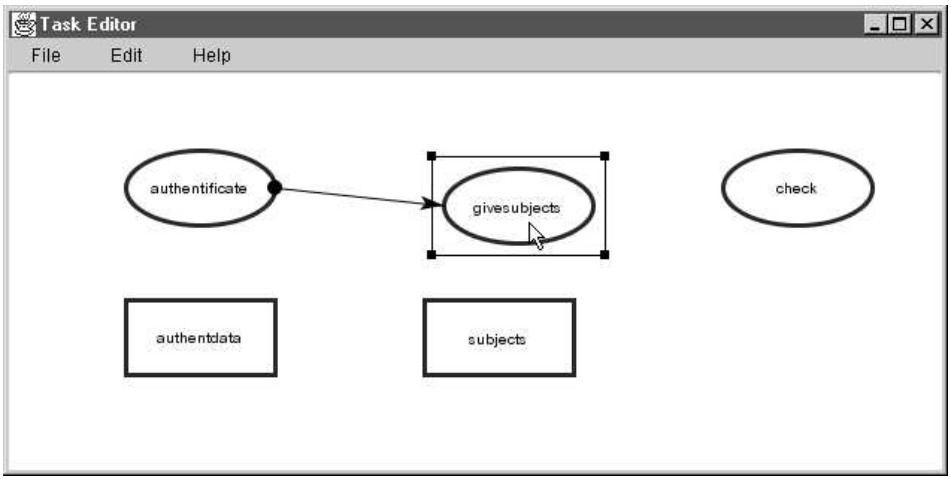


Figure B.2: A customized style

B.2.3 Customizing Interaction

Figure B.2 looks better, but we can still not add any new tasks and transitions to it.

The standard libraries knew nothing about our problem domain with tasks and transitions. The basic editor dialog in that library doesn't contain any code to create our model objects.

We fix this by extending that editor dialog. *grace* describes abstract dialogs with automata containing states and transitions. Each abstract automaton state has its own presentation and transitions are fired when the automaton receives input tokens represented in software by class `grace.editor.Token`. A very basic automaton for selectable objects, that we already used in our prior style definition, looks like that:

```
automaton Selectable {

  tokens select, deselect;
  states not_selected, selected;

  >not_selected:
    (select) -> selected

  selected:
    (deselect) -> not_selected
}
```

In *grace* editors, each graph object as well as the editor itself is associated with an abstract dialog control automaton describing the high level interaction properties of that object. These automata can communicate with each other by sending tokens.

The example above was relatively simple; for our editor to be able to create new nodes, we need something a bit more sophisticated like:

```
automaton TaskEditorDialog extends EditorDialog {

  import model.*;

  tokens addData, addTask;
  states addData, addTask;

  >edit:

  addData:
```

```
    (click) -> edit do {
      DataObject d = new DataObject("new data object");
      editor.getModel().add(d);

      Point p = (Point) token.info;
      editor.setProperty(d, "bounds", new Rectangle(p.x, p.y, 100, 50));
    }

  addTask:
    (click) -> edit do {
      Task t = new Task("new task");
      editor.getModel().add(t);

      Point p = (Point) token.info;
      editor.setProperty(t, "bounds", new Rectangle(p.x, p.y, 100, 50));
    }

  all:
    (addData) -> addData
    (addTask) -> addTask
}
```

Ok, this probably needs some explanation. With `import model.*` we tell *grace* to do a Java `import model.*` in the generated automaton class, because we want convenient access to out model classes in the transition action code. The next two lines declare two tokens `addData` and `addTask` and two states with the same names.

Our intention is that, when in state `addTask`, the editor will create a new task object and add it to the model when the user clicks on a free spot of editor background. In this abstract dialog we do currently not yet care, where this `click` token inherited from `EditorDialog` in the standard library comes from. If we receive it however in that state, the editor automaton should return to the standard state `edit` and execute the bit of action code that creates a new task. When the new task is created and added to the model, the editor is told to put it at a predefined size at the position that is provided by the info field of the input token.

There is also that curious undeclared state `all`. It means that the transitions listed there apply to all states of the automaton. So we specified, that whenever the tokens `addData` and `addTask` are received, no matter what state it currently has, the automaton will change into states `addData` or `addTask` respectively.

