Christian-Albrechts-Universität zu Kiel

Diplomarbeit

Automatisches Layout von Statecharts unter Verwendung von *GraphViz*

cand. ing. Tobias Kloss
(Mat.Nr. 524942)

3. Mai 2005

Institut für Informatik und Praktische Mathematik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden

betreut durch:
Steffen H. Prochnow

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angebenen Hilfsmittel verwendet habe.

Kiel,

____

Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Modul für den Einsatz im Projekt *KIEL* entwickelt, das Layouts von *Statecharts* automatisiert berechnet. Für die Betrachtung *Dynamischer Statecharts* (*Dynamic Charts*) werden mehrere Ansichten eines *Statecharts* bereitgestellt. Nach der Evaluation diverser Graph-Zeichen-Werkzeuge erwies sich das Projekt *GraphViz* als eine geeignete Grundlage zur automatisierten Berechnung von *Statechart*-Layouts.

Diese Arbeit zeigt, dass das hier vorgestellte Verfahren die Lesbarkeit von *Statechart*-Diagrammen durch eine einheitliche und strukturierte Darstellung der einzelnen Elemente erhöht. Da ein automatisiertes Verfahren zur Berechnung eines *Statechart*-Layouts in kurzer Zeit ein gut lesbares und somit verständliches *Statechart*-Diagramm erzeugen kann, empfiehlt sich der generelle Einsatz.

Schlüsselwörter *Statechart*, *Automatisiertes Layout*, *GraphViz*, *Dynamische Statecharts*, *KIEL*

Inhaltsverzeichnis

1. Einleitung **1**

2. Grundlagen **5**
2.1. Statecharts . . . 5
2.2. Das Projekt *KIEL* . . . 8
2.2.1. Der konzeptionelle Aufbau . . . 8
2.2.2. Die Datenstruktur . . . 9
2.2.3. Die Module . . . 15
2.2.4. Die grafische Benutzeroberfläche . . . 16

3. Analyse verwandter Arbeiten **19**
3.1. Generelle Verfahren zum Layout von Graphen . . . 19
3.1.1. Layout von Graphen mittels Planarisierung . . . 20
3.1.2. Orthogonales Layout . . . 20
3.1.3. Baum-Layout . . . 21
3.1.4. Ebenenbasiertes Layout . . . 23
3.1.5. *Force-Directed*-Layout . . . 25
3.2. Auswahl eines geeigneten *Frameworks* . . . 26
3.2.1. Anforderungen . . . 27
3.2.2. Überblick . . . 29
3.2.3. Direkter Vergleich und Selektion . . . 36
3.3. *GraphViz* im Detail . . . 37
3.3.1. Aufbau eines *GraphViz*-Graphen . . . 38
3.3.2. Die möglichen Zugriffsmethoden . . . 39
3.3.3. Der *Dot*-Layout-Algorithmus . . . 42
3.4. Zusammenfassung der Analyse . . . 47

4. Das Layout-Modul **49**
4.1. Spezifikation der Anforderungen . . . 49
4.2. Konzept und Schnittstellendefinition . . . 50
4.3. Die Layouter-Klassen im Detail . . . 53
4.3.1. Der LayerwiseLayouter . . . 53
4.3.2. Der GraphvizLayouter . . . 57
4.3.3. Der GraphvizBinaryLayouter . . . 60
4.3.4. Der GraphvizLibraryLayouter . . . 62
4.3.5. Der HVLayouter . . . 64

4.4. Zusammenfassung 65

5. Zusammenfassung **67**
5.1. Fallbeispiel 67
5.2. Erreichte Ziele 72
5.3. Offene Probleme 74
5.4. Ausblicke 76

6. Literaturverzeichnis **79**

A. Einstellungen **85**

B. *Code*-Beispiel für die Benutzung der *GraphViz*-Bibliothek **97**
B.1. Beschreibung 97
B.2. Der *Code* 97

C. Der Quell-*Code* des Moduls `Layouter` **99**
C.1. Beschreibung 99
C.2. Der *Java-Code* 101
C.2.1. `kiel.layouter.Handler` 101
C.2.2. `kiel.layouter.LayerwiseLayouter` 104
C.2.3. `kiel.layouter.Layouter` 109
C.2.4. `kiel.layouter.LayouterProperties` 111
C.2.5. `kiel.layouter.PseudoLayouter` 119
C.2.6. `kiel.layouter.graphviz.GraphvizLayouter` 120
C.2.7. `kiel.layouter.graphviz.GraphvizBinaryLayouter` 124
C.2.8. `kiel.layouter.graphviz.CircoBinLayouter` 129
C.2.9. `kiel.layouter.graphviz.DotBinLayouter` 130
C.2.10. `kiel.layouter.graphviz.NeatoBinLayouter` 131
C.2.11. `kiel.layouter.graphviz.TwopiBinLayouter` 132
C.2.12. `kiel.layouter.graphviz.GraphvizLibraryLayouter` 133
C.2.13. `kiel.layouter.graphviz.GraphvizAPI` 136
C.2.14. `kiel.layouter.graphviz.CircoLibLayouter` 141
C.2.15. `kiel.layouter.graphviz.DotLibLayouter` 142
C.2.16. `kiel.layouter.graphviz.NeatoLibLayouter` 143
C.2.17. `kiel.layouter.graphviz.TwopiLibLayouter` 144
C.2.18. `kiel.layouter.hvlayouter.HVLayouter` 145
C.2.19. `kiel.layouter.hvlayouter.HVLayoutedLayer` 146
C.2.20. `kiel.layouter.hvlayouter.HorizontalLayoutedLayer` 150
C.2.21. `kiel.layouter.hvlayouter.VerticalLayoutedLayer` 156
C.2.22. `kiel.layouter.hvlayouter.ConversionLookupTable` 162
C.2.23. `kiel.layouter.hvlayouter.HVLayoutedEdge` 163
C.2.24. `kiel.layouter.hvlayouter.HVLayoutedNode` 166
C.2.25. `kiel.layouter.hvlayouter.Level` 169

C.2.26. kiel.layouter.hvlayouter.StateLevel 170
C.2.27. kiel.layouter.hvlayouter.TransitionLevel 171
C.2.28. kiel.layouter.hvlayouter.PropertyComparator 173
C.2.29. kiel.layouter.hvlayouter.InitialStateComparator 174
C.2.30. kiel.layouter.hvlayouter.NumberOfLevelsComparator . . . 175
C.2.31. kiel.layouter.hvlayouter.TransitionCrossingComparator 176
C.2.32. kiel.layouter.hvlayouter.WellBalancedLevelsComparator 177
C.2.33. kiel.layouter.hvlayouter.PermutationIterator 178
C.3. Der *C++-Code* 179
C.3.1. kiel_layouter_graphviz_GraphvizAPI.h 179
C.3.2. kiel_layouter_graphviz_GraphvizAPI.c 181

Tabellenverzeichnis

2.1. *KIEL*-Transitionen im Vergleich 13
2.2. Grafische Informationen und ihre zugehörigen Objekte 14

3.1. Graph-Zeichen-Werkzeuge im direkten Vergleich 36

5.1. Komplexitätsanalyse des Fallbeispiels 71

Abbildungsverzeichnis

1.1. Teilsystem einer Klimaanlage 1
1.2. Gegenüberstellung zweier *Statechart*-Layouts 2

2.1. Der konzeptionelle Aufbau eines *Esterel Studio-Statecharts* 6
2.2. Der konzeptionelle Aufbau eines *Matlab-Stateflow-Statecharts* 7
2.3. Der konzeptionelle Aufbau des Projekts *KIEL* 9
2.4. Klassendiagramm des *KIEL*-Modells 10
2.5. Klassendiagramm der *KIEL*-Zustände 12
2.6. Klassendiagramm der *KIEL*-Transitionen 13
2.7. Klassendiagramm der *KIEL*-Layout-Informationen 14
2.8. Die *KIEL*-Anwendung im `Browser`-Modus 17
2.9. Die *KIEL*-Anwendung im `Editor`-Modus 17

3.1. Vorgehen der Planarisierungsmethode 20
3.2. Platzierung auf einem Gitternetz 21
3.3. Ein typisches orthogonales Layout (Quelle: [26]) 22
3.4. Ein typisches Baum-Layout (Quelle: [26]) 23
3.5. Wandlung eines Graphen: ungerichtet → azyklisch gerichtet 24
3.6. Ein typisches Layout im Sugiyama-Stil (Quelle: [26]) 25
3.7. Ein typisches *Force-Directed*-Layout (Quelle: [26]) 26
3.8. Beispielgraph im *Dot*-Layout 38
3.9. Beispielgraph im *Dot*-Format 39
3.10. Beispielgraph nach dem *Dot*-Layout im *Attributed-Dot*-Format 40
3.11. Erstellen eines Knotens mit der *C++*-Bibliothek 41
3.12. Erstellen einer Kante mit der *C++*-Bibliothek 41
3.13. Finden des optimalen Spannbaums (Quelle: [20]) 43
3.14. Erstellung und Layout eines Hilfsgraphen 45
3.15. Graph mit eingezeichneten Boxen (Quelle: [20]) 46
3.16. Festlegen des *Spline*-Verlaufs (Quelle: [20]) 47

4.1. Das Klassendiagramm des Layout-Moduls 51
4.2. *Statechart* mit einer hierarchieübergreifenden Transition 54
4.3. Ein *Statechart* mit markierten Hierarchieebenen 55
4.4. Simulationsabfolge mit *Semantischem Zoom* 56
4.5. Platzieren des initialen Zustands 59
4.6. Zuordnungsproblem von *Label* und Transition 60
4.7. Deklaration einer *JNI-Java*-Funktion zum Erstellen eine Kante . . . 63

4.8. *JNI*-*C++*-Funktion zum Erstellen eine Kante 63
4.9. Ein *Statechart* im HVLayout 64

5.1. Ausgangslage des Fallbeispiels 68
5.2. Fallbeispiel im *Dot*-Layout 68
5.3. Erste Region des Zustands WaitAB im *Dot*-Format 69
5.4. Erste Region des Zustands ABSync (komplett) 69
5.5. Zweite Region des Zustands ABSync (komplett) 70
5.6. Oberste Hierarchieebene des *Statecharts* 70
5.7. Weitere Beispiele 71
5.8. Fehlerhaftes Layout mit dem `DotLibLayouter` 75
5.9. Transitions-*Label* ragt aus Oberzustand 76
5.10. Platzsparendere Anordnung der Regionen eines parallelen Zustands . 77
5.11. Nutzung mehrzeiliger Transitions-*Label* 78

A.1. Veränderung der initialen Ausrichtung 86
A.2. *Statechart*-Diagramm in konstanter Layout-Richtung 86
A.3. Veränderung der vertikalen Ausrichtung 87
A.4. Veränderung der Schriftgröße 89
A.5. Auswirkungen der Gewichtsreduktion 90
A.6. Ausrichtung und Layout-Verlauf 91
A.7. Hinzufügen virtueller Transitions-*Label* 92
A.8. Veränderung Prioritätseinstellungen 92
A.9. Ausrichtungsmöglichkeiten der Zustände 94
A.10. Einstellungen des `HVLayouter` 95

1. Einleitung

In unserer technologischen Zeit erhalten immer mehr technische Geräte Einzug in das tägliche Leben, sei es die Waschmaschine, der Fernseher, die Mikrowelle oder das Mobiltelefon. Alle diese Gegenstände verfügen über eingebaute „Computer“, wobei diese bei manchen offensichtlicher sind als bei anderen. Der gebräuchliche Computer am heimischen Schreibtisch stellt in der Liste der verbauten Mikrocontroller nur einen geringen Prozentsatz dar. Da sich diese Computer fast unbemerkt in unsere Umwelt integrieren, nennt man sie *Eingebettete Systeme*.

Diese kleinen Computer integrieren sich jedoch nicht nur in unsere Umwelt, sie interagieren auch mit ihr. Zum Beispiel überwacht eine Klimaanlage fortlaufend die Raumtemperatur. Sollte die Raumtemperatur eine gesetzte Mindesttemperatur unterschreiten, schaltet das System die Heizung ein. Sobald die Raumtemperatur auf einen mittleren Wert gestiegen ist, schaltet das System die Heizung wieder ab. Entsprechendes geschieht, wenn die Raumtemperatur zu hoch ist. Da Systeme wie das eben beschriebene auf die Einflüsse ihrer Umwelt reagieren, werden sie auch *Reaktive Systeme* genannt.

Bei der Entwicklung solcher Systeme wird zunächst mit speziellen Computerprogrammen ein Modell erstellt, das vor der realen Umsetzung getestet und analysiert werden kann. Diese Programme werden unter dem Begriff *Modellierungswerkzeuge* zusammengefasst. Die gängigste Methode zur Modellierung *Reaktiver Systeme* ist die Verwendung von Zustands-Übergangs-Diagrammen (*Statecharts*), welche das Systemverhalten zustandsbasiert beschreiben. Bei dieser Beschreibungsart wird einem System eine endliche Menge an Zuständen zugewiesen, in denen es sich befinden kann. Für das Beispiel der Klimaanlage wären Heizung an und Heizung aus mögliche Zustände.

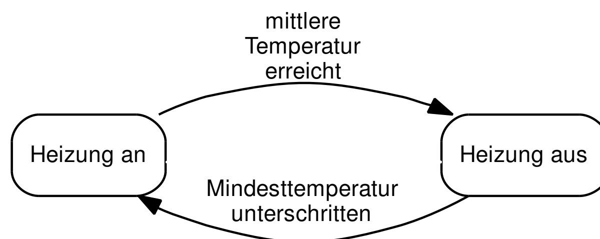
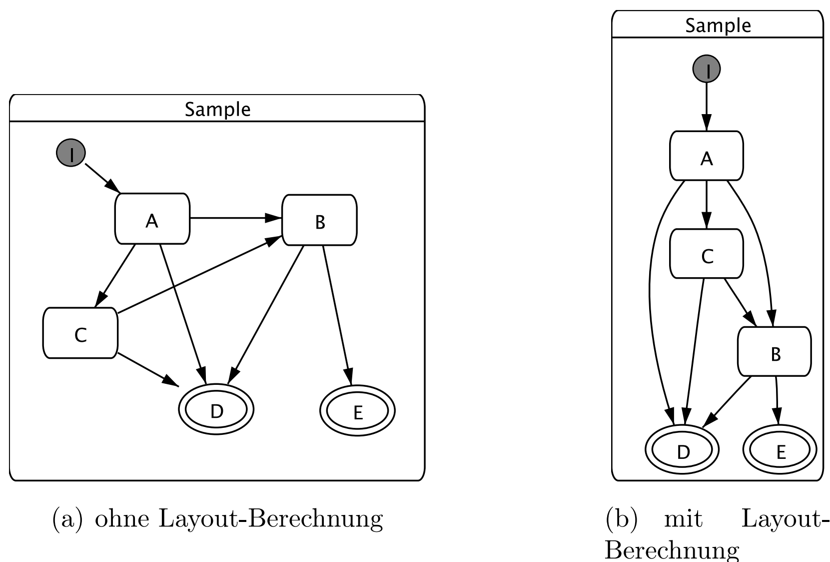


Abbildung 1.1.: Teilsystem einer Klimaanlage[1]

[1]Diese Grafik ist mit dem *GraphViz*-Program *Dot* erstellt worden.

Neben den Zuständen gibt es auch noch Zustandsübergänge (*Transitionen*), die einen Zustandswechsel beschreiben. Im Falle der Klimaanlage wäre eine Transition von Heizung aus nach Heizung an denkbar. Im Folgenden werden solche Transitionen in dieser Form dargestellt: (Heizung aus $\rightarrow$ Heizung an). Ein solcher Zustandswechsel wird durch ein Ereignis (*Trigger*) ausgelöst. Für die beschriebene Transition aus dem Beispiel wäre Mindesttemperatur unterschritten ein mögliches Ereignis. Dieses Ereignis wird an die Transition in Form einer Beschriftung (*Label*) gebunden.

Das betrachtete Teilsystem einer Klimaanlage könnte somit wie in Abbildung 1.1 dargestellt aussehen. Wie man sich leicht vorstellen kann sind die *Statecharts* realer Systeme wesentlich komplexer und nicht so überschaubar wie das gezeigte Beispiel. Das Erstellen solcher *Statecharts* erfordert viel Disziplin, um die Diagramme lesbar zu gestalten. Dies gilt insbesondere für das Hinzufügen neuer Zustände in ein bestehendes *Statechart*, da hierfür zunächst freier Platz geschaffen werden muss. Dadurch wird die bisherige Anordnungmeist stark beeinträchtigt, so dass große Teile des Diagramms erneut arrangiert werden müssen, um die Lesbarkeit wieder herzustellen.



(a) ohne Layout-Berechnung

(b) mit Layout-Berechnung

Abbildung 1.2.: Gegenüberstellung zweier *Statechart*-Layouts[2]

Unstrukturierte Diagramme, wie in Abbildung 1.2(a) dargestellt, verringern die Lesbarkeit und damit das intuitive Verständnis des Systemverhaltens. Da das manuelle Arrangieren der *Statechart*-Elemente sehr zeitaufwendig ist, wird im Rahmen dieser Arbeit eine Methode zur automatischen Berechnung eines *Layouts* beschrieben. Die auf diese Weise bearbeiteten Diagramme, wie in Abbildung 1.2(b) zu sehen, weisen weniger Überlagerungen und Überschneidungen der einzelnen Elemente auf. Durch die einheitliche Darstellung der einzelnen Systemteile wird die Lesbarkeit des Diagramms zusätzlich erhöht.

[2]Diese sowie alle weiteren *Statechart*-Diagramme sind mittels der *KIEL*-Applikation (siehe Abschnitt 2.2) erstellt worden.

Um das Systemverhalten während einer Simulation verständlicher zu machen, werden unterschiedliche Ansichten für ein *Statechart* generiert. Diese Ansichten unterscheiden sich in der jeweils dargestellten Detailstufe. Dafür werden Informationen ausgeblendet, die zu einem bestimmten Simulationszeitpunkt irrelevant sind. Hierdurch erhöht sich die Informationsdichte der Ansicht. Im Rahmen dieser Arbeit wird das Verfahren des *Semantischen Zoom* umgesetzt, das in diesen Kontext einzuordnen ist.

Der Aufbau der vorliegenden Arbeit stellt sich wie folgt dar:

- Im folgenden Kapitel werden zunächst die Grundlagen dieser Arbeit beschrieben. Dafür wird ein kurzer Überblick über das Themengebiet der *Statecharts* gegeben. Anschließend wird das Projekt *KIEL* vorgestellt, in dessen Umfeld diese Arbeit eingebettet ist.

- In Kapitel 3 werden generelle Verfahren zur Berechnung von Graphen-Layouts vorgestellt. Es folgt eine Erörterung der Vor- und Nachteile diverser Graph-Zeichen-Werkzeuge. Für die weitere Verwendung in dieser Arbeit wurde das Projekt *GraphViz* ausgewählt, dessen detaillierte Vorstellung das Kapitel abschließt.

- Aufbauend auf das Projekt *GraphViz* wird in Kapitel 4 die Entwicklung des Layout-Moduls für das Projekt *KIEL* beschrieben. Dafür werden zunächst grundlegende Schnittstellen definiert. Anschließend werden die einzelnen Klassen zur Layout-Berechnung vorgestellt, insbesondere auch die Klasse zur Anbindung des Projekts *GraphViz*.

- In Kapitel 5 werden die erreichten Ziele und offenen Probleme diskutiert. Ein erläutertes Fallbeispiel rundet die Vorgehensweise des implementierten Layout-Verfahrens ab.

2. Grundlagen

In diesem Kapitel werden die Grundlagen dieser Arbeit beschrieben. Dafür wird zunächst in Abschnitt 2.1 ein genereller Überblick über das Themengebiet der *Statecharts* gegeben. Anschließend folgt in Abschnitt 2.2 eine Beschreibung des Projekts *KIEL*, in dessen Umfeld diese Arbeit eingebettet wird.

2.1. Statecharts

Die Verwendung von grafischen Softwareentwicklungswerkzeugen hat gegenüber den rein textbasierten Werkzeugen in den letzten Jahren immer mehr an Bedeutung gewonnen. Dies liegt vor allem daran, dass die textliche Darstellung von Informationen die Fähigkeit des Menschen nicht voll nutzt. Da der Mensch überwiegend bildliche Informationen wahrnimmt, hat sich ein Großteil des menschlichen Gehirns auf die Verarbeitung dieser Informationen spezialisiert.

Durch die visuelle Darstellung eines Systems lassen sich Zusammenhänge zwischen einzelnen Teilen leichter erfassen. Dies gilt insbesondere für die Darstellung paralleler Ausführungspfade, die bei einer textlichen Darstellung meist nicht auf den ersten Blick zu erkennen sind.

Die erste Form der grafischen Systementwicklung war die Gruppe der *Endlichen Automaten*, die ein System bereits zustandsbasiert beschrieben haben. Ein derart beschriebenes System bestand aus mehreren Zuständen, von denen immer nur genau einer aktiv war. Durch das Auftreten bestimmter Ereignisse konnte diese Aktivität auf einen anderen Zustand übergehen. Abhängig von dem gerade aktiven Zustand war es möglich, dass die Systemantwort unterschiedlich ausfiel.

Statecharts erweitern diese *Endlichen Automaten* um Hierarchie und Parallelität, damit Zustandsexplosionen bei großen Systemen verringert werden. Als ihr Erfinder gilt David Harel, der sie 1987 in seinem Papier [25] das erste Mal vorstellte. Das Hinzufügen von Hierarchieebenen ermöglicht eine abstrakte Herangehensweise an die Entwicklung eines Systems. So lässt sich zunächst die Interaktion großer Systemteile spezifizieren, die anschließend immer weiter verfeinert werden können. Auch bei der Analyse eines Systems erhöht das Ein- und Ausblenden unterschiedlicher Abstraktionsstufen das intuitive Verständnis des Systemverhaltens.

Die Nutzung der Parallelität ermöglicht eine übersichtlichere Darstellung des strukturellen Aufbaus, die wiederum das intuitive Verständnis erhöht. Darüber hinaus verringert sich die Anzahl der benötigten Zustände und insbesondere auch der Transitionen rapide, wenn mehrfache Parallelität mit einer Vielzahl von Zuständen pro Ausführungspfad modelliert werden soll.

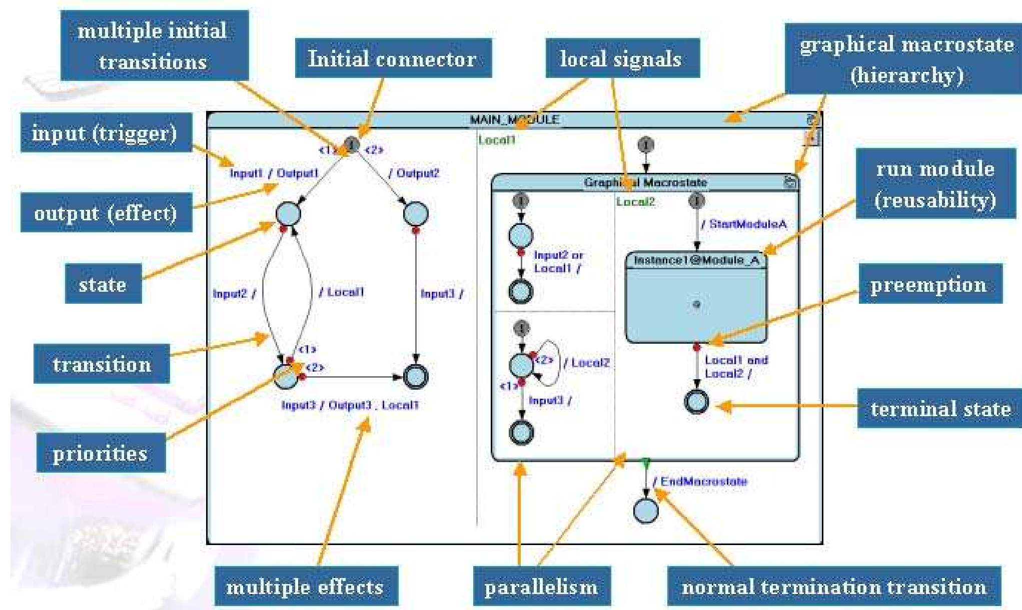


Abbildung 2.1.: Der konzeptionelle Aufbau eines *Esterel Studio*-*Statecharts* (Quelle: *Esterel Studio* Handbuch [13])

Die von Harel vorgestellten *Statecharts* sind nur ein Vorschlag zur Beschreibung des Systemverhaltens. Seitdem haben sich viele unterschiedliche Varianten herausgebildet, die zwar auf den ersten Blick generelle Übereinstimmungen zeigen, sich hingegen in ihren Details unterscheiden. Um diese Unterschiede im Systemverhalten von *Statecharts* zu minimieren, hat die *Object Management Group* (OMG) die *Statecharts* in ihrer Spezifikation der *Unified Modeling Language* (UML) [34] aufgenommen. Allerdings sind diese *Statecharts* inzwischen eher als eine weitere Variante zu betrachten.

Die *Statechart*-Varianten unterscheiden sich jedoch nicht nur in ihrem Verhalten, sondern auch in ihrer Darstellung. Da sich diese Arbeit inbesondere mit der Darstellung von *Statecharts* befasst, wird im Weiteren nicht näher auf die Verhaltensunterschiede, sondern zwei unterschiedliche Darstellungsformen eingegangen. Abbildung 2.1 zeigt den konzeptionellen Aufbau eines *Esterel Studio*-*Statecharts*, Abbildung 2.2 für ein *Matlab*-*Stateflow*-*Statechart*.

Ein Vergleich der beiden Abbildungen zeigt, dass beide Varianten zwar grundlegende Gemeinsamkeiten haben, sich dennoch in manchen Details unterscheiden. *Esterel Studio* verwendet beispielsweise einen initialen Zustand (*Initial connector*) mit einer oder mehreren ausgehenden Transitionen, wohingegen *Stateflow* eine Standardtransition (*Default transition*) definiert. Für die Berechnung eines *Statechart*-Layouts ist insbesondere die unterschiedliche Darstellung der Parallelität zu berücksichtigen. Generell werden in beiden Fällen parallele Ausführungen in separate Bereiche unterteilt. *Esterel Studio* trennt diese Bereiche durch eine markierende Linie, *Stateflow* sieht hierfür einen gestrichelten Rahmen vor.

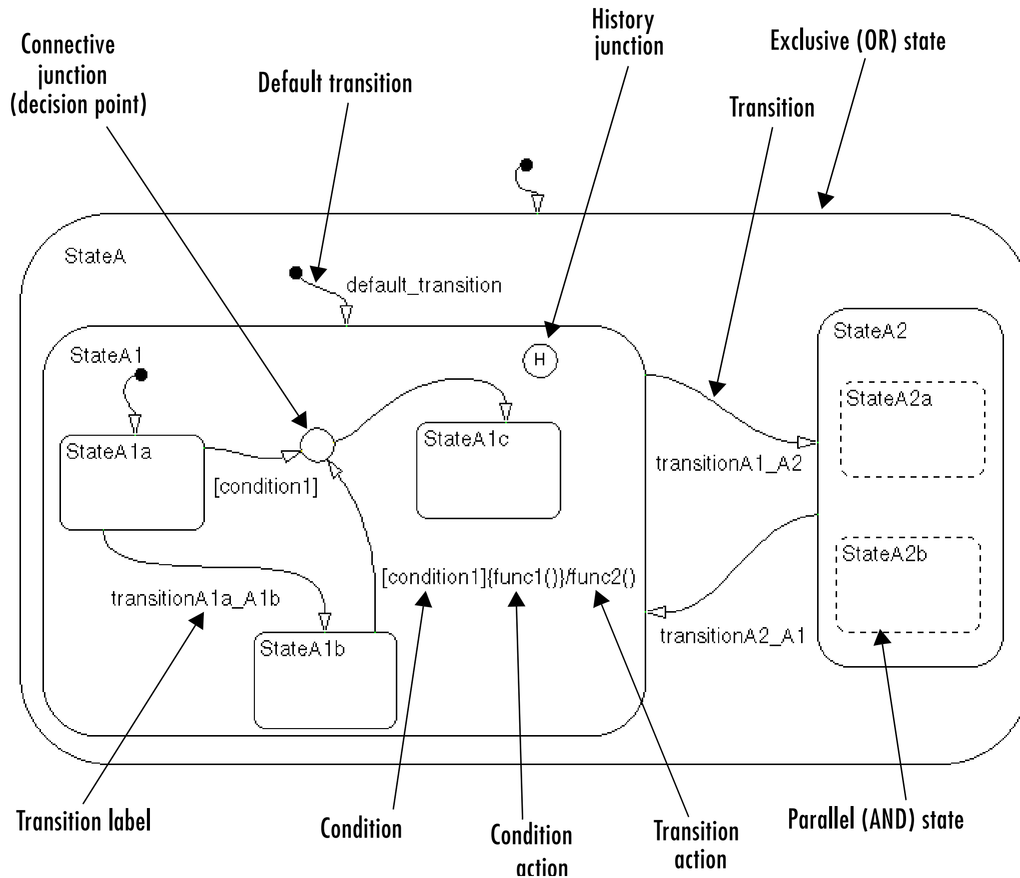


Abbildung 2.2.: Der konzeptionelle Aufbau eines *Matlab-Stateflow-Statecharts* (Quelle: *Stateflow* Handbuch [30])

Trotz dieser marginalen Unterschiede lässt sich eine gemeinsame Obermenge der Darstellungsformen für alle grafischen Objekte erkennen. Zustände werden als Kreise, Ellipsen oder Rechtecke dargestellt, und die Transitionen werden als Pfeile, die diese geometrischen Formen verbinden, gezeichnet. In dieser verallgemeinerten Betrachtung lassen sich *Statecharts* als gerichtete Graphen betrachten, deren Knoten Untergraphen enthalten können. Darüber hinaus können in manchen Werkzeugen (z. B. *Statemate*) Kanten zwischen Knoten unterschiedlicher Untergraphen existieren. Solche Kanten werden im Folgenden als *Hierarchieübergreifende Transitionen* bezeichnet und in Abschnitt 4.3.1 erneut aufgegriffen.

Nach dieser kurzen Einführung in das Themengebiet der *Statecharts* wird im folgenden Abschnitt das Projekt *KIEL* vorgestellt, in dessen Umfeld diese Arbeit eingebettet wird.

2.2. Das Projekt *KIEL*

Das Projekt *KIEL* [50] ist ein Modellierungswerkzeug für *Statecharts*, das am Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme der Christian-Albrechts-Universität zu Kiel entwickelt wird. Der Begriff *KIEL* steht für „**K**iel **I**ntegrated **E**nvironment for **L**ayout“.

Die meisten erhältlichen Modellierungswerkzeuge beschränken sich während einer Simulation auf eine statische Darstellung des Systems. Im Gegensatz dazu versucht das Projekt *KIEL* durch die dynamische Darstellung der Elemente das intuitive Verständnis des simulierten Systemverhaltens zu erhöhen. In diesem Zusammenhang wurde der Begriff der *Dynamischen Statecharts* (*Dynamic Charts*) vom Projekt *KIEL* geprägt.

Um auf eine breite Datenbasis für das Arbeiten mit *Dynamischen Statecharts* zurückgreifen zu können, werden die proprietären Datenformate existierender Modellierungswerkzeuge ausgelesen und weiter verarbeitet. Die dynamische Darstellung des Simulationsverhaltens wird dabei durch ein automatisches Layout unterstützt. Darüber hinaus ist geplant, Funktionsteile zur Synthese und Analyse von *Statecharts* zu integrieren.

Im folgenden Abschnitt wird zunächst der konzeptionelle Aufbau des Projekts *KIEL* beschrieben. Im Anschluss daran wird in Abschnitt 2.2.2 näher auf Implementierungsdetails und die einzelnen Funktionsteile eingegangen. Beide Beschreibungen dienen ausschließlich dem grundlegenden Verständnis der Struktur. Für weiterführende Arbeiten empfiehlt es sich, die API-Dokumentation [49] des Projekts *KIEL* zu nutzen.

2.2.1. Der konzeptionelle Aufbau

Das Projekt *KIEL* ist nach dem *Model-View-Controller*-Prinzip (MVC) entwickelt worden und setzt sich aus den folgenden drei großen Komponenten zusammen:

- Datenstruktur,
- Module und
- einer einheitlichen grafischen Benutzeroberfläche (GUI).

Diese Komponenten setzen sich wiederum aus kleineren Teilen zusammen, wie in Abbildung 2.3 zu sehen ist.

Bei der Entwicklung des Projekts wurde auf eine starke Kapselung der einzelnen Komponenten geachtet, so dass zwischen den einzelnen Teilen möglichst keine Abhängigkeiten bestehen. Dies gilt insbesondere für die Entwicklung der unterschiedlichen Module, da diese im Rahmen von studentischen Arbeiten zeitgleich entstanden sind. Die starke Kapselung vermindert die Gefahr von Konflikten und ermöglicht eine gute parallele Entwicklung.

Um diese Kapselung zu gewährleisten wurde ein zentrales Modul entwickelt, das den Daten- und Kontrollfluss steuert und gleichzeitig die grafischen Komponenten

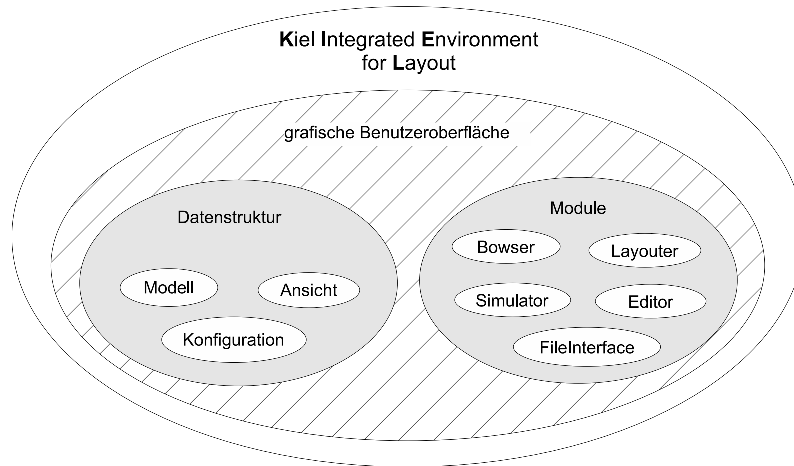


Abbildung 2.3.: Der konzeptionelle Aufbau des Projekts *KIEL*

in einer einheitlichen Benutzeroberfläche vereint. Im nächsten Abschnitt wird die Datenstruktur des Projekts *KIEL* vorgestellt, die die gemeinsame Grundlage für alle Entwicklungen ist.

2.2.2. Die Datenstruktur

Im vorherigen Abschnitt wurde das *Model-View-Controller*-Prinzip bereits angeschnitten. Die Umsetzung dieses Prinzips bedeutet für den Aufbau der Datenstruktur, dass Syntax und Semantik eines *Statecharts* (Modell) und dessen grafische Repräsentation (Ansicht) entkoppelt sind. Auf Grund dieser Entkopplung lassen sich leicht mehrere Ansichten für dasselbe *Statechart* erzeugen und darstellen. Dies bildet die Grundlage für die Arbeit mit *Dynamischen Statecharts*. Es folgt nun eine Beschreibung der Modellteile und anschließend wird deren grafische Repräsentation erläutert.

Das Modell

Die *Object Management Group* (OMG) hat in ihrer Spezifikation [34] der *Unified Modeling Language* (UML) in Abschnitt 2.12 die Semantik von *Statecharts* beschrieben. In Anlehnung an diese Spezifikation ist das Modell des Projekts *KIEL* entwickelt worden. Das Klassendiagramm in Abbildung 2.4 gibt einen Überblick über Semantik und Syntax der Datenstruktur.

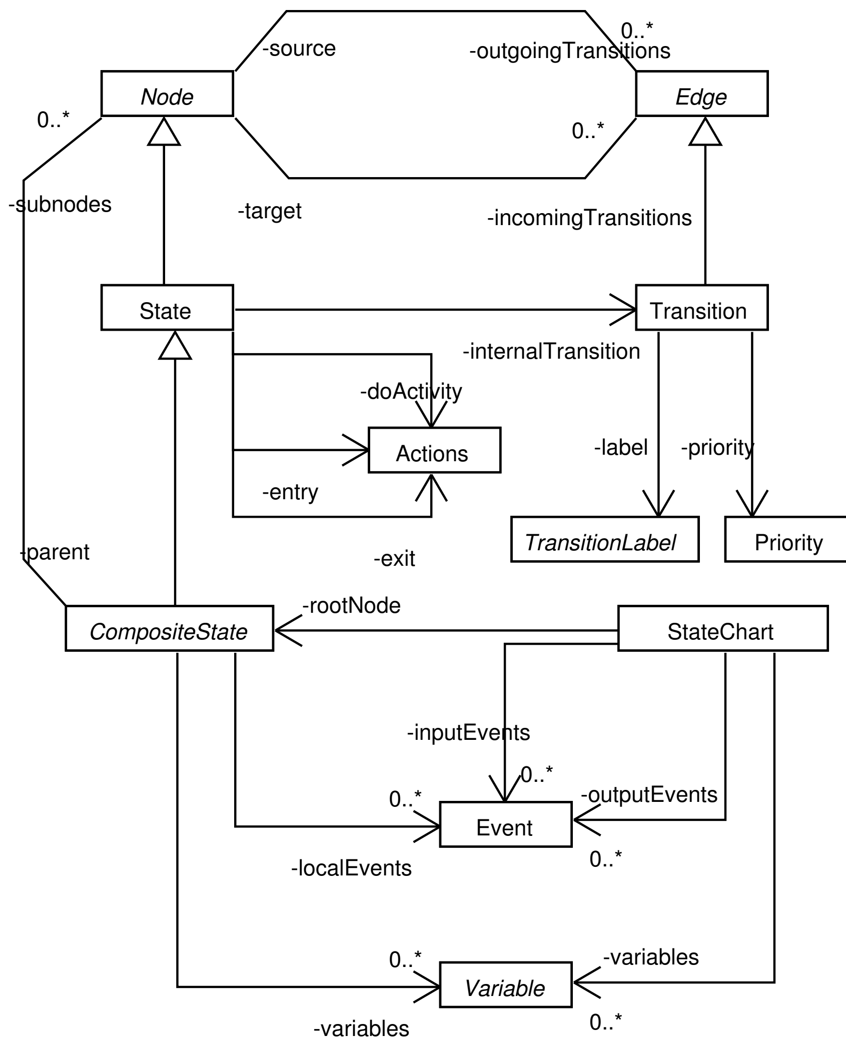


Abbildung 2.4.: Klassendiagramm des *KIEL*-Modells[1]

Ein *KIEL-Statechart* wird durch die Klasse `StateChart` repräsentiert, die sich aus den folgenden Teilen zusammensetzt:

- Eingangsereignisse (`inputEvents`),
- Ausgangsereignisse (`outputEvents`),
- Variablen (`variables`) und
- einem Wurzelknoten (`rootNode`).

[1]Dieses sowie alle weiteren Klassendiagramme sind mit *ArgoUML* [1] entwickelt worden.

Dieser Wurzelknoten ist vom Typ `CompositeState`, für den sich lokale Ereignisse (`localEvents`) und Variablen (`variables`) definieren lassen. Als besonderes Merkmal eines `CompositeState` ist die Zusammensetzung aus mehreren Unterzuständen (`subnodes`) zu nennen.

Ein solcher Unterzustand kann ein beliebiger Knoten (`Node`) sein, der eingehende (`incomingTransitions`) und ausgehende Transitionen (`outgoingTransitions`) haben kann. Eigentransitionen sind sowohl eingehende als auch ausgehende Transitionen. Auf die genaue Struktur der Zustandsklassen wird später noch einmal eingegangen, für den ersten Überblick ist die Betrachtung einer weiteren Zustandsklasse ausreichend. Von dem eben beschriebenen „einfachen“ Knoten leitet sich die Klasse eines Zustands (`State`) ab.

Für einen Zustand (`State`) lassen sich die drei folgenden Arten von Aktivitäten definieren:

- Eintrittsaktivität (`entry`), die beim Betreten eines Zustands ausgeführt wird,
- Austrittsaktivität (`exit`), die beim Verlassen eines Zustands ausgeführt wird, und
- eine laufende Aktivität (`doActivity`), die ausgeführt wird während der Zustand aktiv ist.

Darüber hinaus verfügt ein Zustand noch über eine interne Transition (`internalTransition`).

Eine Transition (`Transition`) erweitert eine „einfache“ Kante (`Edge`) um eine Priorität (`priority`) und ein *Label* (`Label`). Die Prioritäten einer Transition werden, abhängig von dem ausgehenden Zustand, eindeutig vergeben, so dass Nichtdeterminismen während der Simulation ausgeschlossen werden können. Bei dem `Label` wird zwischen dem interpretierbaren `CompoundLabel`, das sich aus *Trigger*, *Condition* und *Effect* zusammensetzt, und einem `StringLabel`, das ein einfacher Textbezeichner ist, unterschieden.

Wie der Überblick zeigt besteht der Modellteil der Datenstruktur aus drei großen Teilen:

- Knoten, auf deren Struktur im Anschluss näher eingegangen wird,
- Kanten, die nach den Knoten vertiefend beschrieben werden, und
- nicht-grafische Objekte, wie Ereignisse, Aktionen und Variablen, auf die in dieser Arbeit nicht näher eingegangen wird, da sie aus Sicht eines automatischen Layouts nicht relevant sind.

Struktur der Knotenklassen Abbildung 2.5 zeigt das Klassendiagramm der unterschiedlichen *KIEL*-Zustände. Allen gemein ist die Basisklasse `Node`, die einen Zustand zunächst eher graphenbezogen als einfachen Knoten darstellt. Diese Klasse unterteilt sich in Zustände (`State`) und Pseudozustände (`PseudoState`), die sich in

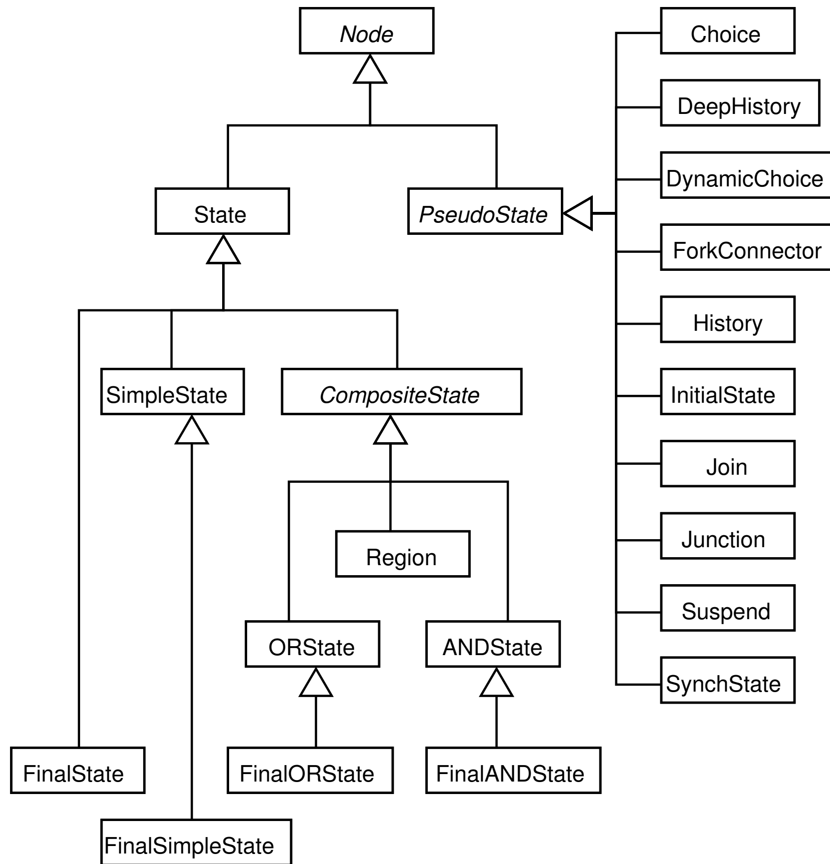


Abbildung 2.5.: Klassendiagramm der *KIEL*-Zustände

ihrem Simulationsverhalten unterscheiden. Man spricht von einem stabilen Zustand der Simulation eines *Statecharts*, wenn kein Pseudozustand und somit ausschließlich „echte" Zustände aktiv sind.

Des Weiteren können Aktivitäten, wie sie im vorangegangenen Überblick beschrieben wurden, nur für „echte" Zustände definiert werden. Zur Gruppe der Pseudozustände zählt unter anderem der initiale Zustand. Die Gruppe der „echten" Zustände unterteilt sich in:

- einfache Zustände (`SimpleState`),
- finale Zustände (`FinalState` bzw. `FinalSimpleState`) und
- Zustände mit Unterebenen (`CompositeState`)

Bei den zusammengesetzten Zuständen wird zwischen jenen mit paralleler Ausführung (`ANDState`) und solchen ohne (`ORState`) unterschieden. Dabei enthält ein `ANDState` pro Ausführungspfad eine `Region`, die sich ebenfalls vom `CompositeState` ableitet. Sowohl der `FinalORState` als auch der `FinalANDState` erweitern ihre Vaterklassen, um als finaler Zustand modelliert werden zu können.

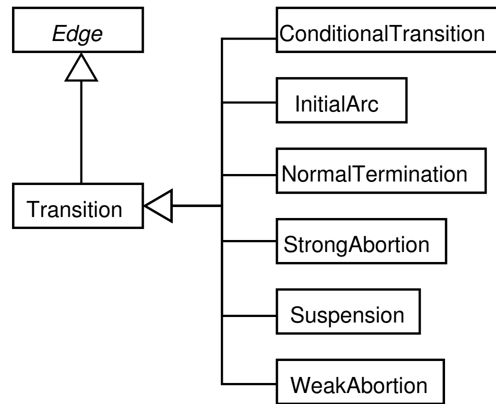


Abbildung 2.6.: Klassendiagramm der *KIEL*-Transitionen

Transition	ausgehend von	Besonderheit
`ConditionalTransition`	`Choice` bzw. `DynamicChoice`	
`InitialArc`	`InitialState`	
`NormalTermination`	alle bis auf oben genannte	hat keine Bedingung, wird ausgelöst, wenn Quellzustand terminiert
`StrongAbortion`	alle bis oben auf genannte	Auslösung hat Priorität vor `WeakAbortion` und Inhalt des Quellzustands
`Suspension`	`Suspend`	
`WeakAbortion`	alle bis auf oben genannte	Bedingung wird erst nach der Ausführung des Inhalts des Quellzustands überprüft

Tabelle 2.1.: *KIEL*-Transitionen im Vergleich

Struktur der Kantenklassen Im folgenden Abschnitt werden die verschiedenen Unterarten einer Kante beschrieben, deren Klassendiagramm in Abbildung 2.6 dargestellt ist.

Die Basisklasse `Edge` ist sehr allgemein gehalten und enthält nur einen Quell- und einen Zielknoten. Eine `Transition` erweitert die Basisklasse um eine Priorität und ein Transitions-*Label*, und stellt somit die Grundlage aller Transitionen eines *Statecharts* bereit. Das Projekt *KIEL* unterstützt eine Vielzahl von Transitionstypen, deren Unterscheidungsmerkmale in Tabelle 2.1 gegenübergestellt sind.

Der bis hierhin beschriebene Einblick in das Modell der *KIEL*-Datenstruktur genügt für das weitere Verständnis dieser Arbeit. Im nächsten Abschnitt wird auf die Struktur der grafischen Repräsentation eines *Statecharts* eingegangen.

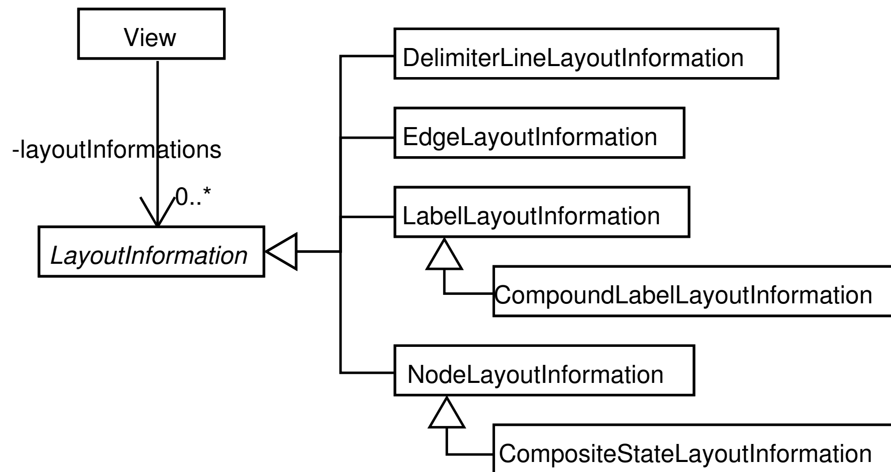


Abbildung 2.7.: Klassendiagramm der *KIEL*-Layout-Informationen

Layout-Information	grafische Komponente
`DelimiterLineLayoutinformation`	`DelimiterLine`
`EdgeLayoutinformation`	`Edge`
`CompoundLabelLayoutinformation`	`CompoundLabel`
`LabelLayoutinformation`	alle übrigen der Klasse `TransitionLabel`
`CompositeStateLayoutinformation`	`CompositeState`
`NodeLayoutinformation`	alle übrigen der Klasse `Node`

Tabelle 2.2.: Grafische Informationen und ihre zugehörigen Objekte

Die grafische Repräsentation

Das bisher beschriebene Modell eines *KIEL-Statecharts* enthält wegen des MVC-Konzepts keinerlei grafische Informationen. Diese Daten werden in separaten Ansichten (`View`) gespeichert.