The remaining question is: Where do these abstract tokens come from? And where does the mouse click position come from anyway?

The answer is: from anywhere. A token can be sent from the rest of the application

user interface, directly from deep inside the application code, or from the editor itself. In case of the `click` token with the mouse position info, it comes from the editor itself. *grace* lets you specify how and when these tokens are created in a relatively convenient way by *interactors*.

A very simple interactor sending `select` tokens to the editor when a figure is clicked on looks like:

```
interactor ClickInteractor {
  states start;

  >start:
    (clicked) sends EditorDialog.select to editor
}
```

The interactor producing the `click` token from above is part of the standard library and called `ClickEditorInteractor`. It is a bit more complicated, because it includes custom information in the token to be sent:

```
interactor ClickEditorInteractor {
  requires editor;

  import java.awt.Point;
  states normal;

  >normal:
    (clicked; editor.getGraphObject(event) == null) do {
      editor.processToken(new Token(EditorDialogTokens.click_TOKEN, this,
                                    new Point(event.getX(), event.getY())));
    }
}
```

The events available in interactors are listed in the grace configuration file[2].

B.2.4 Customizing Presentation

In the previous section, we have created a new abstract dialog for the editor and switching to directory `examples/tutorial/step3/` we have put the module in the file `spec/automaton.spec`.

[2] see also B.1.3

The basic standard editor presentation does not yet know about our customizations. A presentation specification in *grace* is basically a mapping from automaton control state to abstract interaction objects. Abstract interaction objects in *grace* consist of figures and interactors. What we want from our editor is, that in the new states `addTask` and `addData`, the `ClickEditorInteractor` is used sending `click` Tokens instead of the standard deselect behavior, when the user clicks on the editor background.

In a new specification module, that we call `spec/presentation.spec`, we therefore write:

```
editor presentation TaskEditor extends BasicEditor {
  requires automaton TaskEditorDialog;

  addData: (BlankEditor, ClickEditorInteractor)
  addTask: (BlankEditor, ClickEditorInteractor)
}
```

If we use both new modules in our main specification `spec/graph.spec` and also add a new mapping `Editor : (TaskEditor, TaskEditorDialog)` to the `Task` style, we have specified all we need to create new tasks and data objects in the editor.

If we start our application now, everything will look as it has before. We did not yet give the user a possibility to get into one of the states `addData` or `addTask`. This kind of operation is perfectly suited for a toolbar. We look into class `MainWindow` and find that there is already one implemented. It provides a fair set of operations: *open* and *save* for task descriptions, and a group of buttons that is directly concerned with the editor. These include standard editor modes like *select*, *add new task*, *add new data object*, *add new edge* and *remove objects*.

The toolbar implemented by our hard working colleagues invokes callback functions in class `MainWindow` that they have left empty, because they didn't know how to communicate with the editor that is our job to write. These are the places to send tokens to the main editor automaton. We change the callbacks to

```
  public void editMode() {
    view.processToken(new Token(edit_TOKEN, this));
  }

  public void addTaskMode() {
    view.processToken(new Token(addTask_TOKEN, this));
  }

  public void addDataObjectMode() {
```

```
    view.processToken(new Token(addData_TOKEN, this));
  }

  public void removeMode() {
    view.processToken(new Token(remove_TOKEN, this));
  }
```

To get convenient access to the generated token kind constants (e.g. `addData_Token`), we let `MainWindow` implement the interface `gui.editor.TaskEditorDialogTokens` that contains these constants.

If we now generate a new editor and start it, we are able to create new tasks and data objects.

If we also want to be able to create transitions, we can easily do this by the same mechanisms as described above: We extend the abstract dialog and update the presentation mapping. This is the main task in writing a customized editor specification.