Abbildung 2.7 zeigt ein Klassendiagramm der grafischen Zusammenhänge. Die Grundlage der grafischen Repräsentation eines *Statecharts* bildet die Ansichtsklasse (`View`). Diese vereint alle Layout-Informationen (`LayoutInformation`) der einzelnen Komponenten und stellt somit eine mögliche Ansicht des *Statecharts* dar. Über den Zusammenhang von grafischer Komponente und Layout-Information gibt Tabelle 2.2 einen Überblick.

Die Konfigurationen

Neben dem Modell und der Ansicht eines *Statecharts* gibt es noch das Datenstrukturelement der Konfigurationen (`Configurations`). Eine Konfiguration spiegelt den Status einer Simulation wieder, indem sie aufzeigt welche Zustände gegenwärtig aktiv sind. Dabei wird der Vaterzustand eines aktiven Zustands nicht als aktiv markiert.

Wie in dem Abschnitt über Zustände und Pseudozustände bereits erwähnt wurde, kann sich die Simulation eines *Statecharts* in einem stabilen oder einem labilen Zustand befinden. Dies hängt von den gerade aktiven Zuständen ab. In einer stabilen Konfiguration sind keine `PseudoStates` aktiv. Für die Darstellung eines *Dynamischen Statecharts* wird während der Simulation, abhängig von der zugrundeliegenden Konfiguration, eine eigene Ansicht generiert.

2.2.3. Die Module

Abbildung 2.3 lässt erkennen, dass sich der *Controller*-Teil des MVC-Konzepts im Projekt *KIEL* aus einer Vielzahl von Modulen zusammensetzt. In diesem Abschnitt werden die einzelnen Module kurz vorgestellt.

Browser

Der `Browser` übernimmnt die Darstellung einer Ansicht des *Statecharts*. Darüber hinaus stellt er eine grafische Benutzeroberfläche zur Ansteuerung des Simulators bereit, in der sowohl Signale gesetzt als auch zurückgenommen werden können. Desweiteren lässt sich der Simulator für die Berechnung des nächsten *Ticks* ansteuern.

Für ein besseres intuitives Verständnis des Systemverhaltens werden die zu diesem Zeitpunkt relevanten Teile einer Simulation besonders hervorgehoben. Dazu zählt unter anderem die Markierung der derzeit aktiven Zustände. Beim Wechsel von der Ansicht A zu der Ansicht B eines *Statecharts* gleiten die Objekte zu ihren neuen Positionen. Dieses Verfahren wird auch „*gleitende Strukturveränderung*“ genannt.

Die Entwicklung des Moduls `Browser` ist zum gegenwärtigen Zeitpunkt noch nicht abgeschlossen, eine detailliertere Beschreibung liegt somit nicht vor.

ConfigMgr

Dieses Modul berechnet alle Konfigurationen eines *Statecharts*, die während der Simulation erreicht werden können. Mit Hilfe dieses Wissens können im Vorfeld alle daraus resultierenden Ansichten vom Layouter berechnet werden.

Editor

Mit dem `Editor` können *Statecharts* neu erstellt bzw. bearbeitet werden. Dabei zeichnet sich dieses Modul durch die folgenden Merkmale aus:

- syntaxgerichtetes Editieren von *Statecharts*,
- innovative und in dem Kontext von *Statecharts* bisher nicht angewandte Bearbeitunghilfen,
- gleitende Strukturveränderung bei automatischen Layout-Mechanismen,
- Abspielen der Bearbeitungshistorie sowie

- eine mehrschichtige Übersichtsdarstellung.

Für weiterführende Arbeiten auf dem Gebiet des Editierens von *Statecharts* ist die zu diesem Modul gehörige Arbeit Lüpke [29] zu empfehlen.

FileInterface

Dieses Modul ermöglicht das Einlesen bzw. Schreiben von proprietären Dateiformaten bereits existierender Modellierungswerkzeuge wie *Esterel*, *Esterel Studio*, *Matlab* und *Statemate*. Zur Zeit werden bereits folgende Schnittstellen unterstützt:

- lesender Zugriff auf *Esterel Studio* Dateien (Wischer [63]) und
- lesender Zugriff auf *Esterel* Dateien (in der Entwicklung).

Weitere Schnittstellen sowie der schreibende Zugriff sind bereits in Planung.

Layouter

Unter Einbindung des Moduls `Layouter` werden die Objekte eines *Statecharts* nach optischen und funktionalen Gesichtspunkten angeordnet. Die Grundlagen dieses Moduls wurden im Rahmen dieser Arbeit entwickelt. Für das automatische Layout von *Statecharts* werden mehrere Verfahren bereitgestellt, die in Kapitel 4 näher vorgestellt werden.

Simulator

Mit Hilfe dieses Moduls lässt sich das Systemverhalten eines *Statecharts* simulieren. Grundsätzlich können verschiedene Simulationsmodelle eingebunden werden. Zur Zeit stehen bereits folgende Simulationsumgebungen zur Auswahl:

Esterel Studio Die Simulation basiert auf dem Verhalten von SyncCharts, wie sie von *Esterel Studio* benutzt werden. Dieser Simulator ist im Rahmen von Ohlhoff [35] entwickelt worden.

Matlab Um *Matlab*-Modelle im Rahmen des Projekts *KIEL* zu simulieren, wird eine von *Matlab* bereitgestellte Schnittstelle benutzt. Mit Hilfe dieser Schnittstelle lassen sich *Statecharts* erstellen und simulieren, ohne die beiliegende grafische Benutzeroberfläche zu nutzen. Eine erste Arbeit zu diesem Verfahren ist bei Täubrich [48] zu finden.

2.2.4. Die grafische Benutzeroberfläche

Die im vorangegangenen Abschnitt beschriebenen Module sind auf Grund ihrer starken Kapselung als ein entkoppelter Verband zu betrachten, obwohl alle Module auf derselben Datenstruktur arbeiten. Die grafische Benutzeroberfläche (GUI) vereint die einzelnen Module zu einer Anwendung. Hierfür übernimmt sie die Steuerung

des Daten- und des Kontrollflusses, und lässt die einzelnen Module auf diese Weise miteinander interagieren.

Darüber hinaus vereint die GUI die grafischen Module (`Browser`, `Editor`) in einer einheitlichen Umgebung, so dass die Module für den Benutzer zu einer gesamtheitlichen Anwendung verschmelzen.Die Abbildungen 2.8 und 2.9 zeigen die jeweiligen Arbeitsmodi der *KIEL*-Anwendung.

Abbildung 2.8.: Die *KIEL*-Anwendung im `Browser`-Modus

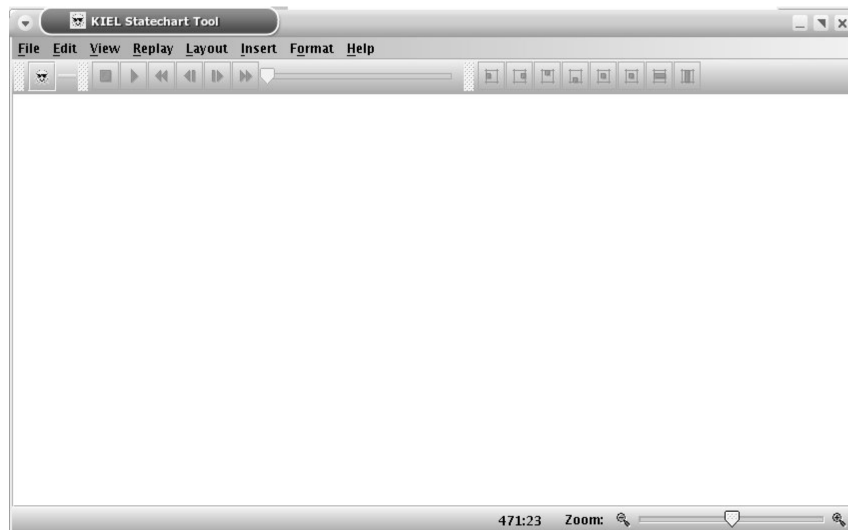


Abbildung 2.9.: Die *KIEL*-Anwendung im `Editor`-Modus

3. Analyse verwandter Arbeiten

Wie in Abschnitt 2.1 bereits beschrieben, können *Statecharts* in einer abstrakten Sichtweise als gerichtete Graphen betrachtet werden. Aus diesem Grund liegt es nahe, für das automatische Layout von *Statecharts* Arbeiten mit ähnlichen Zielen aus dem Gebiet des Graph-Zeichnens zu betrachten.

In diesem Kapitel wird zunächst ein Überblick über die gebräuchlichsten Graph-Layout-Verfahren (Abschnitt 3.1) gegeben. Im Anschluss daran werden einige Graph-Zeichen-Programme vorgestellt und dahingehend untersucht, ob sie sich für das Erstellen von *Statechart*-Layouts einsetzen lassen (Abschnitt 3.2). In Abschnitt 3.3 wird das Projekt *GraphViz* näher beschrieben, das für den Einsatz im Projekt *KIEL* im weiteren Verlauf dieser Arbeit verwendet wird. Abschließend werden die Ergebnisse dieses Kapitels, das als eines der Hauptaugenmerke zu sehen ist, in Abschnitt 3.4 zusammengefasst.

3.1. Generelle Verfahren zum Layout von Graphen

Auf dem Gebiet des Graph-Zeichnens sind eine Vielzahl von Verfahren entwickelt worden, die in den unterschiedlichsten Bereichen zum Einsatz kommen und an die jeweiligen Voraussetzungen angepasst wurden. In diesem Abschnitt wird eine Auswahl von grundlegenden Layout-Varianten vorgestellt.

Als Einstieg wird ein Verfahren zur Planarisierung von Graphen (Abschnitt 3.1.1) beschrieben. Dieses Verfahren ist als eine Art Rahmen um ein Layout-Verfahren zu verstehen, das ausschließlich auf planaren Graphen arbeitet. Zu dieser Klasse von Layout-Verfahren zählt das orthogonale Layout aus Abschnitt 3.1.2. Generell lassen sich die vorgestellten Layout-Verfahren danach klassifizieren, auf welche Weise sie die Knoten des Graphen platzieren.

- Das orthogonale Layout ordnet die Knoten auf einem Gitternetz an, das aus jeweils parallelen X- und Y-Ebenen besteht.
- Sowohl das Baum-Layout (Abschnitt 3.1.3) als auch das ebenenbasierte Layout (Abschnitt 3.1.4) reduzieren diese Platzierungsvorschrift, indem sie auf eine der beiden Ebenenrichtungen verzichten.
- Der *Force-Directed*-Ansatz (Abschnitt 3.1.5) verzichtet ganz auf Vorschriften dieser Art und ordnet die Knoten auf der freien Ebene an.

3.1.1. Layout von Graphen mittels Planarisierung

Die Planarisierung von Graphen ist kein Layout-Verfahren im eigentlichen Sinne. Allerdings stellt es die Grundlage vieler Layout-Algorithmen dar, da diese überwiegend von einem planaren Graphen als Eingabe ausgehen. Die grundlegende Idee für Methoden zur Planarisierung von Graphen stammt von Tamassia et al. [47].

Diese Methode nimmt einen beliebigen Graph G als Eingabe und wandelt diesen in einen planaren Graphen H um. Anschließend wird für den Graphen H von einem beliebigen Layout-Verfahren ein Layout erstellt, das den Graphen H^* ausgibt. In einem abschließenden Schritt wird dieser Graph in den ursprünglichen Graphen G^* mit erstelltem Layout zurückgewandelt.

$$\begin{array}{ccc} G & \xrightarrow{g} & G^* \\ {\scriptstyle h}\downarrow & & \uparrow{\scriptstyle h^{-1}} \\ H & \xrightarrow{g'} & H^* \end{array}$$

Abbildung 3.1.: Vorgehen der Planarisierungsmethode

Um das genaue Vorgehen der vorgestellten Methode zu beschreiben, lässt sich diese in die folgenden vier Phasen unterteilen, wie Abbildung 3.1 mittels eines kommutativen Diagramms verdeutlicht:

1. Erstellen eines beliebigen Graphen-Layouts (G).
2. An die Stellen, an den Kantenkreuzungen existieren, werden künstliche Knoten platziert. Die sich kreuzenden Kanten werden am Kreuzungspunkt durchtrennt und die daraus resultierenden Kantenenden mit dem künstlichen Knoten verbunden (h).
3. Das Zeichnen des Hilfsgraphen H wird vorgenommen. Hier kommt ein beliebiger Layout-Algorithmus zum Einsatz, der auf planaren Graphen arbeitet (g').
4. Die künstlichen Knoten werden resubstituiert und somit der ursprüngliche Graph G^* wiederhergestellt (h^{-1}).

Darüber hinaus wird versucht, die Zahl der Kantenkreuzungen zu minimieren. Dafür werden einzelne Kanten entfernt und anschließend so wieder hinzugefügt, dass der Graph möglichst wenige Kreuzungen enthält.

3.1.2. Orthogonales Layout

Das orthogonale Layout zeichnet sich dadurch aus, dass sich die Kanten ausschließlich aus horizontalen und vertikalen Teilstücken zusammensetzen. Ein typisches Anwendungsgebiet findet sich beim Layout von Schaltplänen.

Eine oftmals genutzte Methode ist der von Batini et al. [3] vorgestellte *Topology-Shape-Metrics*-Ansatz. Dieser planarisiert zunächst den Graphen mittels des in Abschnitt 3.1.1 vorgestellten Planarisierungsverfahrens. Anschließend wird dieser Hilfsgraph orthogonalisiert, in den ursprünglichen Graphen zurückgewandelt und abschließend gezeichnet.

Für das Orthogonalisieren setzt Tamassia [46] in seiner Arbeit planare Graphen voraus, deren Knoten mit nicht mehr als vier Kantenenden verbunden sind. Dies wird deshalb zwingend vorausgesetzt, da die Knoten auf einem Gitternetz platziert werden und die Kanten entlang der Gitternetzlinien verlaufen. Hierbei wird jedes Teilstück nur einfach belegt. Somit kann ein Knoten maximal mit vier Kantenenden verbunden sein. Abbildung 3.2 verdeutlicht die Beschränkung des von Tamassia vorgestellten Verfahrens. Der grau markierte Knoten kann nur mit Kanten verbunden sein, die auf den mit e_n $(n = 1, \ldots, 4)$ beschrifteten Gitternetzlinien liegen.

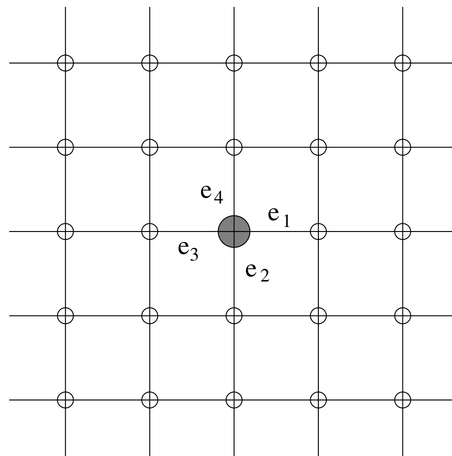


Abbildung 3.2.: Platzierung auf einem Gitternetz

Ein Graph in einem typischen orthogonalen Layout ist in Abbildung 3.3 dargestellt. Dabei ist zu erkennen, dass die Knoten auf einer Gitterstruktur angeordnet sind und die Kanten entlang dieses Gitters verlaufen.

Fößmeier und Kaufmann [14] erweitern diesen Ansatz für beliebige Graphen und stellen in ihrer Arbeit das *Basic-Kandinsky*-Zeichenverfahren vor. Dabei werden parallel verlaufende Kanten übereinander bzw. direkt nebeneinander gezeichnet. Battista et al. [4] ermöglichen durch ihre Arbeit die Verwendung von Knoten, deren Breiten und Höhen definierbar sind.

3.1.3. Baum-Layout

Ein typisches Beispiel für ein Baum-Layout ist der Familienstammbaum, der die Abstammung der einzelnen Familienmitglieder darstellt. Allgemein gilt, dass die Knoten des Graphen, für den ein solches Layout erstellt werden soll, alle eine direkte Vater-Kind-Beziehung haben müssen. Jede Kante des Graphen steht für eine

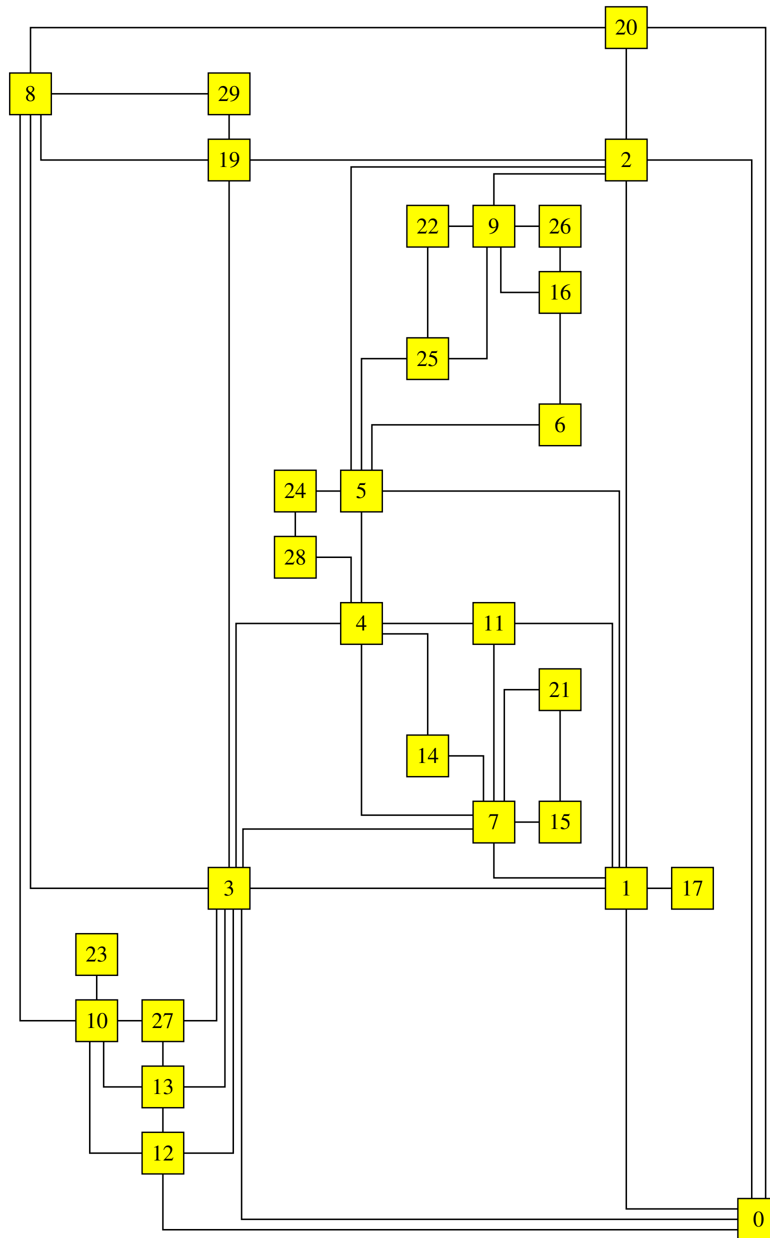


Abbildung 3.3.: Ein typisches orthogonales Layout (Quelle: [26])

solche Beziehung. Darüber hinaus sind keine weiteren Kanten zulässig, was die Anwendbarkeit auf beliebige Graphen stark einschränkt.

Generell werden die Knoten eines Baums so angeordnet, dass zunächst der Wurzelknoten ermittelt und dieser danach auf der ersten Ebene platziert wird. Die Kindsknoten werden auf einer dazu parallelen Ebene angeordnet. Diese Kindsknoten werden im darauffolgenden Schritt als Wurzelknoten eines Teilbaums betrachtet. Der Algorithmus terminiert, wenn alle Knoten platziert sind.

Grundlage des Baum-Layouts ist die Arbeit von Reingold und Tilford [38], die einen Algorithmus zum Zeichnen von binären Bäumen vorstellt. Walker, II [58] erweiterte dieses Verfahren für beliebige Bäume. Anschließend verbesserten Buchheim et al. [6] nochmals die Laufzeit dieses Algorithmus. Eades [9] hat die Kindsknoten nicht auf parallelen Ebenen, sondern auf konzentrischen Kreisen um den Wurzelknoten angeordnet.

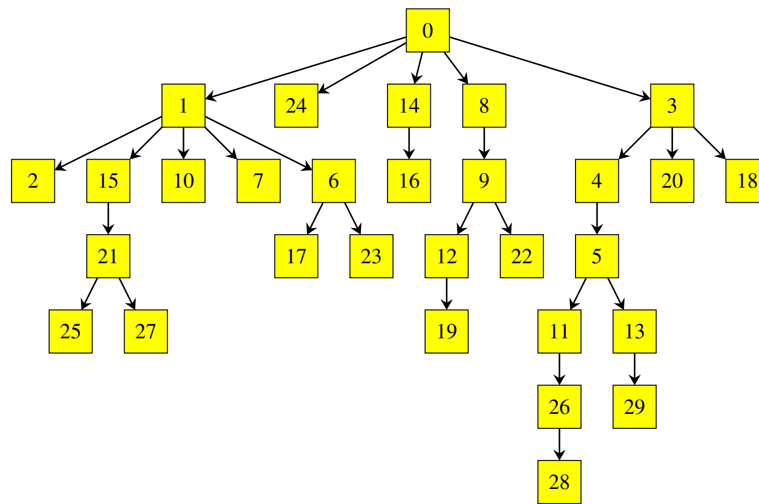


Abbildung 3.4.: Ein typisches Baum-Layout (Quelle: [26])

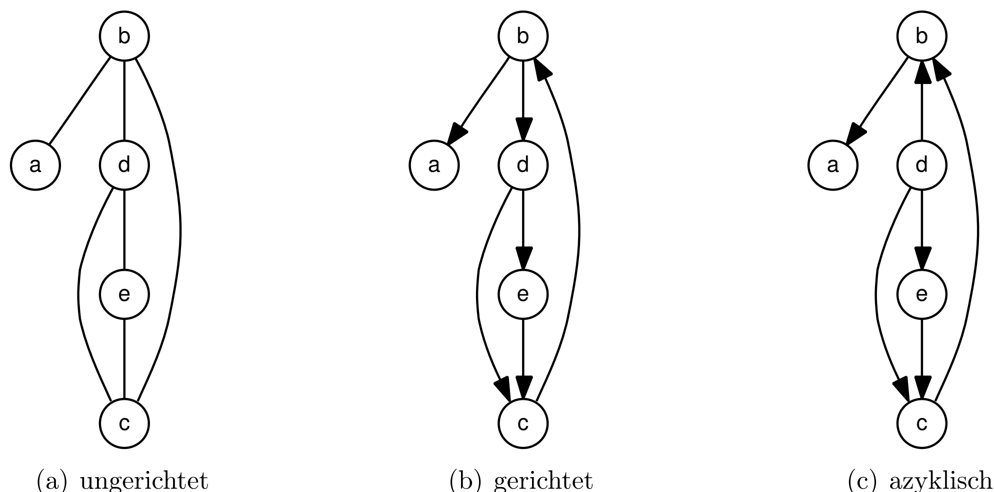
Abbildung 3.4 zeigt einen Graphen im typischen Baum-Layout. Dabei ist zu erkennen, dass die Kanten alle in der gleichen Richtung verlaufen, in diesem Fall verlaufen die Kanten alle von der Wurzel weg. Der umgekehrte Weg wäre für ein Baum-Layout jedoch auch denkbar. Darüber hinaus verdeutlicht diese Art der Knotenanordnung die Vater-Kind-Beziehung zwischen jedem verbundenen Knotenpaar. So hat beispielsweise der Knoten 21 den Vaterknoten 15 und die Kinderknoten 25 und 27.

3.1.4. Ebenenbasiertes Layout

Das ebenenbasierte Layout ordnet die Knoten eines Graphen, ebenso wie das Baum-Layout, in parallelen Ebenen an. Allerdings erweitert es das in Abschnitt 3.1.3 beschriebene Verfahren dahingehend, dass alle Graphen nach diesem Verfahren verarbeitet werden können. Als Meilenstein auf diesem Gebiet gilt die Arbeit von Sugiya-

ma et al. [44]. Deshalb sind ebenenbasierte Layouts auch als Layouts im *Sugiyama-Stil* bekannt.

Die erste Variante des Verfahrens beschränkt sich auf die Betrachtung von azyklischen, gerichteten Graphen. Durch die Tatsache, dass sich jeder Graph in einen azyklischen gerichteten Graphen überführen lässt, hat die Bedeutung dieses Layout-Verfahrens zusätzlich verstärkt. Eine solche Umwandlung lässt sich durch das Umkehren einzelner Kantenrichtungen bzw. durch das Festlegen auf eine Kantenrichtung durchführen.