Because we are still in the process of developing the automaton `TaskEditorDialog`, we this time do not use the extension mechanism, but just change our specification by adding a new token and a new state `addEdge`. We want the user to create a new edge by pressing the mouse button on the source node, dragging the mouse to the target node, and releasing the mouse at the target node. The `EditorDialog` in `grace.automaton` already defines two abstract tokens for exactly this purpose. Using these, our state definition for `addEdge` looks like

```
addEdge:
  (pressed) do { source = ((JNode) token.getSource()).getNode(); }

  (released) -> edit do {
    target = ((JNode) token.getSource()).getNode();

    if (source instanceof Task) {
      Graph model = editor.getModel();

      if (target instanceof Task)
        model.add(new Transition((Task) source, (Task) target));

      if (target instanceof DataObject)
        model.add(new TaskDataObjectCouple((Task) source,
                                           (DataObject) target));
    }
  }
```

```
  all:
   (addEdge) -> addEdge
```

If a `pressed` token is received, the source of that token is assumed to be of type `JNode` and to be the node the user pressed the mouse button on. Its model is stored in a field `source`. The automaton remains in state `addEdge`.

If a `released` token is received, the model of the source of that token is stored as `Node` in `target`. If source and target have the right types, we create a new edge, i.e. a new transition or a new task/data association, and add it to the graph.

It again remains to find an interactor and a figure that implement the behavior that is expected by the abstract dialog description. Taking a look at the standard libraries we find, that there is an interactor `AddEdgeEditorInteractor` together with a figure `LineEditor` doing exactly this job. This leads to a new presentation description:

```
editor presentation TaskEditor extends BasicEditor {
  requires automaton TaskEditorDialog;

  addData: (BlankEditor, ClickEditorInteractor)
  addTask: (BlankEditor, ClickEditorInteractor)
  addEdge: (LineEditor,  AddEdgeEditorInteractor)
}
```

Now, we have in figure B.3 a graph editor that automatically creates the right kind of edge, depending on which kinds of nodes the user connects.

However, if we start the editor we notice, that the toolbar is not always in sync with the editor itself. If we add a new task for instance, the toolbar button for *add new task* remains selected while the editor instantly goes to state `edit` when the task has been created.

B.2.5 Connecting Graph Editor and Application

To keep the toolbar in sync with the editor, we first notice that the toolbar button states directly correspond to the editor automaton control state, and then ask ourselves, how `MainWindow` can be kept informed about the current editor control state.

The solution is the `StateListener` interface. It works very much the way a more high level Java AWT event listener like e.g. `ItemListener` is used. We let `MainWindow` implement the `StateListener` interface and attach it to the editor.

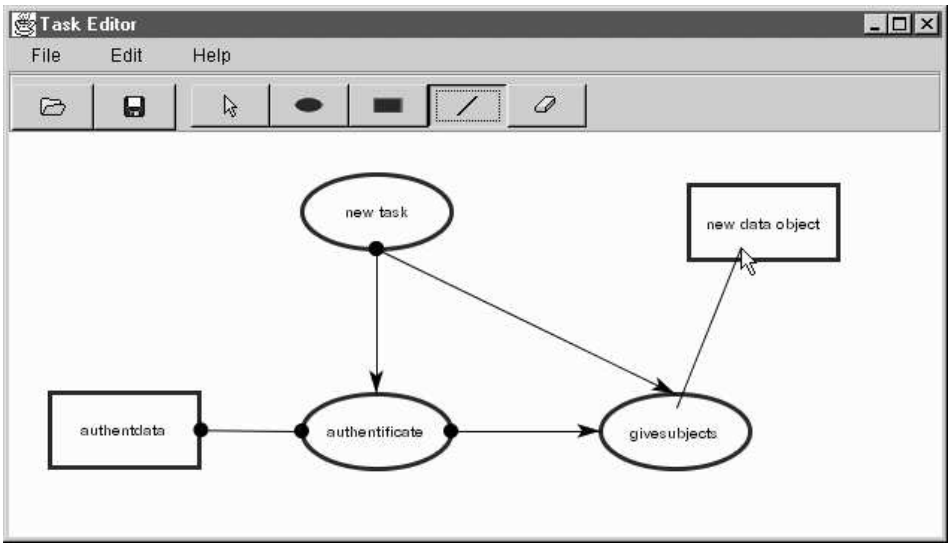


Figure B.3: Customized interaction

As a result the method `void stateChanged(StateChangeEvent event)` is called each time the control state of the editor automaton changes. This method is the place to update the toolbar button state to reflect the current editor state.

This technique – apart from solving our immediate problem – also has the advantage that it is relatively tolerant to changes in the specification. If we for instance changed the transition from `addEdge:(click) -> edit do ..` to `addEdge:(click) do ..`, we do not have to change a single line of code in the user interface and it will still be in sync with the editor states.

This kind of listener interface is also used in other places of *grace*. There is for instance the `java.beans.PropertyChangeListener` that is used for figure properties, the `grace.model.GraphListener` that *grace* uses to keep in sync with the problem domain classes, and the `java.awt.ItemListener` that the main editor provides to keep the application updated about changes in the current selection of graph objects.

The `ItemListener` can for instance conveniently be used for a property sheet that gives the user the ability to modify graph object properties like the one showing in figure B.4.

B.2.6 Persistence with Grace

We have now a graphical direct manipulation editor for our problem domain. We can modify the structure and the appearance of task description, and the editor is tightly

integrated into the rest of the application user interface. In fact it is so tightly integrated, that most people would perceive the menubar, the toolbar, or the property sheet as parts of the editor – even though they either only work on the task model itself and can therefore be used independently of the editor, or they may also have other more application UI related duties.

One of these more application UI related duties is the persistence of task descriptions the user has built with the new editor. Persistence is of course also very strongly related to the editor. Because we did not build or purchase an automatic layout algorithm for our task descriptions, not only the logical structure, but also the physical appearance of the diagram has to be made persistent. *grace* achieves this by using the builtin Java serialization support. All *grace* generated classes know how to make themselves persistent and also how to deserialize themselves again without loosing the connection to their model counterparts in the application code.

By default only the properties of figures, i.e. color, position, size, etc., is stored – the current dialog state is not. It is instead freshly initialized when a persistent diagram is deserialized.

For this scheme to work properly, the model classes have to support serialization too. If they cannot be stored, the diagram cannot be stored either. Fortunately, supporting serialization in Java in most cases only means to implement the empty interface `java.io.Serializable`.

Having read all that, writing the `save()` method for our task description is simple:

```
public void save() {
  ..
  FileOutputStream file  = new FileOutputStream(files.getSelectedFile());
  ObjectOutputStream out = new ObjectOutputStream(file);

  out.writeObject(model);
  out.writeObject(view);

  out.close();
  ..
}
```

The `load()` part is also not very complicated:

```
public void load() {
  ..
  FileInputStream file = new FileInputStream(files.getSelectedFile());
  ObjectInputStream in = new ObjectInputStream(file);
```

```
    center.remove(view);

    model = (TaskModel) in.readObject();
    view  = (TaskModelEditor) in.readObject();

    in.close();

    center.add(view, BorderLayout.CENTER);
    properties.use(view);
    ..
  }
```

If the serialization support of *grace* can not be used for some reason, one could also request the properties of all graph objects in the editor and store them in some custom file format together with information about which properties belong to which model instance.

When upon retrieval a new instance of the editor is provided with these properties for each model instance, diagram appearance will also be restored. This way offers complete control over all persistence aspects, but does of course also cost more own work.

If we create a subclass of the generated editor and override the `paintBackground` method to display a nice positioning grid in the editor background, we get our first real prototype of the task model editor in figure B.4 and already know a lot about how to generate editors with *grace*.