(a) ungerichtet (b) gerichtet (c) azyklisch

Abbildung 3.5.: Wandlung eines Graphen: ungerichtet → azyklisch gerichtet[1]

Abbildung 3.5 verdeutlicht diese Vorgehensweise anhand eines Beispiels. Der dargestellte Graph ist im linken Teil 3.5(a) noch ungerichtet. Im mittleren Teil 3.5(b) sind den Kanten beliebige Richtungen zugewiesen worden, der Graph ist somit gerichtet, aber nicht unbedingt azyklisch. Durch das Umkehren der Kante (b → d) wird der Graph azyklisch.

Der in Abbildung 3.6 dargestellte Graph zeigt ein typisches Sugiyama-Layout. Die einzelnen Ebenen sind klar zu erkennen und zudem seitlich mit $L_{1\ldots}$ markiert. Bei näherer Betrachtung ist zu erkennen, dass bei diesem Layout-Verfahren, im Gegensatz zum Baum-Layout-Verfahren aus Abschnitt 3.1.3, keine eindeutige Vater-Kind-Beziehung zwischen den Knoten existiert. Ansonsten hätte beispielsweise Knoten 3 zwei Väter, nämlich Knoten 1 und Knoten 2. Somit können Kanten zwischen beliebigen Knoten existieren.

Generell lassen sich alle Layout-Verfahren dieser Art in drei Phasen unterteilen:

1. Jedem Knoten wird eine Ebene zugewiesen.
2. Die Knoten einer Ebene werden relativ zueinander angeordnet, so dass die Darstellung des Graphen wenige Kantenkreuzungen enthält.

[1]Diese und alle weiteren Graphen wurden mittels des *GraphViz*-Programms *Dot* erstellt.

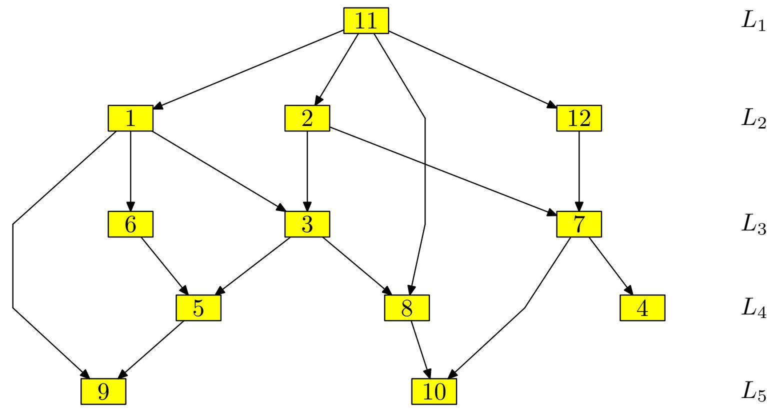


Abbildung 3.6.: Ein typisches Layout im Sugiyama-Stil (Quelle: [26])

3. Die Koordinaten der Knoten werden festgelegt, wobei die Y-Koordinate für alle Knoten einer Ebene gleich ist. Die X-Koordinate wird in Abhängigkeit von der relativen Position innerhalb der Ebene vergeben.

Die verschiedenen Variationen dieses Verfahrens unterscheiden sich rein optisch hauptsächlich durch die Darstellung der Kanten. Das Spektrum reicht von einfachen Direktverbindungen über Polygonzüge bis hin zu *Bezier-Splines*. Die letztgenannte Variante wird von dem in Abschnitt 3.3.3 vorgestellte *Dot*-Layout-Algorithmus genutzt. In diesem Zusammenhang wird das hier vorgestellte Verfahren noch einmal im Detail beschrieben.

3.1.5. *Force-Directed*-Layout

Beim *Force-Directed*-Layout wird ein Graph als ein physikalisches System betrachtet. Die Knoten entsprechen elektrisch geladenen Teilchen, die sich gegenseitig abstoßen. Die Kanten entsprechen Federn, die die jeweiligen Knoten zusammenziehen. Das Layout-Verfahren besteht darin, die Energie eines solchen Systems zu einem lokalen Minimum zu führen.

Wegen der „eingebauten“ Federn ist dieses Verfahren auch als *Spring-Embedder* bekannt. Die Platzierung eines Knoten ergibt sich aus den Kräften, die auf ihn wirken. Diese Kräfte berechnen sich aus seiner relativen Position zu den Nachbarknoten und aus der Entfernung der Knoten, mit denen er durch eine Kante verbunden ist.

Diese Vorgehensweise wurde erstmals von Eades [8] vorgestellt. Sie ist in einer Vielzahl von Arbeiten verfeinert und erweitert worden. Die Arbeiten von Fruchterman und Reingold [16] und Frick et al. [15] unterscheiden sich im Wesentlichen nur durch die Definition der wirkenden Kräfte. Die Arbeit von Sugiyama und Misue [43] hingegen spezialisiert sich auf die Ausrichtung von gerichteten Graphen. Eades et al. [10] haben dieses Verfahren für das Layout von hierarchischen (gruppierbaren) Graphen angepasst. Ein experimenteller Vergleich der verschiedenen *Force-Directed*-Layout-Verfahren ist in der Arbeit von Brandenburg et al. [5] zu finden. Der von

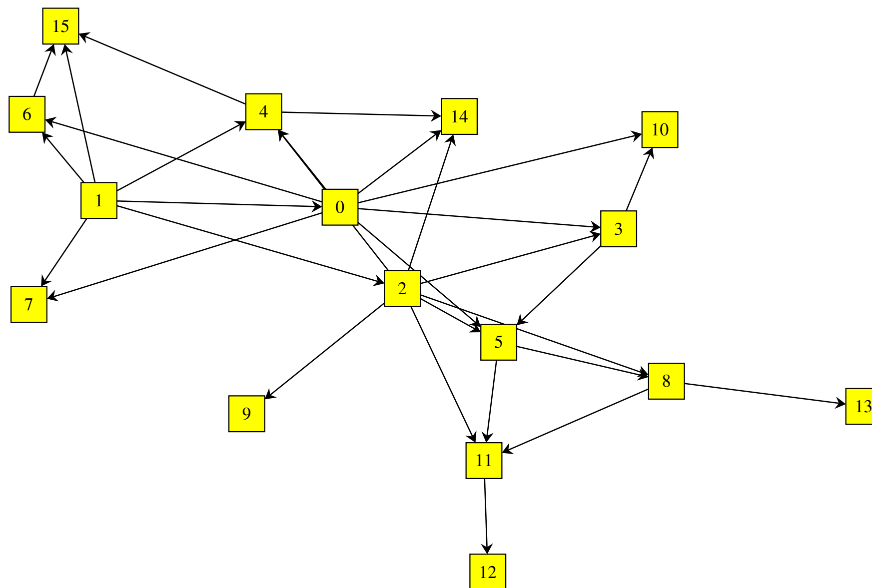


Abbildung 3.7.: Ein typisches *Force-Directed*-Layout (Quelle: [26])

Kamada und Kawai [27] vorgestellte Ansatz ist in die Entwicklung des *GraphViz*-Programms *Neato* eingeflossen.

Abbildung 3.7 zeigt einen Graphen in einem typischen *Force-Directed*-Layout. Insbesondere ist zu erkennen, dass die Kanten zwischen zwei Knoten auf der direkten Verbindungslinie zueinander verlaufen, und somit den direkten Federweg beschreiben. Des Weiteren zeigen die unterschiedlichen Abstände zwischen den Knoten 11 und 12 und den Knoten 8 und 13 an, dass deren jeweilige „Knotenabstoßung“ geringer ist bzw. die jeweiligen „Kantenfederkräfte“ größer sind. Dies spiegelt die Einstellmöglichkeiten des verwendeten Layout-Verfahrens wieder.

3.2. Auswahl eines geeigneten *Frameworks*

Nachdem in Abschnitt 3.1 einige grundlegende Layout-Verfahren vorgestellt wurden, beschäftigt sich dieser Abschnitt mit deren Einsatz in verschiedenen Graph-Zeichen-Programmen. Im Rahmen dieser Arbeit galt es, ein geeignetes Rahmenwerk (*Framework*) zu finden, dass für das automatische Layout von *Statecharts* innerhalb des Projekts *KIEL* eingesetzt werden kann.

Dafür mussten zunächst die Anforderungen an ein solches *Framework* abgesteckt werden (Abschnitt 3.2.1). Anschließend werden verschiedene Graph-Zeichen-Programme vorgestellt und anhand der in Abschnitt 3.2.2 gestellten Anforderungen bewertet. Die Werkzeuge, die diesen Anforderungen genügen, stehen sich in Abschnitt 3.2.3 im direkten Vergleich gegenüber.

3.2.1. Anforderungen

Bei der Beschäftigung mit dem Zeichnen von Graphen kann auf eine Vielzahl von existierenden Arbeiten zurückgegriffen werden. Aus diesem Grund werden im Folgenden einige Graph-Zeichen-Programme dahingehend untersucht, ob sie für das automatische Layout von *Statecharts* innerhalb des Projekts *KIEL* eingesetzt werden können. Dafür ist zunächst eine Anforderungsliste zu erstellen.

Obwohl sich ein *Statechart* in einer abstrakten Sicht als ein Graph betrachten lässt, sind an das Zeichnen von *Statecharts* in manchen Punkten andere Anforderungen zu beachten. Neben diesen eher theoretischen Gesichtspunkten sind auch noch praxisnahe und lizenzrechtliche Bedingungen zu stellen. In der nachstehenden Aufzählung sind die Anforderungen an das zu verwendende *Framework* aufgelistet.

Automatisches Layout von Graphen

Das automatisierte Layout von Graphen ist eine der Grundvoraussetzungen an ein Werkzeug, die bereits in die Auswahl der beschriebenen Programme eingeflossen ist.

Definierbarkeit von Form und Größe der Knoten

In der mathematischen Definition eines Graphen sind keine grafischen Informationen enthalten, lediglich Knoten und Kanten. Somit können die Knoten auch als Punkte dargestellt werden. Die Knoten (Zustände) eines *Statecharts* hingegen haben sowohl eine Ausdehnung als auch unterschiedliche Formen (z. B. Kreise, Rechtecke, ...).

Beschriftung der Kanten

Auch in diesem Punkt sieht die mathematische Definition eines Graphen keine Beschriftung der Kanten (*Label*) vor. Für das *Statechart*-Layout ist die Berücksichtigung der *Label* unerlässlich, weil ein nachträgliches Einfügen der *Label* zu Überschneidungen mit anderen Objekten führen kann.

Hierarchie innerhalb eines Knoten

Im Umfeld des Graphen-Layout versteht sich der Hierarchiebegriff als eine Anordnung in unterschiedlichen Ebenen (siehe Abschnitt 3.1.4). Im Gegensatz dazu ist der Hierarchiebegriff im *Statechart*-Umfeld als Verfeinerung einzelner Zustände zu sehen (siehe Abschnitt 2.1). Diese Art des Hierarchieverständnisses ist in der mathematischen Definition eines Graphen nicht vorgesehen. Die Knoten eines Graphen können keine anderen Knoten beinhalten.

Algorithmenauswahl

Eine breit gefächerte Auswahl an bereitgestellten Layout-Algorithmen eröffnet ein ebenso breites Spektrum an *Statechart*-Layouts. Dieses würde nach einer einmaligen Anbindung des Werkzeugs dem Projekt *KIEL* zur Verfügung stehen.

Verwendete Plattform

Da das Projekt *KIEL* in *Java* entwickelt wurde und somit plattformübergreifend einsetzbar ist, sollte diese Plattformunabhängigkeit weitgehend erhalten bleiben. Die Unterstützung des Betriebssystems *Linux* stellt hier eine harte Forderung dar.

Ein-/Ausgabeformate bzw. Zugriffsverfahren

Das Werkzeug sollte gut nutzbare Zugriffsmöglichkeiten bieten, entweder über direkte Bibliotheksschnittstellen oder proprietäre Dateiformate. Ein Auslesen der Layout-Informationen aus einer erstellten Grafikdatei im `JPEG`- oder `GIF`-Format wäre dabei eine schlechte Zugriffsmethode. Gleiches gilt für die Eingabe über eine grafische Benutzeroberfläche.

Verfügbarkeit

Generell ist Werkzeugen, die kostenfrei eingesetzt werden dürfen, gegenüber kommerziellen Produkten der Vorzug zu geben. Die Verfügbarkeit des Quellcodes ist ebenfalls positiv zu bewerten, da sich solche Projekte meist auf eine breitere Anwenderbasis stützen können.

Durchführungsgeschwindigkeit

Die Anzahl der Knoten eines Graphen betreffend sind für das Layout von *Statecharts* sicherlich nicht die gleichen Komplexitätsanforderungen zu stellen wie an das Zeichnen von Graphen. Da das Layout eines *Statecharts* allerdings während der Simulation oder des Editierens berechnet werden soll, sind Rechenzeiten von mehreren Minuten nicht tolerierbar.

Aktualität

Die Aktualität eines Projekts ist ein Indikator dafür, dass dieses gepflegt und weiterentwickelt wird, und somit mögliche Fehler behoben werden. Ein veraltetes Projekt als Grundstein für das Layout von *Statecharts* zu wählen sollte aus diesen Gründen vermieden werden.

Anwenderbasis

Ebenso wie die Aktualität ist eine breite Anwenderbasis eines Werkzeugs eine gute Voraussetzung für den weiteren Einsatz. Je mehr Projekte sich auf das Werkzeug stützen, desto wahrscheinlicher werden mögliche Fehler entdeckt und können somit behoben werden.

Im folgenden Abschnitt werden einige Graph-Zeichen-Werkzeuge vorgestellt und anhand der soeben erstellten Anforderungen bewertet. Eine feste Prioritätsliste der Anforderungen wurde zu diesem Zweck nicht erstellt. Da es zur Zeit kein Werkzeug gibt, das alle Anforderungen zufriedenstellend erfüllt, sind die einzelnen Anforderungen vielmehr gegeneinander abzuwiegen.

3.2.2. Überblick

In diesem Abschnitt werden zunächst diverse Projekte vorgestellt, die Programme zum Zeichnen von Graphen bereitstellen. Die Grundlage für diesen Überblick stellen hierfür das Buch von Jünger und Mutzel [26] und die Arbeit von Willhalm [60], in denen gute Übersichten über Graph-Zeichen-Werkzeuge gegeben werden. Darüber hinaus werden die vorgestellten Projekte anhand der in Abschnitt 3.2.1 gestellten Anforderungen bewertet. Die Programme, die für den Einsatz im Projekt *KIEL* geeignet sind, werden in Abschnitt 3.2.3 noch einmal im direkten Vergleich gegenübergestellt.

Im Folgenden wird zu jedem Projekt eine kurze Beschreibung gegeben, gefolgt von einer stichpunktartigen klaren Hervorhebung ihrer Vor- und Nachteile bezüglich der gestellten Anforderungen.

WilmaScope

WilmaScope [52] ist eine *Java*-Applikation zur Visualisierung von dreidimensionalen Animationen und dynamischen Graphen. Bei der Entwicklung wurde besonderer Wert auf Flexibilität, Interaktivität und Erweiterbarkeit gelegt. Zur Zeit sind sowohl verschiedene *Force-Directed*-Layout-Algorithmen implementiert als auch ein Verfahren, das das Programm *Dot* aus dem Projekt *GraphViz* (siehe unten) nutzt, um ein Layout im *Sugiyama-Stil* zu erzeugen. Alle hierbei verwendeten Layout-Algorithmen lassen sich über Parameter feinjustieren.

Vorteile:

- Da *WilmaScope* eine *Java*-Applikation ist, steht es für alle gängigen Plattformen zur Verfügung.
- Das Projekt steht wegen der Veröffentlichung unter der *Lesser General Public License (LGPL)* kostenfrei zur Verfügung.

Nachteile:

- Graphen werden dreidimensional dargestellt.
- Die Größe der Knoten lässt sich nicht beeinflussen.
- Die Transitionen können nicht mit *Statechart*-typischen Beschriftungen (*Labels*) versehen werden.

Pajek

Pajek [31] wird für die Analyse und Visualisierung von sehr großen Netzwerken benutzt. Dafür wird der *Divide-And-Conquer*-Ansatz genutzt. Dies bedeutet, dass rekursiv größere Teilnetzwerke in immer kleinere Teil zerlegt werden, die dann einzeln verarbeitet werden. Neben *Force-Directed*- und hierarchischen Layout-Algorithmen für zwei- und dreidimensionale Graphen beinhaltet *Pajek* auch Eigenvektormethoden und Blockmatrix-Darstellungen. Pajek ist eine

reine Anwendung für die Plattform *Microsoft Windows*, die aber auch via *Wine* [62] unter dem Betriebssystem *Linux* läuft.

Vorteile:

- Das Projekt steht für den nicht-kommerziellen Gebrauch kostenlos zur Verfügung.
- Viele Layout-Algorithmen sind bereits implementiert.

Nachteile:

- Die Größe und Form der Knoten lässt sich nicht frei wählen.
- Die Transitionen können nicht mit *Statechart*-typischen *Labels* versehen werden.
- Pajek lässt sich schlecht als *Framework* nutzen, da es eine eigenständige Anwendung ist.

Tulip

Tulip [2] visualisiert sehr große Graphen mit bis zu einer Million Elementen. Das Projekt nutzt verschiedene Algorithmen für die Visualisierung von Graphen. Die wichtigsten sind Graph-Zeichen-Algorithmen, Gruppierungsalgorithmen, metrische Algorithmen und Algorithmen für den Einfluss von visuellen Attributen (z. B. Farben). *Tulip* stellt unter anderem eine *C++*-Bibliothek zur freien Verfügung (Registration erforderlich).

Vorteile:

- Verschiedene Graphenalgorithmen sind bereits implementiert.
- Das Projekt steht unter der *General Public License (GPL)* zur freien Verfügung.

Nachteile:

- Die Größe und Form der Knoten lässt sich nicht frei wählen.
- Die Transitionen können nicht mit *Statechart*-typischen *Labels* versehen werden.

GraphViz

Das Projekt *GraphViz* [23] setzt sich aus mehreren Programmen zur Visualisierung und Manipulation von Graphen zusammen, die in den unterschiedlichsten Gebieten zum Einsatz kommen. Das Feld reicht von Softwareentwicklung über Datenbanken bis hin zur Bioinformatik. Alle Layout-Algorithmen basieren auf der gleichen Datenstruktur, die über eine *C++*-Bibliotheksschnittstelle oder über Kommandozeilenbefehle bereitgestellt werden.

Vorteile:

- *GraphViz* steht als *Open-Source*-Projekt unter der *Common Public License (CPL)* zur freien Verfügung.
- Alle gängigen Plattformen werden unterstützt.
- Knotenform und -größe sind parametrierbar.
- Kanten können mit Beschriftungen versehen werden.
- Der Zugriff ist über Bibliotheksschnittstellen und Kommandozeilenbefehle möglich.

Nachteile:

- Es ist keine *Statechart*-typische Hierarchie der Knoten vorgesehen.

AGD (*Algorithms for Graph Drawing*)

AGD [7] versteht sich als eine Art Brücke zwischen Theorie und Praxis, indem es viele theoretische Ansätze zum Graphen-Layout implementiert und in ihr Projekt einbettet. Somit bietet es eine Vielzahl an Algorithmen für das Zeichnen von Graphen. Das Spektrum reicht vom Zeichnen planarer Graphen über orthogonales Layout bis hin zu Layouts im Sugiyama-Stil. Das Projekt *AGD* setzt sich aus mehreren plattformabhängigen *C++*-Bibliotheken und den dazugehörigen Demoprogrammen zusammen, die für alle gängigen Plattformen übersetzt worden sind.

Vorteile:

- Für den nicht-kommerziellen Einsatz an akademischen Einrichtungen steht das Projekt kostenfrei zur Verfügung (Registration erforderlich).
- Viele der theoretisch fundierten Layout-Algorithmen sind bereits implementiert.
- Die gängigsten Plattformen werden unterstützt.

Nachteile:

- Da sich das Projekt als Graph-Zeichen-Werkzeug versteht, ist keine *Statechart*-typische Hierarchie der Knoten vorgesehen.

yFiles

yFiles [64] ist eine umfangreiche *Java*-Klassenbibliothek für die Analyse, Visualisierung und das automatische Layout von Graphen. Das Produkt bietet vielfältige Einsatzmöglichkeiten, wie zum Beispiel Software-Entwicklung, Datenbank-Management und Prozessmodellierung. *yFiles* hat einen modularen Aufbau, der unter anderem eine Layout-Komponente enthält. Diese stellt diverse Layout-Varianten bereit, wie beispielsweise Baum-Layout, *Force-Directed*-Layout und orthogonales Layout.

Vorteile:

- Eine breite Basis an Layout-Verfahren wird bereitgestellt.
- Die Plattformunabhängigkeit ist durch die Implementierung in *Java* gewährleistet.
- Knotenform und -größe sind parametrierbar.
- Kanten können mit Beschriftungen versehen werden.

Nachteile:

- Das Produkt ist auch für den nicht-kommerziellen Gebrauch gebührenpflichtig, allerdings wird auch eine kostenfreie Version zur Evaluierung bereitgestellt.

GDS (*Graph Drawing Server*)

Der *GDS* [22] ist ein webbasierter Graph-Zeichen- und Übersetzungsservice. Dafür werden Daten, die den zu zeichnenden Graphen beschreiben, und der zu verwendende Zeichenalgorithmus an den *GDS* gesandt. Das Layout und Zeichnen wird seitens des *Servers* vorgenommen, auf der zum Beispiel Algorithmen des oben beschriebenen *AGD* eingebunden sind. Auf diese Weise vereinigt das Projekt unter anderem einfache Handhabung, Plattformunabhängigkeit und Flexibilität.

Vorteile:

- Der *GDS* vereint viele unterschiedliche Graph-Zeichen-Werkzeuge und bietet somit eine Vielzahl an Layout-Verfahren.
- Durch den *Client/Server*-Ansatz ist eine plattformunabhänige Benutzung möglich.

Nachteile:

- Die Verwendung des *GDS* erfordert wegen des *Client/Server*-Konzepts einen permanenten Internetzugang.

BioPath

BioPath [56] ist ein webbasiertes Werkzeug für die Visualisierung von biochemischen Reaktionspfaden. Das Layout basiert auf einem Layout-Algorithmus im Sugiyama-Stil, der um ein Gruppierungsverfahren erweitert wurde. Dieses Gruppierungsverfahren lässt während der Layout-Berechnung einzelne Untergraphen zu einem gemeinsamen Knoten verschmelzen und berechnet deren Inhalt separat. Durch das Verschmelzen wird erreicht, dass die Knoten einer Ebene ähnliche Größen haben. Aus diesem Vorgehen ergibt sich eine bessere Ausnutzung der einzelnen Ebenen, da somit weniger „leere“ Flächen entstehen. Die Umsetzung der *BioPath-Server*-Anwendung stützt sich auf *Java*- und *C++*-Teile. Der Fokus des Projekts wurde hierbei auf die dynamische Visualisierung gelegt.

Vorteile:

- Sehr gute Erweiterung des ebenenbasierten Layout-Verfahrens im Sugiyama-Stil.
- Durch den *Client/Server*-Ansatz ist eine plattformunabhänige Benutzung möglich.

Nachteile:

- Die Verwendung des Projekts *BioPath* erfordert wegen des *Client/-Server*-Ansatzes einen permanenten Internetzugang.
- Das zugrundeliegende Layout-Verfahren ist zu sehr auf die Visualisierung von biochemischen Reaktionspfaden spezialisiert.

DBdraw

DBdraw [53] erstellt ein automatisches Layout der Tabellen einer relationalen Datenbank. Dabei werden die besonderen Anforderungen berücksichtigt, die an das Darstellen von solchen Diagrammen gestellt werden. Die Knoten symbolisieren dabei jeweils eine Tabelle, und werden als eine vertikale Anordnung des Namens gefolgt von der Attributsliste dargestellt. Bei der Platzierung der Kanten müssen diese von solchen Attributen aus- bzw. eingehen. Für die Berechnung des Layouts wird ein Algorithmus zur orthogonalen Anordnung verwendet.

Vorteile:

- Das Projekt steht für den nicht-kommerziellen Einsatz kostenlos zur Verfügung.
- *DBdraw* stellt eine leicht zu benutzende Programmierschnittstelle (API) bereit.

Nachteile:

- Das verwendete Layoutverfahren ist zu spezialisiert, da es als Eingabe eine *Microsoft-Access*-Datenbank erwartet.
- Das Projekt ist nur für das Betriebssystem *Microsoft Windows* erhältlich.

GoVisual

GoVisual [36] bietet Software-Bibliotheken zum automatischen Layout von UML-Klassendiagrammen an, die in eigene Entwicklungen eingebunden werden können. Für den Zugriff auf die Bibliotheken werden Schnittstellen für *C++*, *Java* über *JNI*, *.NET* und *COM* bereitgestellt. Zu den implementierten Layout-Varianten zählen unter anderem ein Baum-Layout, ein ebenenbasiertes und ein orthogonales Layout.

Vorteile:

- Eine Version zur Evaluierung ist kostenfrei zu erhalten.
- Das Softwarepaket ist für alle gängigen Plattformen verfügbar.

Nachteile:

- *GoVisual* ist zu sehr auf das Zeichnen von UML-Klassendiagrammen ausgelegt, welches sich nur schwer auf das Zeichnen von *Statecharts* übertragen lässt.
- Für den akademischen Einsatz ist keine kostenfreie Variante erhältlich.

CrocoCosmos

CrocoCosmos [42] ist ein Werkzeug zur dreidimensionalen Visualisierung von großen objektorientierten Softwaresystemen. Anhand der Darstellung soll das betrachtete System leichter verständlich werden, und es lassen sich bessere Qualitätseinschätzungen machen. Das Projekt verwendet für das Layout einen *Force-Directed*-Graph-Zeichen-Algorithmus.

Vorteile:

- Das Projekt nutzt ein *Force-Directed*-Verfahren zur Berechnung des Layouts.

Nachteile:

- *CrocoCosmos* ist Teil eines experimentellen Software-Projekts und kann somit nicht als einzelstehende Applikation genutzt werden.
- Das Themengebiet ist zu speziell und die zugrundeliegenden Ansätze lassen sich schlecht übernehmen.

ViSta

ViSta [57] steht für *Visualizing Statecharts*. Wie der Name bereits sagt werden mit diesem Werkzeug Layouts für *Statecharts* berechnet und dargestellt. Die Zustände unterteilen sich in einfache, AND- und OR-Zustände.[2] Das Layout der AND-Zustände hat Bezüge zu Grundrissplanungen, und für das Zeichnen wird ein abgewandelter Sugiyama-Algorithmus verwendet. Allerdings ist die Weiterentwicklung des Projekts nach Aussage der angegebenen Referenz im Jahr 2003 eingestellt worden.

Vorteile:

- Der strukturelle Aufbau ähnelt dem des Projekts *KIEL*.
- Hierarchieübergreifende Transitionen werden unterstützt.

[2] ähnlich wie im Projekt *KIEL* (siehe Abschnitt 2.2)

Nachteile:

- Das Projekt ist nicht mehr verfügbar.
- Die Darstellungen der *Statecharts* entsprechen nicht den Vorstellungen des Projekts *KIEL*. Zum Beispiel lassen sich alle Knoten ausschließlich als Rechtecke darstellen, deren Größe sich aus der Namenslänge ergibt.

Visone

Visone [51] beschäftigt sich mit der Analyse und Visualisierung von sozialen Netzwerken. Die Haupteinsatzgebiete des Projekts sind Lehre und Forschung. Das Projekt nutzt verschiedene Layout-Algorithmen, um die Zusammenhänge eines sozialen Netzwerks mit unterschiedlichen Betrachtungsschwerpunkten zu versehen. Die in sich abgeschlossene Applikation ist für alle gängigen Plattformen zur freien Verfügung bereitgestellt.

Vorteile:

- *Visone* implementiert verschiedene Layout-Algorithmen, unter anderem einen für die ebenenbasierte Darstellung.
- Das Projekt ist für alle gängigen Plattformen erhältlich.

Nachteile:

- Die Form und Größe eines Knoten ist abhängig von seiner sozialen Stellung innerhalb des Netzes und lässt sich nicht im Vorfeld festlegen.
- Das Projekt ist eine in sich abgeschlossene Applikation, die sich schlecht als *Framework* nutzen lässt.

Polyhemus und *Hermes*

Polyhemus [54] und *Hermes* [55] behandeln die Visualisierung von Computernetzwerken, wie dem Internet. Für das Layout der Graphen wird eine Mischung aus Gruppierungsalgorithmen, sternförmigem und orthogonalem Layout benutzt. Beide Applikationen sind in *Java* implementiert und in sich abgeschlossen.

Vorteile:

- Die *Java*-Implementierung gewährt einen plattformübergreifenden Zugriff.
- Beide Projekte stehen kostenfrei zur Verfügung.

Nachteile:

- Die zugrundegelegten Ansätze sind speziell auf das Themengebiet der Netzwerkanalyse und -visualisierung zugeschnitten.

- Die Projekte sind in sich abgeschlossen und lassen sich nur bedingt als *Framework* nutzen.

Abschließend ist zu sagen, dass viele der eben vorgestellten Werkzeuge sehr spezialisiert auf ihr jeweiliges Themengebiet zugeschnitten sind. Diese Werkzeuge bieten keine günstige Basis für weiterführende Arbeiten auf anderen Gebieten, wie zum Beispiel dem Layout von *Statecharts*. Im folgenden Abschnitt werden die Projekte, die als Grundlage für das *Statechart*-Layout innerhalb des Projekts *KIEL* eingesetzt werden können, noch einmal detaillierter miteinander verglichen.

3.2.3. Direkter Vergleich und Selektion

Jedes der in Abschnitt 3.2.2 beschriebenen Werkzeuge beschäftigt sich mit dem Zeichnen von Graphen. Allerdings stellt das Zeichnen von *Statecharts* zum Teil andere Anforderungen. So weisen die Zustände eines *Statecharts* einen semantischen Zusammenhang dadurch auf, dass die Zustände während der Simulation in einer bestimmten Reihenfolge durchlaufen werden. Ein vergleichbarer Zusammenhang wird bei den Knoten eines Graphen nicht wiedergespiegelt. Darüber hinaus ist die Leistungsfähigkeit eines Layout-Algorithmus in Hinblick auf die Verarbeitung einer Vielzahl von Knoten und Kanten als weniger relevant einzustufen, weil ein *Statechart* im Allgemeinen eine relativ geringe Anzahl an Zuständen und Transitionen aufweist. Dies gilt insbesondere dann, wenn jede Hierarchieebene eines *Statecharts* separat verarbeitet wird.

	AGD	*GraphViz*	*yFiles*
Algorithmenauswahl			
Layout planarer Graphen	++	-	-
orthogonales Layout	++	-	++
ebenenbasiertes Layout	++	+	+
kreisförmiges Layout	-	+	+
spiralförmiges Layout	-	+	-
Force-Directed-Layout	-	+	+
Eingabeformate	-	+	-
Ausgabeformate	-	++	+
Programmiersprache	*C++*	*C++*	*Java*
Zugriffsmöglichkeiten			
Bibliothek	+	+	+
ausführbares Programm	-	+	-
Lizenz	kostenfrei	kostenfrei	kommerziell
Aktualität	+	+	+
letztes Release	Dez. 2003	Dez.2004	Dez. 2004
Anwenderbasis	+	++	+

Tabelle 3.1.: Graph-Zeichen-Werkzeuge im direkten Vergleich

Viele der in Abschnitt 3.2.2 beschriebenen Werkzeuge sind bereits auf ein bestimmtes Themengebiet zugeschnitten und in diesen Bereich eingebettet. Deshalb wurden bei der Auswahl eines *Frameworks* vor allem die Projekte berücksichtigt, die eine sehr allgemeine Herangehensweise an das Graph-Zeichnen aufweisen. Aus den genannten Gründen sind nur drei Werkzeuge für die weitere Verwendung im Projekt *KIEL* geeignet. Diese werden in Tabelle 3.1 anhand der wichtigsten Entscheidungskriterien direkt miteinander verglichen.

Die Tabelle zeigt, dass alle Werkzeuge sehr gute Voraussetzungen für den weiteren Einsatz im Projekt *KIEL* haben. Als erstes wurde der Kostenfaktor berücksichtigt, in dem *AGD* und *GraphViz* einen klaren Vorteil haben, und *yFiles* somit nicht weiter berücksichtigt wurde. Beim Vergleich zwischen *AGD* und *GraphViz* hat *AGD* wegen seiner großen Algorithmenauswahl das eindeutig größere Potential. Allerdings ist der Einstieg in *GraphViz* wegen seiner guten Dokumentation und der leichten Erlernbarkeit des *Dot*-Formats (siehe hierzu Abschnitt 3.3.2) sehr viel leichter. Aus diesem Grund wurde *GraphViz* für die Einbettung in das Projekt *KIEL* ausgewählt. Trotzdem sollte die Anbindung des Projekts *AGD* nicht völlig verworfen werden, da sie ein sehr großes Potential bietet.

Im folgenden Abschnitt wird das ausgewählte Projekt *GraphViz* im Detail beschrieben. Hierbei wird zunächst der konzeptionelle Aufbau beleuchtet und anschließend die unterschiedlichen Zugriffsmöglichkeiten vorgestellt.

3.3. *GraphViz* im Detail

Das Projekt *GraphViz* umfasst eine Vielzahl von Programmen zur Betrachtung und Darstellung von beliebigen Graphen. Im Rahmen dieser Arbeit sind insbesondere die folgenden Programme zur Berechnung von graphischen Layouts von Interesse:

Dot erzeugt ein hierarchisches Layout im *Sugiyama-Stil* von gerichteten Graphen. Dabei wird versucht, alle Kanten in derselben Richtung (von oben nach unten oder von links nach rechts) anzuordnen. Gleichzeitig werden Überschneidungen vermieden und die Kantenlänge kurz gehalten.

Neato verfolgt den *Spring-Embedder*-Ansatz, wofür die Arbeit von Kamada und Kawai [27] als Grundlage dient. In der Arbeit von Gansner et al. [19] ist das Verfahren weiter optimiert worden.

Circo erstellt ein kreisförmiges Layout nach den beiden Arbeiten von Six und Tollis [40, 41], bei dem die Knoten auf einem Kreis angeordent werden.

Twopi stützt sich auf die Arbeit von Wills [61]. Bei diesem Verfahren werden die Knoten sprialförmig platziert.

Das Projekt *GraphViz* basiert auf einer einheitlichen Datenstruktur, die den Kern bildet. Auf dieser Grundlage wurde eine Vielzahl von Layout-Routinen implemen-

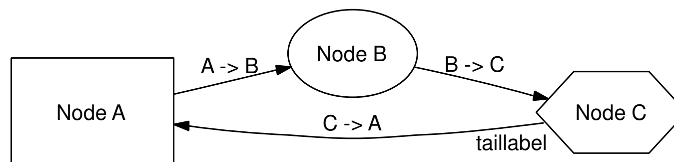


Abbildung 3.8.: Beispielgraph im *Dot*-Layout

tiert. Jedes der oben beschriebenen Programme setzt sich hierbei aus diesen Einzelteilen zusammen, so dass jedes ein eigenständiges Layout-Verfahren für statische Graphen bereitstellt.

Neben den verschiedenen Layout-Verfahren bietet *GraphViz* auch unterschiedliche Methoden des Zugriffs auf die einzelnen Verfahren an. Die Verwendung einer grafischen Benutzeroberfläche ist in Hinblick auf die Einbettung in das Projekt *KIEL* nicht geeignet. Im Folgenden wird zunächst der syntaktische Aufbau eines *GraphViz*-Graphen beschrieben und anschließend die Nutzung der Layout-Verfahren über den Aufruf des jeweiligen ausführbaren Programms bzw. die mitgelieferte *C++*-Bibliothek erläutert. Dafür dient das in Abbildung 3.8 dargestellte Fallbeispiel als gemeinsame Grundlage. Abschließend wird der *Dot*-Layout-Algorithmus vorgestellt.

3.3.1. Aufbau eines *GraphViz*-Graphen

Beim Aufbau eines Graphen unterscheidet *GraphViz* zwischen gerichteten und ungerichteten Graphen. Für diesen Graphen lassen sich beliebige Knoten und Kanten definieren. Hierbei ist darauf zu achten, dass jeder Knoten einen eindeutigen Namen erhält. Die Kanten werden anhand ihres Quell- und Zielknotens definiert.

Über die Angabe von Attributen kann man die Darstellung des Graphen näher spezifizieren. Ein Attribut besteht immer aus einem Namen und einem Wert. So kann der Benutzer zwischen verschiedenen Formen (`shape`) eines Knoten wählen oder auch eigene Formen einbetten. Ebenso lassen sich auf diese Weise Breite (`width`) und Höhe (`height`) eines Zustands den eigenen Vorstellungen anpassen. Für die Kanten sind drei Arten von Beschriftungen vorgesehen: je eine am Anfang (`headlabel`) und am Ende (`taillabel`), sowie eine in der Mitte (`label`) der Kante. Die Ausrichtung (vertikal, horizontal) des Graphen lässt sich über das Attribut `rankdir` bestimmen.

Sowohl für die Knoten als auch für die Kanten lassen sich globale Standardwerte definieren, die dann für alle Knoten bzw. Kanten, sofern nicht lokal anders definiert, gelten. Diese Einstellungen eignen sich besonders gut für das Setzen der zu verwendenden Schriftart (`fontname`) und Schriftgröße (`fontsize`).

Im folgenden Abschnitt wird gezeigt, wie sich ein solcher Graph über die unterschiedlichen Zugriffsmöglichkeiten erstellen und verarbeiten lassen kann.

3.3.2. Die möglichen Zugriffsmethoden

In diesem Abschnitt werden die Zugriffsmethoden auf das Projekt *GraphViz* anhand des in Abbildung 3.8 dargestellten Fallbeispiels erläutert.

Zugriff über die Kommandozeile

Alle Layout-Verfahren des Projekts *GraphViz* sind als separates Programm über die Kommandozeile aufrufbar. An den Umgang mit den Programmen *Dot* [17] und *Neato* [33] wird der Benutzer in den entsprechenden Papieren gut herangeführt. Im Allgemeinen ist die Handhabung der Programme einheitlich. Die Eingabedaten werden entweder von einer übergebenen Datei oder vom Eingabestrom gelesen. Die Ausgabe erfolgt in eine spezifizierte Datei oder in den Ausgabestrom. Darüber hinaus kann der Benutzer zwischen verschiedenen Ausgabeformaten wählen. Darunter sind viele grafische Ausgaben wie PostScript, `JPEG`, `PNG` oder `SVG`, aber auch einige textbasierte Ausgaben. Unter diesen ist insbesondere das *Attributed-Dot*-Format zu erwähnen, welches das Projekt *KIEL* für die weitere Verarbeitung der Daten verwendet. Im späteren Verlauf dieses Abschnitts wird genauer auf dieses Format eingegangen.

```
1  digraph dotsample {
2    graph[rankdir="LR"];
3
4    A [label="Node A", width="1.50", height="1.00",
5        shape="box"];
6    B [label="Node B", width="1.20", height="0.80",
7        shape="ellipse"];
8    C [label="Node C", width="1.40", height="0.75",
9        shape="hexagon"];
10
11   A -> B [label="A -> B"];
12   B -> C [label="B -> C"];
13   C -> A [label="C -> A", taillabel="taillabel",
14            labelangle="25", labeldistance="2.5"];
15 }
```

Abbildung 3.9.: Beispielgraph im *Dot*-Format

Als Eingabe werden von allen Programmen Daten im *Dot*-Format akzeptiert. Abbildung 3.9 zeigt das Fallbeispiel von Abbildung 3.8 im *Dot*-Format. Das Schlüsselwort `digraph` (Zeile 1) signalisiert, dass ein gerichteter Graph erstellt werden soll. `A`, `B` und `C` (Zeile 4-9) sind die Knoten, deren Attribute in den eckigen Klammern durch Kommata getrennt angegeben werden. Mit `A -> B`, `B -> C` und `C -> A` (Zeile 11-14) werden die Kanten definiert. Alle Kanten erhalten ihre Bezeichnung als `label`. Die Kante von `C` nach `A` (Zeile 13-14) definiert darüber hinaus noch ein `taillabel`. Die schließende geschweifte Klammer beendet die Definition des Graphen (Zeile 15).

```
digraph dotsample {
        graph [rankdir="LR"];
        node [label="\N"];
        graph [bb="0,0,450,110"];
        A [label="Node A", width="1.50", height="1.00",
           shape="box", pos="54,45"];
        B [label="Node B", width="1.20", height="0.80",
           shape="ellipse", pos="229,81"];
        C [label="Node C", width="1.40", height="0.75",
           shape="hexagon", pos="399,45"];
        A -> B [label="A -> B",
                pos="e,188,73 108,56 131,60 156,66 178,71",
                lp="147,75"];
        B -> C [label="B -> C",
                pos="e,357,54 270,72 293,68 322,62 347,56",
                lp="310,75"];
        C -> A [label="C -> A", taillabel=taillabel,
                labelangle="25", labeldistance="2.5",
                pos="e,108,36 355,37 331,33 300,29 272,27
                       234,24 225,24 186,27 164,29 140,31
                       118,34",
                lp="229,35", tail_lp="334,23"];
}
```

Abbildung 3.10.: Beispielgraph nach dem *Dot*-Layout im *Attributed-Dot*-Format

Das oben angesprochene Ausgabeformat *Attributed-Dot*-Format erweitert das *Dot*-Format, das einen Graphen rein syntaktisch beschreibt, um Informationen für die grafische Repräsentation. So werden den Knoten und Kanten Positionsattribute (`pos`) zugewiesen, und die Größe des Graphen (`bb`) festgelegt. Abbildung 3.10 zeigt das Fallbeispiel nach dem Layout durch das Programm *Dot*.

Im folgenden Abschnitt wird beschrieben, auf welche Weise sich das eben beschriebene Vorgehen auch über den direkten Zugriff auf die *GraphViz*-Bibliothek umsetzen lässt.

Zugriff über die *C++*-Bibliotheksschnittstelle

Neben der Ausführung eines der Layout-Programme ermöglicht das Projekt *GraphViz* auch den direkten Zugriff über die mitgelieferte *C++*-Bibliothek. Diese Vorgehensweise ist in der Arbeit von Gansner [18] sehr gut beschrieben worden. Für den tieferen Einstieg in die zugrunde liegende Datenstruktur (`Agraph`) ist das Papier von North [32] zu empfehlen.

Zu Beginn muss die Bibliothek mit `aginit` initialisiert werden. Um ein Graphobjekt zu erstellen, kann dieses aus einer Datei mittels `agread` gelesen oder durch den Aufruf von `agopen` direkt erstellt werden. Insbesondere die letztgenannte Variante wird vom Projekt *KIEL* verwendet.

Die Funktionen `agnode` und `agedge` fügen dem Graphen neue Knoten bzw. Kanten

```
33      a = agnode(graph, "A");
34      agset(a, "label", "Node A");
35      agset(a, "width", "1.50");
36      agset(a, "height", "1.00");
37      agset(a, "shape", "box");
```

Abbildung 3.11.: Erstellen eines Knotens mit der *C++*-Bibliothek

```
54      ca = agedge(graph, c, a);
55      agset(ca, "label", "C -> A");
56      agset(ca, "taillabel", "taillabel");
57      agset(ca, "labelangle", "25");
58      agset(ca, "labeldistance", "2.5");
```

Abbildung 3.12.: Erstellen einer Kante mit der *C++*-Bibliothek

hinzu. Beim Setzen der Attribute ist zunächst sicherzustellen, dass das entsprechende Attribut bereits definiert ist. Dafür prüft man, ob das gewünschte Attribut von `agfindattr` gefunden wird. Ist dies nicht der Fall, dann muss vor dem Setzen mit `agset` der Standardwert mit `agraphattr`, `agnodeattr` oder `agedgeattr` festgelegt werden.

Nachdem der Graph erstellt und alle Attribute gesetzt wurden, kann man eine der Layout-Routinen mittels `dot_layout`, `neato_layout`, `circo_layout` oder `twopi_layout` aufrufen. Anschließend lassen sich die erstellten Layout-Informationen als Attribute, äquivalent zum *Attributed-Dot*-Format, mittels `attach_attr` den Komponenten des Graphen hinzufügen. Diese lassen sich dann vergleichsweise unkompliziert mit dem Aufruf von `agget` abfragen. Abschließend sollte nach jedem Layout die entsprechende Reinigungsroutine (`[dot|neato|circo|twopi]_cleanup`) aufgerufen werden.

Abbildung 3.11 zeigt am Beispiel von `Node A`, wie ein Knoten erstellt wird und ihm anschließend verschiedene Attribute zugewiesen werden. In Abbildung 3.12 wird die Erzeugung der Kante von `Node C` nach `Node A` exemplarisch dargestellt. Im Anhang B ist das vollständige Programm in der Gesamtheit aufgelistet, das den Graphen des Fallbeispiels von Abbildung 3.8 erstellt, ein *Dot*-Layout berechnet und anschließend die Größe des Graphen ausgibt.

Nachdem die Zugriffsmöglichkeiten auf das Projekt *GraphViz* beschrieben wurden, befasst sich der nächste Abschnitt mit dem Layout-Algorithmus der *Dot*-Programms. Dieser Abschnitt erklärt sehr detailliert die Vorgehensweise des Verfahrens und erhöht somit das Verständnis für den in Abschnitt 4.3.2 entwickelten `GraphvizLayouter`.

3.3.3. Der *Dot*-Layout-Algorithmus

Der folgende Abschnitt befasst sich mit der Arbeitsweise des *Dot*-Layouts, wie sie in der Arbeit von Gansner et al. [20] ausführlich beschrieben ist. Diese ist für weiterführende Arbeiten zu empfehlen, da an dieser Stelle nur ein zusammenfassender Überblick gegeben werden soll. Der hier vorgestellte Algorithmus gliedert sich in vier Phasen, die im Anschluss näher beschrieben werden.

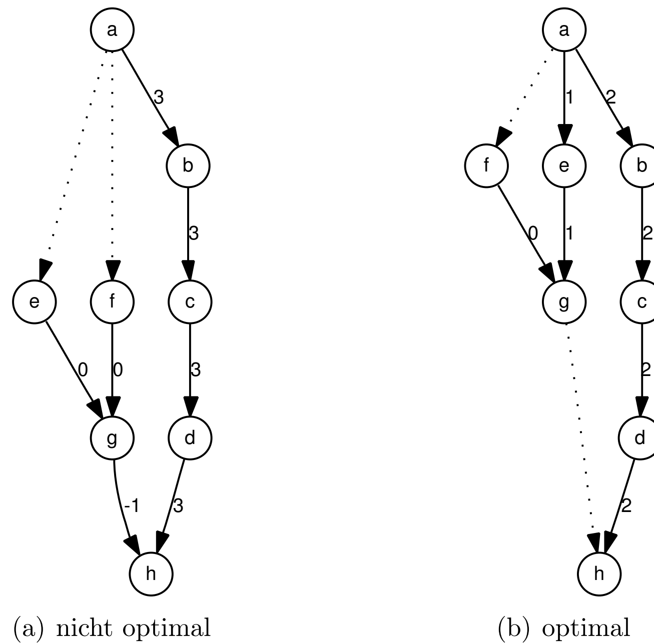
Phase 1: Einteilen der Knoten in Ränge

In der ersten Phase werden zunächst mögliche Zyklen innerhalb des Graphen aufgelöst. Dazu wird die Richtung einiger Kanten temporär umgekehrt, wie es in in der Arbeit von Rowe et al. [39] beschrieben ist. Das Finden der geeigneten Kanten ist nach Garey und Johnson [21] und Eades et al. [11] NP-vollständig, so dass hierfür eine Heuristik benutzt wird. Diese findet Kanten, die an vielen Zyklen beteiligt sind, und kehrt deren Richtung um (siehe Abbildung 3.5). Verschiedene Experimente haben ergeben, dass das Umdrehen der „besten" Kante nicht zwangsläufig ein „besseres" Bild erzeugen muss. Gerade bei Graphen, die eine einheitliche Flussrichtung aufweisen, funktioniert die Heuristik zufriedenstellend. Ein Graph hat dann eine einheitliche Flussrichtung, wenn die meisten Kanten des Graphen dieselbe Richtung haben (z. B. von oben nach unten). Insbesondere bei *Statecharts* ist dieses Merkmal zu finden, da hier der Ausführungspfad vom initialen zum finalen Zustand führt.

Das Zuordnen der Ränge geschieht mittels des Netzwerk-Simplex-Algorithmus. Dafür wird zunächst ein Spannbaum über den Graphen gelegt. Abbildung 3.13 zeigt zwei mögliche Spannbäume eines Graphen, wobei nur die durchgezogenen Linie Teil des Spannbaums sind. Anhand des Spannbaums kann dann jedem Knoten ein Rang zugewiesen werden. Im initialen Schritt wird ein beliebiger Knoten auf einem mittleren Rang platziert. Anschließend werden alle Nachbarknoten soweit entfernt platziert, wie es die Mindestlänge der verbindenden Kante vorgibt. Standardmäßig gilt: Mindestlänge=1. Dieser Ablauf wiederholt sich, bis alle Knoten einem Rang zugeordnet sind.