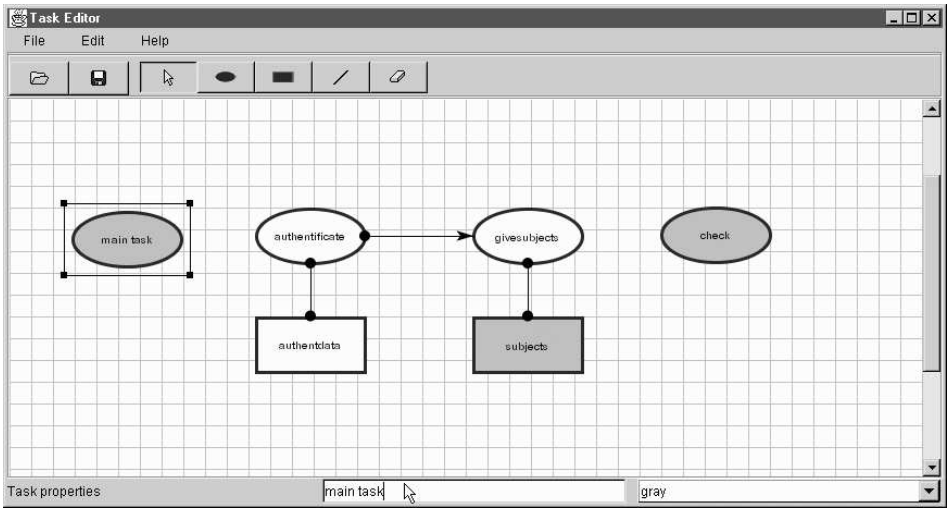


Figure B.4: More integration

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[2] Farahangiz Arefi, Charles E. Hughs, and David A. Workman. Automatically generating visual syntax-directed editors. *Communications of the ACM*, 33(3):349–360, March 1990.

[3] Bernhard Bauer. Generating the user interface from formal specifications of the application. In Jean Vanderdonckt, editor, *Computer-Aided Design of User Interfaces, Proceedings of CADUI'96*, Namur, 1996. Presses Universitaires de Namur.

[4] Roland Bechtel. Einbettung des μ-Kalkül Model-Checkers Mucke in AutoFocus. diploma thesis, Chair of Computer Science IV, Technical University of Munich, 1999.

[5] Alfons Brandl and Gerwin Klein. FormGen: A Generator for Adaptive Forms Based on EasyGUI. In Hans-Jörg Bullinger and Jürgen Ziegler, editors, *Human-Computer Interaction: Ergonomics and User Interfaces. Proceedings of HCI International '99 (the 8th International Conference on Human-Computer Interaction), Munich, Germany, August 22-26*, pages 1172–1176, London, 1999. Lawrence Erlbaum Associates.

[6] Max Breitling, Ursula Hinkel, and Katharina Spies. Formale Entwicklung verteilter reaktiver Systeme mit FOCUS. In Hartmut König and Peter Langendörfer, editors, *Formale Beschreibungstechniken für Verteilte Systeme, 8. GI/ITG Fachgespräch*, pages 63–74, Aachen, 1998. Shaker Verlag.

[7] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons, New York, 1996.

[8] M. Chen, P. Townsend, and C. Y. Wang. Automated construction of application-specific graph editors in an object-oriented paradigm. In *Proceedings of the Fifth*

International Conference on Human-Computer Interaction, volume 2 of *Software Tools*, pages 415–420, 1993.

[9] Sitt Sen Chok and Kim Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Structuring Pen Input, pages 185–194, 1998.

[10] Jürgen Eickel. Generierung von Benutzungsoberflächen. Lecture at the Technical University of Munich, 1998.

[11] John Ellson, Emden Gansner, Eleftherios Koutsofios, and Stephen North. GraphViz – tools for viewing and interacting with graph diagrams. Available in the World Wide Web at `http://www.research.att.com/sw/tools/graphviz`.

[12] M. Fröhlich and M. Werner. Demonstration of the interactive graph-visualization system davinci. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pages 266–269. DIMACS, Springer-Verlag, October 1994.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1996.

[14] James Gosling, Bill Joy, and Guy Steele. *The Java^{tm} Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[15] Herbert Göttler. Graph grammars and diagram editing. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 216–231, 1987.

[16] Michael Himsolt. Graph^{Ed}: An interactive graph editor. In B. Monien and R. Cori, editors, *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science*, volume 349 of *Lecture Notes in Computer Science*, pages 532–533, Berlin, February 1989. Springer.

[17] Michael Himsolt. The Graphlet system. In *Proc. Graph Drawing, Berkeley, California*, volume 1190 of *Lecture Notes in Computer Science*, pages 233–240, Berlin, 1996. Springer.

[18] Franz Huber. Ein generisches Werkzeug zur Bearbeitung von Graphen und deren Umsetzung in Textdarstellungen. diploma thesis, Chair of Computer Science IV, Technical University of Munich, 1994.