Abschließend wird der Baum dahingehend untersucht, ob alle Kanten „optimal" sind. Optimal ist eine Kante dann, wenn ihr kein negativer *Schnittwert* zugewiesen wird. Zur Berechnung des *Schnittwertes* einer Kante wird der Baum an dieser Kante durchschnitten, so dass der Baum in zwei Teile zerfällt. Der Teil, der den Quellknoten enthält, wird Quellkomponente genannt. Der andere heißt Senkenkomponente. Wenn mehrere Kanten des gesamten Graphen von der Senkenkomponente zur Quellkomponente gehen, dann ist die durchtrennte Kante negativ zu bewerten. Eine Kante, die nicht „optimal" ist, wird durch eine Kante ersetzt, die von der Quellkomponente ausgeht. Der so entstandene Spannbaum wird erneut betrachtet. Wenn alle Kanten des Spannbaums „optimal" sind, terminiert der Algorithmus und die erste Phase des Layout-Verfahrens ist abgeschlossen.

Abbildung 3.13 zeigt zwei mögliche Spannbäume eines Graphen. Die gepunkteten Kanten sind Graphkanten, aber keine Baumkanten. Der Schnittwert der Baumkan-



(a) nicht optimal

(b) optimal

Abbildung 3.13.: Finden des optimalen Spannbaums (Quelle: [20])

ten ist jeweils an der Kante vermerkt. Im linken Baum ist der Kante (g → h) ein Wert von −1 zugeordnet. Dieser ergibt sich wie folgt: Nach dem Durchschneiden der Kante (g → h) ergeben sich die Quellkomponente, bestehend aus den Knoten e, f und g, und die Senkenkomponente, bestehend aus den restlichen Knoten. Die Quellkomponente hat eine verbindende Kante zur Senkenkomponente ((g → h)). In der anderen Richtung gibt es zwei Verbindungen ((a → e), (a → f)). Im rechten Spannbaum sind alle Kanten „optimal".

Phase 2: Ordnen der Knoten innerhalb der Ränge

In der zweiten Phase werden die Knoten innerhalb eines Ranges sortiert. Zunächst werden alle Kanten, die über mehrere Ränge hinwegreichen, in einzelne Segmente unterteilt. Dafür werden temporär virtuelle Knoten auf jedem Zwischenrang hinzugefügt. Eigenkanten werden für diesen Schritt ignoriert und Mehrfachkanten vereinigt. Somit enthält der betrachtete Graph nur noch Kanten zwischen benachbarten Knoten.

Eine gute Anordnung enthält wenige Kantenkreuzungen. Da das Finden der optimalen Lösung, wie bereits oben erwähnt, NP-vollständig ist, wird eine Heuristik verwendet, die auf der Arbeit von Warfield [59] basiert. Im ersten Schritt wird eine initiale Anordnung berechnet. Anschließend werden die Ränge in mehreren Iterationsschritten vorwärts und rückwärts bearbeitet. Dabei erhält jeder Knoten eine Gewichtung, die von den relativen Positionen der Verbindungsknoten aus dem vorangegangenen Rang abhängt. Als Verbindungsknoten werden die Knoten bezeichnet, zu denen eine verbindende Kante existiert. Abschließend werden die Knoten anhand

dieser Gewichtung neu angeordnet.

Für die Gewichtung wird eine weitere Heuristik benutzt. Diese erweitert die Median Methode von Eades und Wormald [12] um zwei Feinheiten:

1. Wahl des Median
2. Vermeidung von Kantenkreuzungen

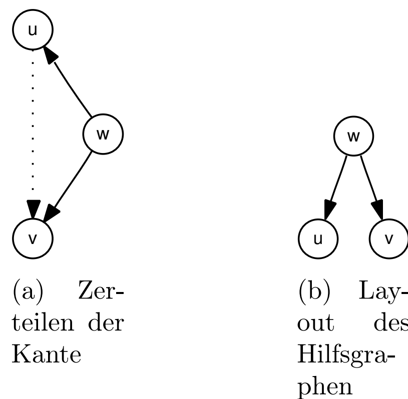
Beide Verfahren berechnen zunächst den Median anhand der Positionen der Verbindungsknoten. Das heißt, dass bei einer ungeraden Anzahl an Verbindungsknoten der Median der mittleren Position entspricht und bei gerader Anzahl den mittleren beiden Positionen. Eine der Neuerungen sieht eine verbesserte Wahl des Medians vor, wenn zwei Mediane ermittelt wurden. In diesem Fall wird der Median verwendet, dessen Wert dichter am Interpolationswert aller Positionen liegt. Die ursprüngliche Methode legt sich auf den linken oder rechten Median fest. Die zweite Neuerung sieht bei der Neuanordnung der Knoten eine Reduktion der offensichtlichen Kantenkreuzungen vor. Dafür wird überprüft, ob ein Vertauschen zweier benachbarter Knoten die Anzahl der Kreuzungen minimiert.

Ein Ästhetikkriterium des *Dot*-Layouts besagt, dass möglichst alle Kanten in derselben Richtung verlaufen sollen. Dies gilt insbesondere für Kanten, die innerhalb eines Rangs verlaufen. Solche Kanten werden als flach bezeichnet. Knoten, die durch flache Kanten verbunden sind, bilden wegen des einheitlichen Verlaufs der Kanten eine partielle Ordnung. Diese Ordnung wird durch die oben beschriebene Neuanordnung der Knoten nicht verletzt, so dass beim Neuordnen immer die gesamte Gruppe verschoben wird.

Es hat sich herausgestellt, dass die zweite Phase bessere Ergebnisse liefert, wenn sie zweimal ausgeführt wird. Im ersten Schritt werden die Ränge der initialen Ordnung zuerst vorwärts durchlaufen, im zweiten zuerst rückwärts. Abschließend wird das Ergebnis übernommen, das die geforderten Ästhetikkriterien besser erfüllt (z. B. weniger Kantenkreuzungen).

Phase 3: Vergabe konkreter Knotenkoordinaten

Die genauen Koordinaten der Knoten werden in der dritten Phase des Layout-Algorithmus festgelegt. Dabei werden die beiden Koordinaten getrennt voneinander berechnet. Zur Berechnung der Y-Koordinate werden die Ränge so platziert, dass der definierbare Rangabstand zwischen den jeweils größten Knoten eingehalten wird. Anschließend erhalten alle Knoten eines Rangs die gleiche Y-Koordinate. Die Berechnung der X-Koordinate erweist sich als schwieriger. Hierfür wird ein Hilfsgraph erstellt, für den der ursprüngliche Graph erweitert wird. Für jede Kante von u nach v wird dem Hilfsgraphen ein virtueller Knoten w hinzugefügt, und die Kante (u $\rightarrow$ v) wird durch die Kanten (w $\rightarrow$ u) und (w $\rightarrow$ v) ersetzt. Das Erstellen des Hilfsgraphen ist im linken Teil von Abbildung 3.14 dargestellt. Die Originalkante (u $\rightarrow$ v) ist gepunktet markiert. Das Resultat des anschließenden Layouts ist im rechten Teil der Abbildung zu sehen.



(a) Zerteilen der Kante

(b) Layout des Hilfsgraphen

Abbildung 3.14.: Erstellung und Layout eines Hilfsgraphen

Für den oben beschriebenen Hilfsgraphen wird erneut der Netzwerk-Simplex-Algorithmus aus der ersten Phase benutzt. Die X-Koordinate ergibt sich, ähnlich der Berechnung der Y-Koordinate, aus dem zugewiesenen Rang innerhalb des Hilfsgraphen.

Phase 4: Zeichnen der Kanten als *Splines*

In der abschließenden vierten Phase werden die Kanten gezeichnet. Für jede Kante werden zunächst Boxen, die parallel zu den Rängen verlaufen, berechnet, in denen die Kante dann als *Spline* verlaufen soll. Anschließend wird der *Spline* gezeichnet. Da die Kanten nacheinander platziert werden, erscheint es lohnenswert, die Kanten im Vorfeld ihrer Länge nach zu sortieren, da kurze Kanten weniger Platz beanspruchen als lange. Dennoch beeinflusst dieser Schritt die Qualität des Layouts nur in geringem Maße. Generell werden drei Arten von Kanten unterschieden:

- rangübergreifende Kanten, deren Knoten auf unterschiedlichen Rängen liegen,
- flache Kanten, deren Knoten auf demselben Rang liegen, und
- Eigenkanten, deren Startknoten auch deren Zielknoten sind.

Die Boxen der rangüberbreifenden Kanten Bei den rangübergreifenden Kanten gibt es an Anfang und Ende meist mehrere kleine Boxen und nur wenige große Boxen zwischen den Rängen. Abbildung 3.15 zeigt ein typisches Beispiel für das Anlegen der Boxen. Im dargestellten Fall soll der Kantenverlauf von Interdata nach Unix/TS 3.0 berechnet werden. Um die *Splines* „geschmeidig“ zu zeichnen, werden die einzelnen Boxen möglichst groß gewählt, so dass später beim Berechnen des *Spline*-Verlaufs keine unnötigen „Knicke“ entstehen. Bei der Auswahl der Boxen werden folgende ästhetische Kriterien berücksichtigt:

- Unvermeidbare Kreuzungen zwischen Kanten sollen einen „sauberen“ Schnitt erhalten, so dass sie sich nur einmal in einem gut erkennbaren Winkel schneiden.

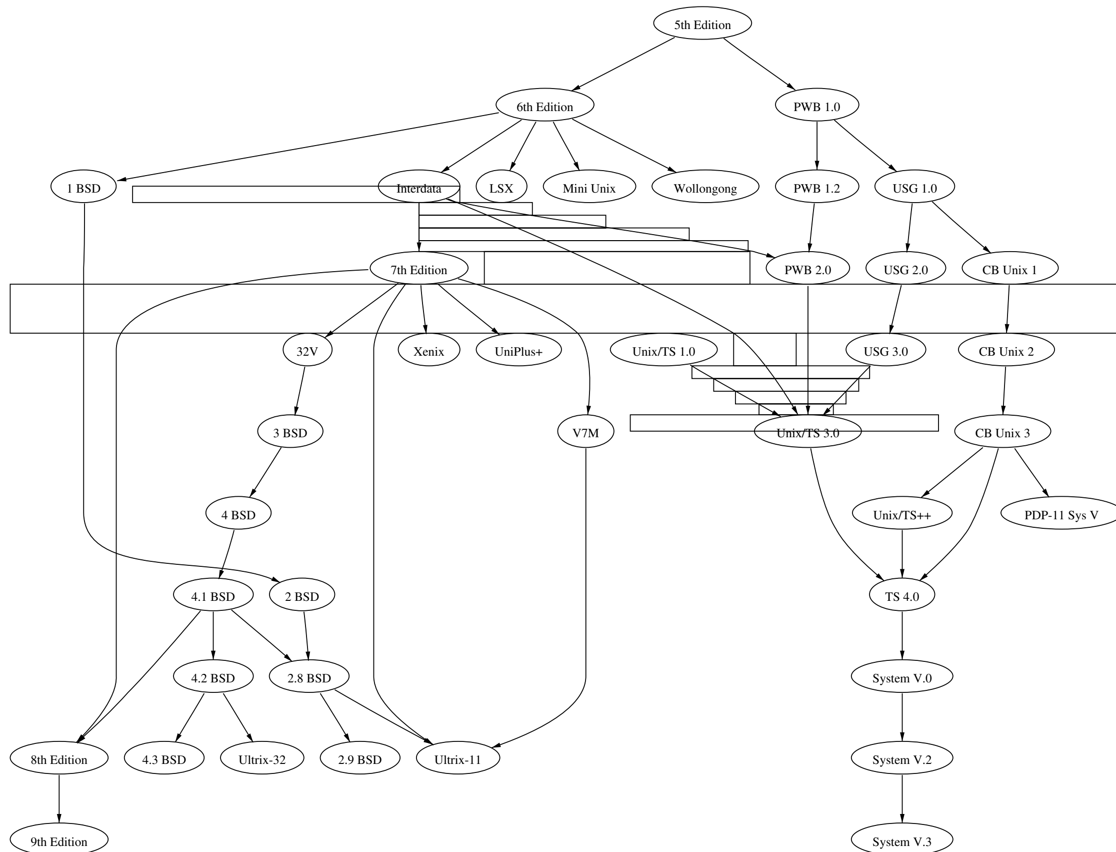


Abbildung 3.15.: Graph mit eingezeichneten Boxen (Quelle: [20])

- Lange, fast vertikale Abschnitte einer Kante werden in „echte“ vertikale Abschnitte überführt.
- Wenn ein *Spline* an einem Knoten endet, an dem bereits andere *Splines* enden, so werden dessen dichteste Nachbarn als Begrenzung für den eigenen Weg gewählt. Auf diese Weise werden unnötige Schnitte vermieden.

Wenn mehrere Kanten den selben Start- und Zielknoten haben, wird zunächst der Verlauf einer Kante wie gewohnt berechnet. Anschließend werden ihre Geschwister leicht verschoben platziert.

Die Boxen der flachen Kanten und Eigenkanten Auf ähnliche Weise wird auch der Verlauf von flachen Kanten berechnet. Allerdings gibt es hier einen Spezialfall. Wenn die Knoten einer Kante direkte Nachbarn sind, wird die Kante als Gerade gezeichnet. Bei Mehrfachkanten wird analog zu den rangübergreifenden Kanten verfahren. Eigenkanten werden als Schleifen an der Seite des jeweiligen Knoten platziert. Mehrfache Eigenkanten werden hierbei ineinander verschachtelt.

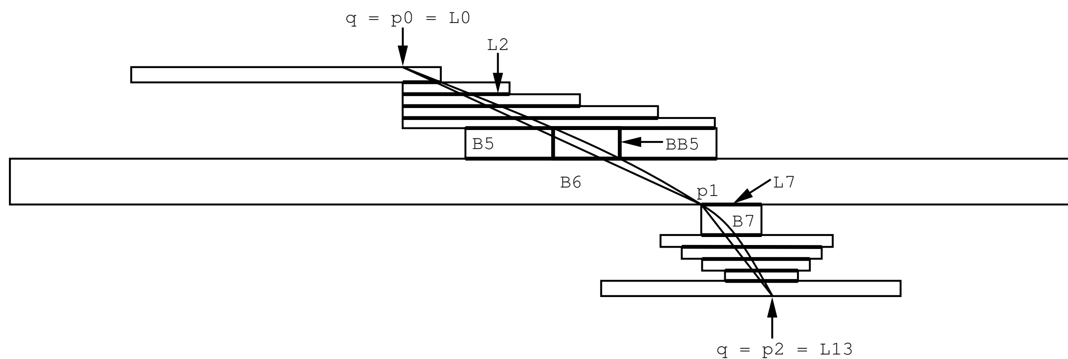


Abbildung 3.16.: Festlegen des *Spline*-Verlaufs (Quelle: [20])

Das Zeichnen des Kantenverlaufs Auf das Berechnen der Boxen für den Kantenverlauf folgt das eigentliche Zeichnen der Kante, deren Beschriftung (*Label*) bereits als virtueller Knoten in die Berechnungen eingeflossen ist. Generell wird versucht die Beschriftungen mittig zum Kantenverlauf zu platzieren. Das Zeichnen einer Kante erfolgt in drei Schritten, die in Abbildung 3.16 zusammengefasst sind:

1. Zunächst wird ein Polygonzug entlang der kürzesten Verbindung innerhalb der Boxen ermittelt. Dieser verläuft entlang der Punkte $p0, p1$ und $p2$.
2. Die Punkte dieses Polygonzugs dienen als Stützstellen des zu zeichnenden *Bezier-Splines*.
3. Abschließend wird der tatsächlich benötigte Platz berechnet. Die mit $B5$ markierte Box zeigt die ursprüngliche Größe, die mit $BB5$ markierte Box zeigt dagegen die tatsächlich benötigte Größe.

Die zugrundeliegende Beschreibung des Layout-Algorithmus stammt aus dem Jahr 1993, und das Projekt *GraphViz* ist während der Zeit ständig weiterentwickelt worden. Somit spiegelt das hier vorgestellte Verfahren nur einen generellen Ablauf des *Dot*-Layouts wieder. Feinheiten des Layouts haben sich bereits geändert und werden sich vermutlich auch in Zukunft noch weiter entwickeln.

3.4. Zusammenfassung der Analyse

In diesem Kapitel wurden verwandte Arbeiten aus dem Bereich des Graph-Zeichnens betrachtet. Generell lassen sich *Statecharts* als gerichtete Graphen betrachten, und somit sollte die Berechnung eines *Statechart*-Layouts analog zum Zeichnen eines Graphen verlaufen. Allerdings unterscheidet sich die Darstellung eines Graphen von der eines *Statecharts*. Diese Unterschiede müssen bei der Wahl eines Graph-Zeichen-Verfahrens berücksichtigt werden.

Die in Abschnitt 3.1 vorgestellten Graph-Layout-Verfahren lassen sich prinzipiell alle auf das Layout von *Statecharts* übertragen. Auch wenn die Voraussetzungen für

die Anwendung des Baum-Layouts (siehe Abschnitt 3.1.3) meistens nicht gegeben sind, da die Transitionen eines *Statecharts* nicht die Vater-Kind-Beziehung zweier Zustände wiederspiegeln. Für eine strukturierte und somit übersichtliche Darstellung eines *Statecharts* bieten sich ebenenbasierte und orthogonale Layouts an. Die einfache Variante des *Force-Directed*-Ansatzes eignet sich für das Layout von *Statecharts* nicht, weil hierbei eine Minimierung der Kantenkreuzungen nicht berücksichtigt wird. Jedoch würde die Art und Weise wie die Kanten verlaufen die Lesbarkeit solcher Layouts bereits erhöhen.

Bei der Wahl eines geeigneten *Frameworks* in Abschnitt 3.2 hat sich herausgestellt, dass die beschriebenen Layout-Verfahren auf den verschiedensten Gebieten erfolgreich zum Einsatz kommen. Je nach Einsatz werden an das Layout der Graphen ganz unterschiedliche Anforderungen gestellt. Unterscheidungsmerkmale sind neben der Form und Größe der Knoten auch der Verlauf der Kanten und die räumliche Ausdehnung des Graphen.

Für die Einbettung eines solchen Werkzeugs in das Projekt *KIEL* war dessen allgemeine Herangehensweise an das Erstellen eines Graph-Layouts eine entscheidende Voraussetzung. Neben der guten Verwendbarkeit des Projekts *GraphViz* war vor allem die leichte Handhabbung ein wichtiges Entscheidungskriterium. Darüber hinaus wird das Projekt bereits erfolgreich in unterschiedlichen Bereichen eingesetzt, was dafür spricht, dass es sich sehr gut an die jeweiligen Anforderungen anpassen lässt.

In Abschnitt 3.3 wurde das Projekt *GraphViz* eingehend beschrieben. Neben dem konzeptionellen Aufbau wurden auch die beiden Zugriffsmöglichkeiten vorgestellt, die sowohl eine direkte Benutzung der *C++*-Bibliothek als auch den Aufruf eines der ausführbaren Programme vorsehen. Für die Nutzung der Programmaufrufe wurde die *GraphViz*-eigene Beschreibungssprache im *Dot*-Format vorgestellt.

Die zusammenfassende Beschreibung des *Dot*-Layout-Algorithmus ist auf die wesentlichen Merkmale des Verfahrens beschränkt und verdeutlicht so die Vorgehensweise des *Dot*-Layouts. Dabei zeigt die beschriebene Vorgehensweise exemplarisch das generelle Vorgehen eines ebenenbasierten Layout-Verfahrens. Die Einbettung des Projekts *GraphViz* in das Projekt *KIEL* wird im folgenden Kapitel im Zusammenhang mit der Entwicklung des Moduls `Layouter` beschrieben.

4. Das Layout-Modul

In diesem Kapitel wird auf die Entwicklung des Moduls `Layouter` als Teil des Projekts *KIEL* eingegangen. Sowohl der Aufbau dieses Moduls als auch die Integration eines Layout-Verfahrens bilden den praktischen Teil dieser Arbeit. Der zugehörige Quell-*Code* ist im Anhang C hinterlegt.

In Abschnitt 4.1 werden zunächst die grundlegenden Anforderungen an das Modul spezifiziert. Das Konzept zur Umsetzung dieser Anforderungen wird im darauf folgenden Abschnitt 4.2 vorgestellt. Darüber hinaus wird die Schnittstelle zu den übrigen Modulen des Projekts *KIEL* definiert. Dieser Abschnitt ist für jeden Entwickler des Projekts *KIEL* zu empfehlen, der das Modul `Layouter` benutzen möchte.

Im Anschluss daran werden in Abschnitt 4.3 die einzelnen Layout-Klassen vorgestellt. Die instantiierten Objekte dieser Klassen werden im Folgenden als *Layouter* bezeichnet. Für einen dieser Layouter bildet das *Framework* aus Kapitel 3 die Grundlage. Die Beschreibung dieses *GraphViz*-Layouters ist in Abschnitt 4.3.2 zu finden. Abschließend wird die Entwicklung des Layout-Moduls in Abschnitt 4.4 zusammengefasst.

4.1. Spezifikation der Anforderungen

Bei der Entwicklung des Layout-Moduls mussten sowohl die allgemeinen Anforderungen an ein *KIEL*-Modul, wie zum Beispiel die Kapselung gegenüber anderen Modulen (siehe Abschnitt 2.2.1), als auch modulspezifische Anforderungen berücksichtigt werden. Diese Anforderungen sind im Folgenden aufgelistet.

Verwaltung der konfigurationsabhängigen Ansichten

Wie in Abschnitt 2.2.4 bereits beschrieben können für ein *KIEL-Statechart* verschiedene Ansichten hinterlegt werden. Neben der rein statischen Ansicht, in der alle Hierarchieebenen zu sehen sind, können abhängig vom zugrundeliegenden Layout-Verfahren diverse dynamische Ansichten generiert werden, die während eines Simulationsschritts relevante Informationen hervorheben. Da diese Generierung von jedem Layouter auf andere Weise vorgenommen werden kann, sollte die Verwaltung der Ansichten ebenfalls vom Layouter übernommen werden.

Topologisches Layout von *Statecharts*

Der zu entwickelnde Layouter soll ein topologisches Layout auf den Zuständen, Transitionen und *Labels* eines *Statecharts* durchführen können. Weitergehen-

de Betrachtungsfilter, die auf einer bestehenden Topologie arbeiten, wie zum Beispiel *Fish-Eye*, sollen dabei nicht berücksichtigt werden.

Erweiterbarkeit für zukünftige Layouter

Bei der Entwicklung des Layout-Moduls ist darauf zu achten, dass sich zukünftige Layouter, die andere Layout-Verfahren bereitstellen, leicht integrieren lassen. Dabei darf das Hinzufügen neuer Layouter die anderen Module des Projekts *KIEL* nicht beeinflussen.

Parametrierbarkeit

Die unterschiedlichen Einstellmöglichkeiten eines benutzten *Frameworks* sollen dem Benutzer auch nach der Einbettung in das Layout-Modul zur Verfügung stehen.

In dem nachfolgenden Abschnitt wird zunächst der konzeptionelle Aufbau des Layout-Moduls beschrieben.

4.2. Konzept und Schnittstellendefinition

Bei der Entwicklung des Layout-Moduls wurde insbesondere auf eine starke Kapselung geachtet. Aus diesem Grund ist die Schnittstelle zu den anderen Modulen des Projekts *KIEL* bewusst klein gehalten worden und umfasst deshalb nur drei Klassen:

- Der `Handler` instanziiert die einzelnen Layouter-Klassen und stellt diese Layouter den Modulen des Projekts *KIEL* zur Verfügung.
- Die Klasse `Layouter` stellt zum einen die Basisklasse für die einzelnen Layouter-Entwicklungen bereit, zum anderen definiert sie die Schnittstelle für den Zugriff der anderen Module. Darüber hinaus hält dieser Basis-Layouter die statische Ansicht auf ein *Statechart* vor.
- Der `PseudoLayouter` erweitert die Basisklasse `Layouter` um eine Methode, die den schreibenden Zugriff auf die statische Ansicht eines *Statecharts* bereitstellt.

Im Folgenden wird näher auf die einzelnen Klassen eingegangen.

Handler

Als zentrales Verbindungsglied stellt die Klasse `Handler` verschiedene Methoden für den externen und internen Zugriff bereit. Die Klasse selbst wurde nach dem Entwurfsmuster `Singleton` entwickelt, um sicherzustellen, dass nur eine Instanz der Klasse zur Zeit geöffnet ist. Diese Vorgehensweise ist besonders bei Verwaltungsklassen zu empfehlen.

Für externe Module stellt der `Handler` eine Liste bereit und erteilt Auskunft darüber, welche Layout-Verfahren (Layouter) zur Verfügung stehen. Diese können anschließend den gewünschten Layouter aus dieser Liste auswählen. Den

internen Klassen stehen verschiedene Methoden für den Zugriff auf die gemeinsame Protokolldatei zur Verfügung, die vom `Handler` gepflegt wird.