[19] Franz Huber, Sascha Molterer, Andreas Rausch, Bernhard Schätz, Marc Sihling, and Oscar Slotosch. Tool supported specification and simulation of distributed systems. In Bernd Krämer, Naoshi Uchihira, Peter Croll, and Stefano Russo, editors, *Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 155–164, Los Alamitos, California, 1998. IEEE Computer Society.

[20] Franz Huber and Bernhard Schätz. Specification modules for methodical system development. In H. König and P. Langendörfer, editors, *8. GI/ITG Fachgespräch "Formale Beschreibungstechniken für verteilte Systeme"*, pages 75–86. Shaker Verlag, Aachen, 1998.

[21] Franz Huber, Bernhard Schätz, and Geralf Einert. Consistent graphical specification of distributed systems. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: 4th International Symposium of Formal Methods Europe*, volume 1313 of *Lecture Notes in Computer Science*, pages 122–141, Berlin, 1997. Springer.

[22] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus - a tool for distributed systems specification. In *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*, pages 467–470, Berlin, 1996. Springer.

[23] Scott E. Hudson. CUP LALR Parser Generator for Java. Available in the World Wide Web at `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[24] Gerwin Klein. JFlex – The Fast Lexical Analyzer Generator for Java. Available in the World Wide Web at `http://www.jflex.de/`.

[25] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly, Sebastopol, 2nd edition, 1992.

[26] Franz Lonczewski and Siegfried Schreiber. The FUSE-system: an integrated user interface design environment. In Jean Vanderdonckt, editor, *Computer-Aided Design of User Interfaces, Proceedings of CADUI'96*, pages 37–56, Namur, 1996. Presses Universitaires de Namur.

[27] Kim Marriott, Sitt Sen Chok, and Alan Finlay. A tableau based constraint solving toolkit for interactive graphical applications. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, pages 340–354, Berlin, 1998. Springer.

[28] Carolyn McCreary and Larry Barowski. VGJ: Visualizing Graphs Through Java. In Sue H. Whitesides, editor, *Graph Drawing: 6th International Symposium*, volume 1547 of *Lecture Notes in Computer Science*, pages 454–455, Berlin, 1998. Springer.

[29] Mark Minas. Diagram editing with hypergraph parser support. In *Proceedings of the IEEE Symposium on Visual Languages (VL'97), Capri, Italy*, pages 230–237. IEEE Computer Society Press, 1997.

[30] S. Näher. LEDA: a library of efficient data types and algorithms. In Patrice Enjalbert, Alain Finkel, and Klaus W. Wagner, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS '93)*, volume 665 of *Lecture Notes in Computer Science*, pages 710–723, Berlin, Germany, February 1993. Springer. Available in the World Wide Web at `http://www.mpi-sb.mpg.de/LEDA/`.

[31] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, August 1973.

[32] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 119–156. Springer, 1999.

[33] Frances Newbery Paulish. *The design of an extendible [sic] graph editor*, volume 704 of *Lecture Notes in Computer Science*. Springer, New York, 1993. Revision of the author's thesis (doctoral — University of Karlsruhe, 1991).

[34] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language Based Editors*. Springer-Verlag, 1989.

[35] Jason Robbins, David Hilbert, and Adam Gauthier. GEF graph editing framework: An open source java library for connected graph editors. Available in the World Wide Web at `http://www.ics.uci.edu/pub/arch/gef/`.

[36] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[37] Ben Schneiderman. *Designing the User Interface, Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Reading, Massachusetts, 1998.

[38] Siegfried Schreiber. Specification and generation of user interfaces with the BOSS-system. In Brad Blumenthal, Juri Gornostaev, and Claus Unger, editors, *Human-Computer Interaction: 4th International Conference, EWHCI '94 St. Petersburg, Russia, Selected Papers*, volume 876 of *Lecture Notes in Computer Science*, pages 107–120, 1994.

[39] Siegfried Schreiber. The BOSS-System: Coupling Visual Programming with Model Based Interface Design. In F. Paterno, editor, *Proceedings Eurographics Workshop Design, Specification, Verification of Interactive Systems*, Berlin, 1994. Springer.