Bei der Instanziierung der Klasse werden die einzelnen Layouter angelegt und in einer Tabelle gespeichert. Der Zugriff auf diese Tabelle erfolgt über den eindeutigen Namen des jeweiligen Layouters. Die Namen der Layouter werden vom `Handler` vergeben, so dass die Eindeutigkeit gewährleistet ist. Die Verknüpfung zwischen einem Namen und dem Layouter hat zusätzlich den Vorteil, dass sich auf diese Weise leicht Auswahlmöglichkeiten für eine grafische Benutzeroberfläche erstellen lassen (beispielsweise als Eintrag in einer Menüleiste).

Layouter

Abbildung 4.1 gibt eine Übersicht über die zur Zeit implementierten Layouter und ihre vererbungsbedingten Zusammenhänge. An der Wurzel des Baums steht die Basisklasse `Layouter`, die die Schnittstelle zu den externen Modulen bereitstellt. Der Zugriff auf die einzelnen Layouter ist auf die Methoden der `Layouter`-Klasse beschränkt, weil der `Handler` nur Layouter dieses konkreten Typs zurückgibt.

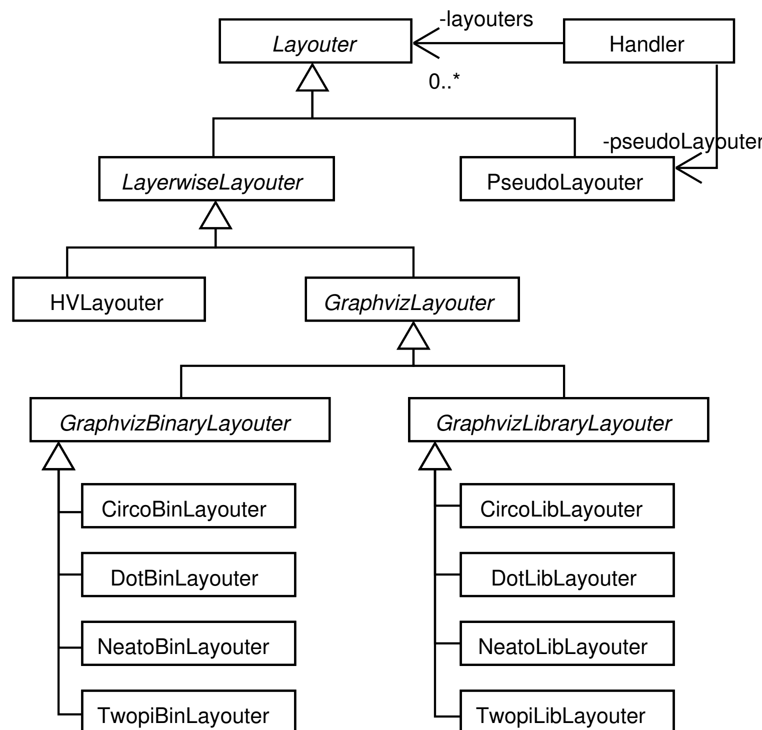


Abbildung 4.1.: Das Klassendiagramm des Layout-Moduls

Neben der Schnittstellendefinition verwaltet die `Layouter`-Klasse auch die statische Ansicht auf das zugrundeliegende *Statechart*. Jeder Layouter kann dar-

über hinaus dynamische Ansichten generieren, die er eigenständig verwalten muss. Diese Ansichten werden abhängig von der Konfiguration (`Configuration`), in der sich das *Statechart* während der Simulation befindet, angefordert. Der in Abschnitt 4.3.1 vorgestellte `LayerwiseLayouter` nutzt diese Möglichkeit.

Dabei gewährt der Aufbau der `Layouter`-Klasse sowohl die Berechnung aller Ansichten zu Beginn einer Sitzung, als auch das Erstellen einzelner Layouts während der laufenden Simulation.

PseudoLayouter

Der `PseudoLayouter` bildet eine Ausnahme unter den Layoutern, da er eine Methode zum Setzen der statischen Ansicht bereitstellt. Diese Ansicht kann entweder vom `Editor`-Modul erstellt oder vom `FileInterface`-Modul aus einer Datei gelesen worden sein. Beide Module sind im Grundlagenkapitel in Abschnitt 2.2.3 beschrieben worden. Die Methode zum Setzen der statischen Ansicht ist allerdings nicht Teil der `Layouter`-Methodenspezifikation. Deshalb erhalten Module, die diese Methode nutzen wollen, den `PseudoLayouter` über eine gesonderte Methode der `Handler`-Klasse.

Wie der Klassenname bereits andeutet, ist der `PseudoLayouter` kein Layouter im eigentlichen Sinne, da er kein Layout-Verfahren implementiert. Trotzdem kann dieser Layouter die konfigurationsabhängigen Ansichten während einer Simulation bereitstellen, indem er immer die statische Ansicht liefert. Auf diese Weise lassen sich Simulationen auch auf dem originalen Layout durchführen.

LayouterProperties

Für die Anforderung der Parametrierbarkeit der einzelnen Layouter existiert die Klasse `LayouterProperties`, die alle Einstellungen der Layouter zentral verwaltet. Zu Beginn einer Sitzung werden die benutzereigenen Einstellungen aus einer Datei gelesen. Sollte diese Datei nicht existieren, wird eine neue Datei mit den Standardeinstellungen erstellt. Im laufenden Betrieb stellt die `Layouter`-Klasse sicher, dass vor jedem Layout die Einstellungen aktualisiert werden.

Der Benutzer kann die Auswirkungen der veränderten Einstellungen somit auf intuitive Weise nachvollziehen, da kein erneutes Starten der Anwendung nötig ist. Darüber hinaus sind die einzelnen Parameter sowohl in Anhang A ausführlich beschrieben, als auch innerhalb der Datei kommentiert, um die Benutzung der Einstellmöglichkeiten zu erleichtern.

Die soeben vorgestellten Klassen bilden das Grundgerüst des Layout-Moduls. Hierauf aufbauend sind alle weiteren Layouter entwickelt worden. Um einen neuen Layouter in das Projekt *KIEL* zu integrieren, muss dieser von der Basisklasse `Layouter` oder von einer vererbten Variante abgeleitet werden. Anschließend muss der neuentwickelte Layouter in die Tabelle des `Handlers` eingetragen werden, damit

die anderen Module auf ihn zugreifen können. Die Anforderung der leichten Erweiterbarkeit wurde somit erfüllt. Im folgenden Abschnitt werden die entwickelten Layouter im Einzelnen detailliert vorgestellt.

4.3. Die Layouter-Klassen im Detail

Das im vorherigen Abschnitt vorgestellte Konzept stellt lediglich die Schnittstelle für den Zugriff der anderen Module bereit, ein Layout-Verfahren wurde bisher noch nicht implementiert. Im Folgenden werden diverse Layouter-Klassen beschrieben, die sukzessive aufeinander aufbauen. Dabei stellt der `LayerwiseLayouter` aus Abschnitt 4.3.1 eine breite Basis für viele Layouter bereit, indem er ein *Statechart* in dessen Hierarchieebenen aufteilt und das Layout dieser Ebenen jeweils separat von seinen Nachfolgern berechnen lässt. Auf dieser Basis baut der `GraphvizLayouter` aus Abschnitt 4.3.2 auf, der die Grundlagen zur Einbettung des Projekts *GraphViz* bereitstellt. Sowohl der `GraphvizBinarayLayouter` (Abschnitt 4.3.3) als auch der `GraphvizLibraryLayouter` (Abschnitt 4.3.4) verfeinern die Art der Anbindung. Der in Abschnitt 4.3.5 beschriebene `HVLayouter` erweitert den `LayerwiseLayouter` mit einem eigenen Layout-Verfahren.

4.3.1. Der LayerwiseLayouter

Der `LayerwiseLayouter` ist kein eigenständiger Layouter, da er kein vollständiges Layout-Verfahren bereitstellt. Dennoch übernimmt er einen Großteil des Layouts, das von seinen Nachfahren nicht mehr durchgeführt werden muss. Die folgenden Punkte sind für den `LayerwiseLayouter` charakteristisch:

- Betrachtung des *Statecharts* in einzelnen Ebenen,
- Layout der Zustände (insbesondere der `CompositeStates`) und
- Umsetzung von *Semantischem Zoom*.

Wie diese Charakteristiken im Einzelnen umgesetzt wurden, wird im späteren Verlauf dieses Abschnitts detailliert erläutert.

Das generelle Vorgehen des Layouters ist so zu beschreiben, dass bei der Verarbeitung mit dem Layout des Wurzelzustands (`rootNode`) eines *Statecharts* begonnen wird. Da der Wurzelzustand immer ein `CompositeState` ist, muss zuerst dessen „Inhalt“ betrachtet werden. Dabei werden Breite und Höhe jedes Zustands festgelegt, die sich aus Namenslänge und möglichem „Inhalt“ ergeben. Anschließend ermittelt das eigentliche Layout-Verfahren die genauen Koordinaten der Zustände und Transitionen.

Somit ergibt sich ein Layout-Verfahren, das mittels Tiefensuche für das gesamte *Statechart* ein Layout anfertigt. Der Benutzer kann dabei über Einstellungen in den `LayouterProperties` zwischen horizontaler, vertikaler und automatischer Layout-Richtung wählen. Bei der automatischen Einstellung werden die Ausmaße

der jeweiligen Richtungen miteinander verglichen und die platzsparendere Variante ausgewählt. Darüber hinaus ist das Alternieren der Layout-Richtung von Hierarchieebene zu Hierarchieebene einstellbar.

Weil der `LayerwiseLayouter` bereits wesentliche Teile des Layout-Prozesses übernimmt, genügt es, dass seine Nachfahren ausschließlich das Layout für eine einzelne Ebene berechnen.

Allerdings ist anzumerken, dass sich der `LayerwiseLayouter` auf *Statecharts* beschränkt, die keine hierarchieübergreifenden Transitionen beinhalten. Als hierarchieübergreifend gilt eine Transition dann, wenn der Quellknoten und der Zielknoten nicht den gleichen Vaterzustand haben. Abbildung 4.2 zeigt ein *Statechart* mit einer hierarchieübergreifenden Transition.

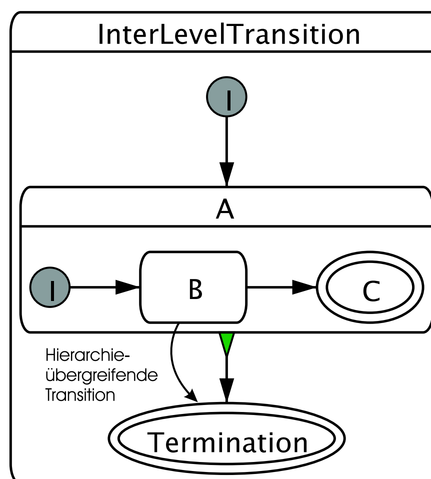


Abbildung 4.2.: *Statechart* mit einer hierarchieübergreifenden Transition[1]

Diese Einschränkung relativiert sich allerdings dadurch, dass viele Modellierungswerkzeuge, wie zum Beispiel *Esterel Studio*, die Verwendung von hierarchieübergreifenden Transitionen ebenfalls nicht unterstützen.

Im Folgenden werden die zu Beginn des Abschnitts vorgestellten Merkmale des `LayerwiseLayouters` detailliert vorgestellt.

Ebenenweise Betrachtung

Das markanteste Merkmal des `LayerwiseLayouters` ist die Betrachtung eines gegebenen *Statecharts* in den einzelnen Ebenen. Dabei wird eine Ebene durch den „Inhalt“ eines `ORStates` oder einer `Region` definiert, der sich aus den Unterzuständen (`subnodes`) und deren ausgehenden Transitionen zusammensetzt. Die eingehenden Transitionen müssen nicht berücksichtigt werden, weil sie gleichzeitig ausgehende Transitionen eines der Unterzustände sind. Es ist nicht möglich, dass Transitionen von anderen Zuständen eingehen, da der `LayerwiseLayouter` hierarchieübergreifende Transitionen ausschließt.

[1]Dieses *Statechart* ist mit der *KIEL*-Applikation erstellt und anschließend manipuliert worden.

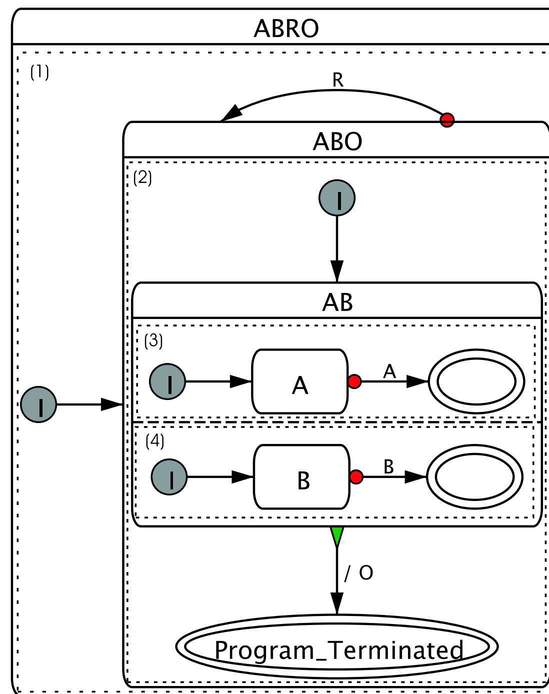


Abbildung 4.3.: Ein *Statechart* mit markierten Hierarchieebenen[2]

Abbildung 4.3 zeigt ein *Statechart*, in dem die einzelnen Ebenen durch Umrandungen hervorgehoben sind. Wie man sieht, besteht das dargestellte *Statechart* aus vier Ebenen, die entsprechend nummeriert sind. Für die Erstellung des Layouts werden zuerst die beiden inneren Ebenen (Ebene 3 und 4), die den „Inhalt" von AB bilden, berechnet. Danach wird das Layout der Ebene 2 vor der Ebene 1 erstellt.

Layout der Zustände

Den zweiten Aufgabenteil, den der `LayerwiseLayouter` übernimmt, bildet das Layout der einzelnen Zustände. Dabei werden Höhe und Breite der Zustände festgelegt. Bei den Pseudozuständen werden diese Größen direkt von den *Rendering*-Einstellungen übernommen. Für die einfachen Zustände geben diese Einstellungen lediglich eine Mindestgröße vor. Sollte die Namenslänge eines Zustands die Mindestbreite überschreiten, wird der Zustand verbreitert, bis der Name ausreichend Platz findet.

Bei den erweiterten Zuständen (`CompositeState`) wird nicht nur die Namenslänge berücksichtigt, sondern auch die Ausmaße ihres „Inhalts". Für `ORStates` und `Regions` ergeben sich diese Ausmaße direkt aus dem Layout der innenliegenden Ebene. Für `ANDStates` müssen zuvor die einzelnen `Regions` platziert und durch `DelimiterLines` optisch getrennt werden.

[2]Dieses *Statechart* ist mit der *KIEL*-Applikation erstellt und anschließend manipuliert worden.

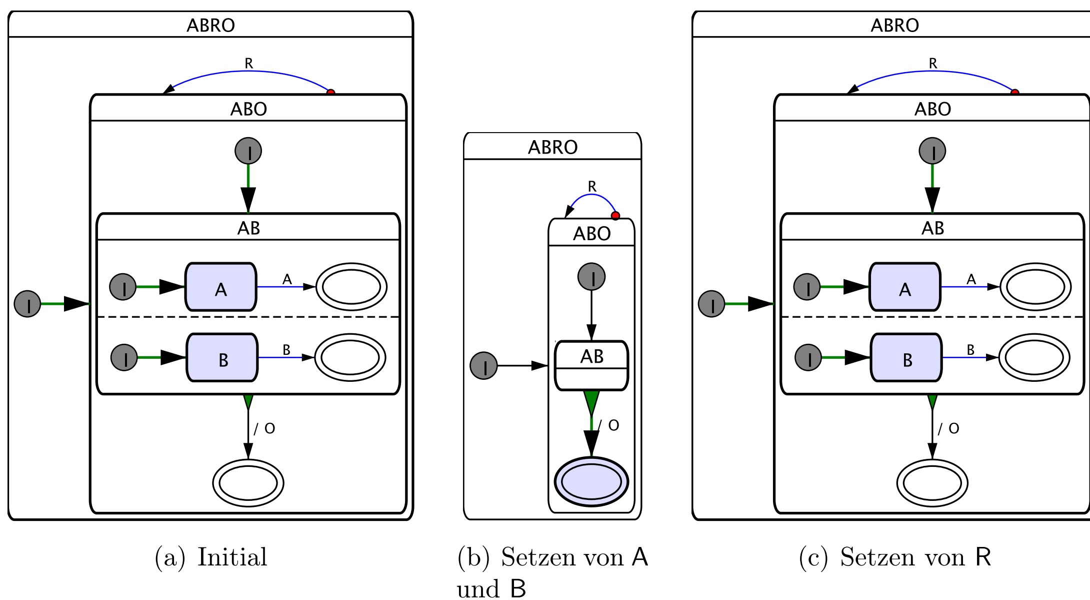
Dieses Anordnen wird ebenfalls vom `LayerwiseLayouter` übernommen. Falls die Regionen in vertikaler Richtung angeordnet werden sollen, platziert der Layouter sie untereinander und trennt jeweils zwei Regionen durch eine Trennlinie. Die Breite des Zustands wird so gewählt, dass sowohl der Name als auch die breiteste Region Platz finden. Die Höhe ergibt sich aus der Summe der Regionshöhen. Dabei werden die Regionen abhängig von den Einstellungen in den `LayouterProperties` (siehe Anhang A) platziert. Die möglichen Einstellungen sind

- zentriert,
- linksbündig und
- rechtsbündig.

Das Layout in horizontaler Richtung verläuft in entsprechender Weise.

Semantischer Zoom

Als drittes Merkmal ist die Unterstützung vom *Semantischem Zoom* zu nennen. Der *Semantische Zoom* ist eine Möglichkeit, die Lesbarkeit von großen *Statecharts* zu erhöhen, indem weniger wichtige Informationen ausgeblendet werden. Diese und weitere Möglichkeiten, die Lesbarkeit von *Statecharts* zu erhöhen, sind in dem Bericht von Prochnow und Hanxleden [37] nachzulesen.



(a) Initial (b) Setzen von A und B (c) Setzen von R

Abbildung 4.4.: Simulationsabfolge mit *Semantischem Zoom*

Der *Semantische Zoom* wird im Projekt *KIEL* ausschließlich während der Simulation eines *Statecharts* eingesetzt, weil nur in diesem Kontext automatisch entschieden werden kann, welche Informationen für den Benutzer wichtig sind.

Für die dabei auftretenden Konfigurationen des *Statecharts* wird eine Ansicht geliefert, die die aktuellen Gegebenheiten der Konfiguration berücksichtigt. Auf diese Weise werden Hierarchieebenen ausgeblendet, wenn sowohl alle Zustände als auch alle Unterzustände einer Ebene inaktiv sind. Abbildung 4.4 zeigt die Simulationsabfolge eines *Statecharts*, in der der „Inhalt“ des Zustands AB durch den *Semantischen Zoom* ausgeblendet wird.

Damit nicht für jede Konfiguration eine eigene Ansicht hinterlegt werden muss, wertet der `LayerwiseLayouter` die Konfigurationen danach aus, welche Zustände eingeklappt werden können. Ein Zustand wird als eingeklappt bezeichnet, wenn seine Unterzustände ausgeblendet und seine Größe auf die eines „einfachen“ Zustands reduziert wird. Für den Fall, dass in zwei Konfigurationen unterschiedliche Zustände aktiv sind, aber dennoch die gleichen Zustände eingeklappt werden können, resultiert daraus eine für den Layouter gleichwertige Ansicht. Das Layout ist dabei nur für die Größe eines Zustands, nicht aber für seine farbliche Markierung relevant.

Damit jede Ansicht nur einmal erstellt werden muss, hinterlegt der `LayerwiseLayouter` alle bereits angefertigten Ansichten. Diese werden zusammen mit einer Auflistung der jeweils eingeklappten Zustände in einer Tabelle gespeichert. Somit beschränkt sich der Arbeitsaufwand von der Summe aller Konfigurationsmöglichkeiten auf die Summe der daraus resultierenden Ein- und Ausklappmöglichkeiten.

Die soeben vorgestellte Layouter-Klasse bietet eine gute Basis für weitere Entwicklungen, weil ein Großteil des immer wiederkehrenden Layout-Prinzips bereits implementiert ist. Alle von dieser Klasse abgeleiteten Layouter müssen nur noch das Layout für eine Ebene erzeugen. Das erleichtert insbesondere die Anbindung von Graphzeichenalgorithmen, da Graphen ein anderes Hierarchieverständnis haben als *Statecharts*. Bei Graphen gibt es schließlich keine Knoten innerhalb eines anderen Knoten.

4.3.2. Der GraphvizLayouter

Auch der `GraphvizLayouter` ist kein eigenständiger Layouter. Er stellt vielmehr die Basis für die Anbindung des Projekts *GraphViz*[3] bereit, das in Abschnitt 3.3 detailliert beschrieben wurde. Da diese Klasse vom zuvor vorgestellten `LayerwiseLayouter` abgeleitet ist, wird nur das Layout einer einzelnen Hierarchieebene betrachtet. Darüber hinaus sind die Breiten und Höhen der Zustände dieser Ebene bereits festgelegt, so dass nur noch die Koordinaten der einzelnen Elemente berechnet werden müssen.

[3]Die Entwicklung dieser Klasse stützt sich auf die *GraphViz*-Version 2.2.1

Erstellen des *GraphViz*-Graphen

Dafür wird zuerst ein *GraphViz*-Graph erstellt, der die Eingabe für das jeweilige Layout-Verfahren bildet. Anschließend werden diesem Graphen globale Einstellungen für den Graphen selbst, seine Knoten und seine Kanten übergeben.

Zu den Grapheneinstellungen gehören Parameter wie Layout-Richtung und Auflösung. Da die Breiten- und Höhenangaben innerhalb des *GraphViz*-Graphen in Zoll (*inch*) und im Projekt *KIEL* in Punkten definiert werden, wird die Auflösung (*dots per inch (dpi)*) für die Umrechnung benötigt.

Für die Knoten wird deren Schriftart und -größe global definiert. Ein weiterer Parameter (`fixedsize`) legt fest, dass die gewählten Knotengrößen nicht vom Layout-Verfahren verändert werden.

Bei den Kanten werden neben der Schriftart und -größe auch noch Einstellungen für die relative Positionierung der Prioritäten angegeben. Insbesondere die Angabe der zu verwendenden Schrift ist sehr wichtig, weil sich hieraus die Ausdehnung der einzelnen *Label* ergibt. Diese Ausdehnung wird beim Layout der Ebene berücksichtigt und sollte ausreichend Platz für die Darstellung der *Label* im `Browser` bzw. `Editor` bieten. Deshalb sollten `Browser`, `Editor` und `Layouter` hierfür die gleichen Schriften verwenden.

Einfügen der Knoten

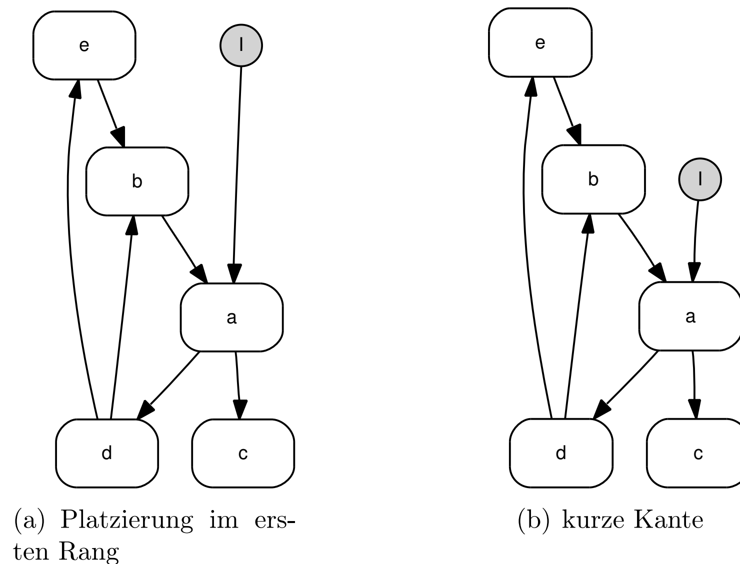
Anschließend wird für jeden Zustand der Ebene ein Knoten in den Graphen eingefügt. Dabei wird der initiale Zustand vor den übrigen Zuständen erstellt, damit dieser bevorzugt links bzw. oben platziert wird. Zwar könnte der initiale Zustand auch fest in dem ersten Rang des Layouts verankert werden, doch würde dies unter bestimmten Umständen dazu führen, dass das Layout kein optimales Ergebnis mehr liefert. Auf diesem Weg wird lediglich eine Priorität dafür vergeben, dass der initiale Zustand links bzw. oben platziert wird, aber keine harte Forderung gestellt.