[40] Siegfried Schreiber. *Specification Techniques and Generation Tools for Graphical User Interfaces (in german).* Phd dissertation, Technical University of Munich, Herbert Utz Verlag Wissenschaft, Munich, 1997.

[41] Pedro Szekely. Retrospective and challenges for model-based interface development. In Jean Vanderdonckt, editor, *Computer-Aided Design of User Interfaces, Proceedings of CADUI'96*, Namur, 1996. Presses Univ. de Namur.

[42] Gerd Szwillus. User interface definition based on graphical structure editors. In Gerd Szwillus and Lisa Neal, editors, *Structure-Based Editors and Environments*, Computers and People Series, pages 253–275, San Diego, CA, 1996. Academic Press.

[43] Gerhard Viehstaedt. *A Generator for Diagram Editors.* Dissertation, Institut für Mathematische Maschinen und Datenverarbeitung (Informatik), Friedrich Alexander Universität, Erlangen-Nürnberg, 1995.

[44] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, 1990.

[45] Josef Voss and Dietmar Nentwing. *Entwicklung von graphischen Benutzungsschnittstellen: Modelle Techniken und Werkzeuge der User-Interface-Gestaltung.* Carl Hanser Verlag, München, 1998.

[46] Robin J. Wilson. *Introduction to Graph Theory.* John Wiley & Sons, New York, 3rd edition, 1985.

[47] Robin J. Wilson and Lowel W. Beinecke, editors. *Applications of graph theory.* Academic Press, London, 1979.

[48] Gaby Zinßmeister and Carolyn McCreary. Drawing graphs with attribute graph grammars. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proceedings of the Fifth International Workshop on Graph Grammars and Their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 443–453. Springer, 1996.

Index

action, 45, 51, 79, 84, 85, 105, 106
actor, 28–30, 32
adapter, 69, 76, 118
AddEdgeEditorInteractor, 110
ancestor, 86, 91
architecture, 47, 56, 65, 69, 77, 79
ArrowHead, 113

BasicEditor, 111
BasicEditorDialog, 109
BasicEditorInteractor, 110
beans, 77
BlankEditor, 113
BoxFigure, 112
BoxNode, 111

CircleFigure, 112
CircleNode, 111
CirclePerimeter, 113
classgen, 94
classpath, 103, 116, 119
ClickEditorInteractor, 110
ClickInteractor, 109
CompositeEdge, 112
CompositeNode, 112
CUP, 94
Curve, 111
CurveCirclePerimeter, 113
CurveControl, 112
CurveFigure, 112
CurveInteractor, 110

DCA, 45–54, 59, 70, 83, 86, 104
decorator, 73, 91, 107, 113
delegation, 70–72
DotDecorator, 113
DotEdge, 111
DragInteractor, 110

EditorDialog, 109
event, 39, 45, 51, 70, 75, 88, 89, 106, 107

figure, 56–57, 66, 71–74, 90, 105–107
FourPointConnection, 113

generated code, 39, 66, 77
GraphListener, 75, 76, 128

hypergraph, 25

ItemListener, 127

JFlex, 94

layout, 3, 5, 31, 35, 38, 66, 74–75
LineEditor, 110, 113
listener, 70, 75–77, 87, 88, 106, 127

NullDecorator, 113
NullDialog, 109, 111
NullInteractor, 109

observer, 75–76
OrthoConnection, 113
OrthoEdge, 111

OrthoEdgeFigure, 112

PerimeterConnection, 113
pgen, 93
priority, 48, 51, 86, 87, 104
properties, 53, 73, 75–77, 79, 128
PropertyChangeListener, 77

ResizeInteractor, 110

search path, 119
Selectable, 109, 111
SelectedNodeFigure, 110, 112
semantics, 46, 47, 83, 86, 103
StateListener, 77
StraightEdge, 111
StraightEdgeFigure, 112
style, 1, 2, 38, 60, 91, 103, 108, 118
syntax, 44, 55, 59, 61, 101, 116

TextBox, 111
TextCircle, 111
TextCurve, 111
TextCurveIntercator, 110
TextEdge, 112
TextFigure, 112
TextInteractor, 110
TextLine, 111
token, 45, 46, 51, 70, 76, 85, 105, 106

UML, 11, 12

visitor, 94