Abbildung 4.5 zeigt anhand eines Beispiels, dass die Länge der Kante, die vom initialen Zustand ausgeht, höher zu bewerten ist als die Positionierung des initialen Zustands im obersten Rang. Da der initiale Zustand nur einmal pro Ebene definiert sein kann, ist er auf jeden Fall leicht zu erkennen. Die zwingende Positionierung im obersten Rang erhöht die Lesbarkeit weniger, als eine unnötig lange Kante die Lesbarkeit verschlechtert.

Jedem Knoten wird neben seiner Größe, die vom `LayerwiseLayouter` festgelegt wurde, auch eine Form zugewiesen. Die Form eines Zustands sollte, ähnlich der Schrift der *Label*, ebenfalls der späteren Darstellung im `Browser` bzw. `Editor` ähneln, damit die berechneten Anfangs- und Endpunkte der Transitionen später auf dem Rand ihrer Knoten liegen und keine Lücken bzw. Überschneidungen auftreten.

Bei der Definition der Form eines Zustands wird zwischen drei Zustandstypen unterschieden, die wie folgt dargestellt werden:

- Pseudozustände als Kreis,



(a) Platzierung im ersten Rang

(b) kurze Kante

Abbildung 4.5.: Platzieren des initialen Zustands

- finale Zustände als Ellipsen und
- die restlichen Zustände als Rechtecke.

Die *GraphViz*-Darstellung der Knoten lässt sich bei möglichen Veränderungen der *KIEL*-Darstellung leicht über die Einstellungen der `LayouterProperties` anpassen, wie in Anhang A beschrieben.

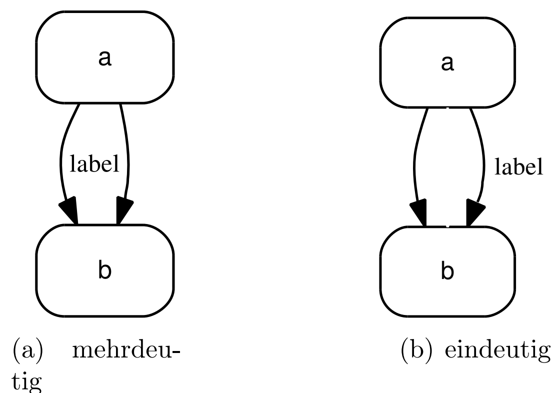
Einfügen der Kanten

Die Definition der Kanten schließt die Abbildung der Ebene in einen *GraphViz*-Graphen ab. Dafür werden die Transitionen zunächst in Transitionen mit *Label* und Transitionen ohne *Label* unterteilt. Diese Gruppen werden außerdem noch anhand ihrer Prioritäten sortiert.

Diese Sortierung ist wieder als eine Art Priorität zu verstehen, die das Layout insofern beeinflusst, als dass die Reihenfolge ihrer Definition der einzige Unterschied zwischen gleichwertigen Transitionen ist. Somit werden gleichwertige Transitionen in der Reihenfolge ihrer Prioritäten angeordnet.

Anschließend werden die Transitionen der beiden Gruppen abhängig von der Layout-Richtung nacheinander erstellt. Im horizontalen Fall werden die Transitionen mit *Label* zuerst erstellt, gefolgt von den Transitionen ohne *Label*. Im vertikalen Fall wird diese Reihenfolge umgekehrt.

GraphViz platziert die ausgehenden Kanten von oben nach unten bzw. von links nach rechts am entsprechenden Wurzelknoten. Das *Label* einer Kante wird entweder oberhalb oder rechts des Kantenverlaufs platziert. Aus diesem Vorgehen von *GraphViz* und der Gruppierung der Transitionen ergibt sich eine verbesserte Zuordnung der *Label* zu ihren Transitionen, die auch ohne dieses Hintergrundwissen für den



(a) mehrdeutig

(b) eindeutig

Abbildung 4.6.: Zuordnungsproblem von *Label* und Transition

Betrachter eindeutig ist. Abbildung 4.6 verdeutlicht dieses Zuordnungsproblem. Im linken Graphen (Abb. 4.6(a)) lässt sich das *Label* keiner Kante eindeutig zuordnen, aber im rechten Graphen (Abb. 4.6(b)) gehört es eindeutig zur rechten Kante.

Neben der Zuweisung von *Label* und Priorität kann der Benutzer über Einstellungen in den `LayouterProperties` jeder Transition noch eine Gewichtung zuweisen, die vom Layout-Verfahren direkt unterstützt wird (siehe Abschnitt 3.3.3). Über die Einstellungen kann das Gewicht von initialen Transitionen erhöht werden. Das führt dazu, dass der Abstand zwischen initialem Zustand und dem Zielzustand möglichst gering ausfällt. Darüber hinaus lässt sich das Gewicht von Transitionen mit *Label* verringern, so dass diese im Vergleich zu ansonsten gleichwertigen Transitionen ohne *Label* länger ausfallen.

Des Weiteren kann der Benutzer vorgeben, dass Transitionen ohne *Label* ein definierbares *Label* zugewiesen wird. Dieses vergrößert den Abstand zwischen zwei parallelen Transitionen und erhöht gleichzeitig die Lesbarkeit.

Da der `GraphvizLayouter` nur die Basis für die Anbindung des Projekts *GraphViz* bildet, kann das Erstellen der Knoten und Kanten, sowie die Zuweisung der Parameter, nicht direkt geschehen. Diese Aufgaben werden von den abgeleiteten Klassen übernommen, die in den folgenden Abschnitten beschrieben sind. Gleiches gilt für die Ausführung des Layout-Verfahrens und das Auslesen der erstellten Layout-Informationen.

4.3.3. Der GraphvizBinaryLayouter

Auch der `GraphvizBinaryLayouter` ist kein eigenständiger Layouter. Vielmehr konkretisiert er den Zugriff des `GraphvizLayouters` auf das Projekt *GraphViz*, indem er die ausführbaren Layout-Programme nutzt. Diese Art des Zugriffs wurde in Abschnitt 3.3.2 beschrieben.

Für die Arbeit des `GraphvizBinaryLayouters` wird das entsprechende Layout-Programm als separater Prozess im Hintergrund gestartet. Dieser Prozess liest über seinen Eingabestrom die Graphdaten im *Dot*-Format, führt das Layout durch, sobald

die Eingabe des Graphen abgeschlossen ist und gibt dann über seinen Ausgabestrom den Graphen im *Attributed-Dot*-Format aus. Anschließend wartet der Prozess auf neue Eingabedaten.

Erstellen der Einträge

Das Erstellen der einzelnen Knoten und Kanten des *GraphViz*-Graphen wird vom `GraphvizLayouter` initiiert, wobei die Attribute der jeweiligen Komponente als eine Liste von Paaren übergeben werden. Diese Paare bestehen aus einem Attributnamen und dem zu setzenden Wert. Der `GraphvizBinaryLayouter` generiert daraus einen Eintrag im *Dot*-Format, der direkt in den Eingabestrom des Hintergrundprozesses geschrieben wird.

Da jeder Knoten und jede Kante über eine eindeutige Identifikation (*ID*) verfügt, wird diese genutzt, um einen eindeutigen Zusammenhang zwischen *KIEL*-Objekt und *Dot*-Eintrag zu gewährleisten. Bei einem Knoten kann diese Zuordnung über den vorangestellten Bezeichner geschehen. Für eine Kante muss das für das Layout unwichtige Attribut `comment` mit der *ID* belegt werden. Da der vorangestellte Bezeichner einer Kante sich aus Start- und Zielknoten zusammensetzt, ist die Zuordnung im Fall von parallelen Kanten nicht mehr eindeutig.

Der Eintrag für einen Knoten hat dabei folgendes Format:

"*Knoten ID*" [*attribut1=wert1, attribut2=wert2,...*];

Der Eintrag für eine Kante hat dieses Format:

"*Start ID*" -> "*Ziel ID*" [**comment**="*Kanten ID*", *attribut2=wert2,...*];

Die Ausgabe im *Attributed-Dot*-Format hat einen äquivalenten Aufbau.

Auslesen der Daten

Da jeder Knoten zwar seine eigene *ID* kennt, aber über die *ID* der Knoten nicht ermittelbar ist, erstellt der Layouter eine Liste für die Verknüpfung von *ID*-`String` und *KIEL*-Objekt. Diese Liste wird beim Erstellen der Knoten und Kanten angelegt und beim anschließenden Auslesen der Layout-Informationen aus der *Attributed-Dot*-Ausgabe des Prozesses benötigt.

Bei der Auswertung der Prozessausgabe geht der Layouter zeilenweise vor. Dabei werden die Zeilen anhand ihres Musters ausgewertet. Auf diese Weise wird unter anderem die Ausdehnung des Graphen ermittelt.

Wenn eine Zeile dem Format eines Knoteneintrags entspricht, wird zunächst die ID des Knoten aus dem Bezeichner gelesen und das zugehörige *KIEL*-Objekt mit Hilfe der Liste ermittelt. Anschließend wird der Positionseintrag (`pos`) in der zugehörigen Attributsliste gesucht und die darin enthaltene Position in die Layout-Information des entsprechenden Zustands geschrieben.

Entspricht eine ausgelesene Zeile dem Format eines Kanteneintrags, wird zunächst der Kommentareintrag (`comment`) der Attributsliste gesucht und das *KIEL*-Objekt

für die hinterlegte *ID* ermittelt. Anschließend werden die Layout-Informationen für Kantenverlauf, Position des *Labels* und Prioritätsposition übernommen. Sobald eine Zeile mit einer schließenden geschweiften Klammer beginnt, ist der Graph und somit auch das Auslesen der Layout-Informationen abgeschlossen.

Das *Parser*-Generator-Paar

Das eben beschriebene Verfahren benötigt einen Generator zum Schreiben der Eingabe im *Dot*-Format und einen *Parser* zum Lesen der Ausgabe im *Attributed-Dot*-Format. Das in *Java* implementierte, frei verfügbare Projekt *Grappa* [24] beinhaltet ein solches *Parser*-Generator-Paar. In einer früheren Phase der Entwicklung sind mit der Anbindung von *Grappa* einfach und schnell Ergebnisse erzielt worden. Allerdings hat sich im späteren Verlauf herausgestellt, dass nicht alle Attribute vom *Parser* des Projekts *Grappa* unterstützt worden sind. Aus diesem Grund wurde das *Parser*-Generator-Paar direkt in den `GraphvizBinaryLayouter` eingebettet und auf die weitere Verwendung von *Grappa* verzichtet.

Der `GraphvizBinaryLayouter` implementiert alle Funktionen für die Kommunikation mit einem ausführbaren Layout-Programm, das als separater Prozess im Hintergrund läuft. Damit die Plattformunabhängigkeit des in *Java* entwickelten Projekts *KIEL* erhalten bleibt, müssen für jede Plattform, die unterstützt werden soll, die ausführbaren Programme hinterlegt werden.

Die einzelnen *GraphViz*-Programme *Dot*, *Neato*, *Twopi* und *Circo* sind durch die entsprechenden Layouter-Klassen `DotBinLayouter`, `NeatoBinLayouter`, `TwopiBinLayouter` und `CircoBinLayouter` an das Projekt *KIEL* gebunden. Diese Klassen erteilen dem `GraphvizBinaryLayouter` Auskunft darüber, auf welche Weise ihr jeweiliges Programm zu starten ist. Eine Erweiterung der Klassen zum Setzen von programmspezifischen Graphattributen wäre denkbar. Die Anbindung von zukünftigen *GraphViz*-Programmen wäre entsprechend leicht umzusetzen.

4.3.4. Der GraphvizLibraryLayouter

Der `GraphvizLibraryLayouter` ist das Analogon zum `GraphvizBinaryLayouter` und somit auch kein eigenständiger Layouter. Allerdings konkretisiert er den Zugriff des `GraphvizLayouter` auf das Projekt *GraphViz*, indem er direkt auf die Funktionen der *GraphViz*-Bibliothek zugreift. Diese Art des Zugriffs wurde in Abschnitt 3.3.2 detailliert beschrieben.

Da das Projekt *KIEL* in *Java* entwickelt wurde, das Projekt *GraphViz* aber eine *C++*-Bibliothek bereitstellt, können die Funktionen nicht direkt benutzt werden. Für solche Fälle ist das **J**ava **N**ative **I**nterface (*JNI*)[45] entwickelt worden, das den Zugriff auf *C++*-Funktionen aus einer *Java*-Umgebung heraus ermöglicht. Der umgekehrte Weg ist mit dem *JNI* ebenfalls möglich, allerdings für den Einsatz im Projekt *KIEL* nicht von Bedeutung.

Die Klasse `GraphvizAPI`, die Teil des *GraphViz*-Layouter-Pakets ist, nutzt das *JNI*, um eine Reihe von Funktionen der *GraphViz*-Bibliothek im *Java*-Umfeld bereit-

```
182     protected static native int createEdge(final int graph,
183             final int source, final int target);
```

Abbildung 4.7.: Deklaration einer *JNI*-*Java*-Funktion zum Erstellen eine Kante

```
50   * Signature: (III)I
51   */
52  JNIEXPORT jint JNICALL
53    Java_kiel_layouter_graphviz_GraphvizAPI_createEdge
54      (JNIEnv *env, jclass obj, jint graph, jint source, jint target)
55  {
56      Agedge_t *edge;
57
58      edge = agedge((Agraph_t*) graph, (Agnode_t*) source,
```

Abbildung 4.8.: *JNI*-*C++*-Funktion zum Erstellen eine Kante

zustellen. Die eigentliche Anbindung findet dabei in den korrespondierenden *C++*-Funktionen statt, die ihrerseits die *GraphViz*-Funktionen aufrufen. Abbildung 4.7 zeigt den *JNI*-Methodenaufruf zum Erstellen einer Kante aus der *Java*-Umgebung. Die Implementierung dieser Methode erfolgt in einer separaten *C++*-Datei. Das entsprechende *Code*-Fragment ist in Abbildung 4.8 dargestellt.

Viele der *C++*-Funktionen erwarten als Parameter bzw. liefern als Rückgabewert einen `Pointer` auf Elemente der internen Struktur. Da *Java* als rein objektorientierte Programmiersprache keine *Pointer* kennt, werden diese `Pointer` als eine natürliche Zahl betrachtet und im *Java*-Umfeld als `int` gespeichert.

Für das Erstellen der Knoten und Kanten, welches vom `GraphvizLayouter` initiiert wird, werden die von der `GraphvizAPI` bereitgestellten Funktionen genutzt. Die Attribute werden ebenfalls mit Hilfe von entsprechenden Funktionen gesetzt. Dabei stellt die `GraphvizAPI` sicher, dass zuvor nicht genutzte Attribute zuerst deklariert werden.

Um die unterschiedlichen Datenstrukturen einander zuordnen zu können, pflegt der Layouter eine Liste mit Verknüpfungen zwischen *KIEL*-Objekt und einer natürlichen Zahl (`int`), die dem `Pointer` auf das *GraphViz*-Objekt entspricht. Da der `Pointer` eindeutig ist, muss der Umweg über die *ID*, wie es beim `GraphvizBinaryLayouter` der Fall ist, nicht gegangen werden.

Auch für das Auslesen der Attribute stellt die `GraphvizAPI` Funktionen bereit. Auf diese Weise kann für jedes *KIEL*-Objekt dessen Positionsangabe in die Layout-Information übernommen werden.

Der `GraphvizLibraryLayouter` implementiert alle Funktionen für die Kommunikation mit der *GraphViz*-Bibliothek. Damit die Plattformunabhängigkeit des in *Java* entwickelten Projekts *KIEL* erhalten bleibt, müssen für jede Plattform, die unterstützt werden soll, die zugehörigen *GraphViz*-Bibliotheken hinterlegt und die *C++*-Datei der `GraphvizAPI` entsprechend kompiliert sein.

Die einzelnen *GraphViz*-Layout-Verfahren *Dot*, *Neato*, *Twopi* und *Circo* sind durch die entsprechenden Layouter-Klassen `DotLibLayouter`, `NeatoLibLayouter`, `TwopiLibLayouter` und `CircoLibLayouter` an das Projekt *KIEL* gebunden. Diese Klassen rufen ihre jeweiligen Layout-Funktionen und Reinigungsfunktionen auf. Eine Erweiterung der Klassen zum Setzen von programmspezifischen Graphattributen wäre denkbar.

Die Anbindung von zukünftigen *GraphViz*-Layout-Verfahren wäre entsprechend leicht umzusetzen. Auch die Nutzung von Teilfunktionen der Layout-Algorithmen wäre denkbar und böte ein großes Potential für zukünftige Entwicklungen.

4.3.5. Der HVLayouter

Der `HVLayouter` ist ein vollwertiger Layouter, der ohne weitere Ableitungen einsetzbar ist. Da er vom `LayerwiseLayouter` abgeleitet ist, muss nicht das gesamte *Statechart* betrachtet werden, sondern lediglich eine Ebene des *Statecharts*. Das generelle Layout-Verfahren ist im Rahmen der Studienarbeit Kloss [28] für die Verwendung unter ArgoUML [1] entwickelt worden.

Im Rahmen dieser Arbeit wurde das Layout-Verfahren nicht nur an die *KIEL*-Datenstruktur angepasst, sondern wurde in manchen Punkten sogar erweitert. Zum einen werden beim Layout nun auch Prioritäten berücksichtigt und zum anderen wurde die Darstellung von parallelen Zuständen auf Grund der Ableitung vom `LayerwiseLayouter` verbessert.

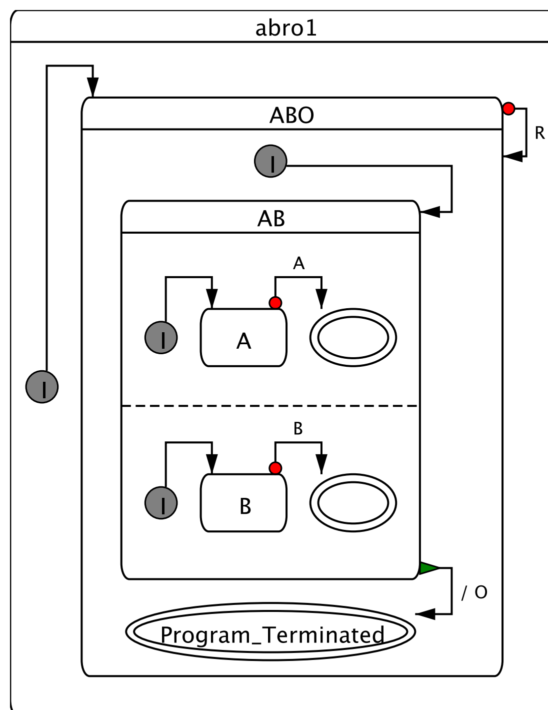


Abbildung 4.9.: Ein *Statechart* im HVLayout

Das Verfahren des `HVLayouters` sieht eine schichtweise Einteilung der Komponenten vor. Dabei werden die Zustände auf einer mittleren Schicht nebeneinander angeordnet. Für die horizontale Ausrichtung werden in den darüber liegenden Schichten die ausgehenden Transitionen eines Zustands platziert und in den darunter liegenden Schichten die eingehenden Transitionen. Eigentransitionen werden an den Seiten eines Zustands platziert. Analog dazu verhält sich die vertikale Ausrichtung.

Bei der Anordnung der Zustände werden die folgenden Gütekriterien berücksichtigt:

- Positionierung des initialen Zustands am Rand,
- Minimierung der Transitionskreuzungen,
- Minimierung der Schichtenanzahl und
- Ausgewogenheit der Transitionsschichten.

Abbildung 4.9 zeigt ein *Statechart* im HVLayout. Wie man sieht, verlaufen alle Transition in einer einheitlichen Richtung, entweder von links nach rechts oder von oben nach unten. Die Eigentransition mit dem *Label* R ist am Rand des Zustands ABO so platziert, dass die orthogonal zur Laufrichtung der eingehenden Transition (init → ABO) verläuft. Die „offenen“ Enden der Transitionen zu einem finalen Zustand entstehen deshalb, weil der `HVLayouter` ausschließlich auf der Basis von rechteckigen Zuständen arbeitet.

4.4. Zusammenfassung

In diesem Kapitel wurde die Entwicklung eines Layout-Moduls für den Einsatz im Projekt *KIEL* beschrieben. Zu diesem Zweck wurde zunächst eine gemeinsame Basis geschaffen, die zugleich eine Schnittstelle für den Zugriff der anderen Module bereitstellt. Diese Schnittstelle umfasst wegen der geforderten Kapselung sehr wenige Klassen, so dass mögliche Erweiterungen des Moduls die bestehenden Module nicht negativ beeinflussen. Neuentwickelte Layouter müssen lediglich in einer Tabelle des `Handlers` eingetragen werden, damit sie für die Verwendung innerhalb der *KIEL*-Applikation zur Verfügung stehen.

Für die gemeinsame Datenhaltung der Einstellmöglichkeiten der einzelnen Layouter wurde die Klasse `LayouterProperties` entwickelt. Diese aktualisiert während einer Sitzung laufend die benutzerspezifischen Einstellungen, so dass diese sofort, ohne einen Neustart der Anwendung, den Layout-Prozess beeinflussen.

Für die Einbettung zukünftiger Layout-Verfahren wurden verschiedene Layouter-Klassen erstellt, die bereits grundlegende Aufgaben übernehmen. Zu diesen gehört unter anderem der `LayerwiseLayouter`, der ein gegebenes *Statechart* in dessen Hierarchieebenen unterteilt, so dass nur noch ein Layout für die einzelnen Ebenen erstellt werden muss. Darüber hinaus wurden diverse Layouter-Klassen erstellt, die eine der

Basisklassen soweit konkretisieren, dass sie das vollständige Layout eines *Statecharts* berechnen können.

Im Rahmen dieser Arbeit wurden somit fünf unterschiedliche Layout-Verfahren in das Projekt *KIEL* integriert. Hierbei lag das Hauptaugenmerk der Entwicklung auf der Einbettung des *Dot*-Layouts. Aus diesem Grund sind alle Parametereinstellungen auf dieses Verfahren zugeschnitten. Diese Einstellungen sind auch für die anderen *GraphViz*-Layout-Varianten nutzbar, obwohl hierfür keine Optimierung vorgenommen wurde. Im folgenden Kapitel 5 werden die Ergebnisse dieser Arbeit erörtert, dabei werden erreichte Ziele und offene Probleme gegenüber gestellt.

5. Zusammenfassung

In diesem Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst. Zunächst wird im folgenden Abschnitt anhand eines Fallbeispiels die Arbeitsweise des `Dot-Layouters` verdeutlicht. Im Anschluß daran werden in Abschnitt 5.2 die erreichten Ziele zusammengefasst und in Abschnitt 5.3 die noch offenen Probleme diskutiert. Abschnitt 5.4 schließt die Arbeit mit einem Ausblick auf zukünftige Entwicklungen ab.

5.1. Fallbeispiel

In diesem Abschnitt wird anhand eines Fallbeispiels die Arbeitsweise des auf dem *GraphViz*-Programm *Dot* basierenden Layouter (`DotBinLayouter`) veranschaulicht. Abbildung 5.1 zeigt zunächst die Ausgangslage des Beispiel-*Statecharts*. Das Resultat der Layout-Berechnung ist in Abbildung 5.2 zu sehen. Die Neuanordnung der Elemente hat eine klare Struktur geschaffen, die leichter lesbar und somit verständlicher ist. Darüber hinaus ist sie kompakter und weist weniger Überlagerungen auf. Die marginalen Unterschiede in den Details der Darstellung sind darauf zurückzuführen, dass die Abbildungen aus technischen Gründen in unterschiedlichen Arbeitsmodi der *KIEL*-Applikation erstellt wurden. Die Vorheransicht wurde im `Editor`-Modus und die Nachheransicht im `Browser`-Modus dargestellt.

Auf welche Weise dieses Resultat erzielt wurde, wird im Weiteren beschrieben und illustriert. Um das Layout eines *Statecharts* zu berechnen, beginnt der Layouter mit dem Wurzelknoten des *Statecharts*. In dem Fallbeispiel ist dies der Zustand ABSyncronization. Da dieser Zustand ein `CompositeState` ist und somit einen „Inhalt“ hat, muss zunächst diese innere Ebene verarbeitet werden. Auf die gleiche Weise arbeitet sich das Layout-Verfahren über den Zustand ABSync bis zu der ersten Region des Zustands WaitAB vor. Da diese Region keine tieferliegenden Hierarchieebenen aufweist, beginnt an dieser Stelle die Arbeit des *GraphViz*Programms *Dot*.

Dafür wird zunächst ein *GraphViz*-Graph im *Dot*-Format erstellt, wie in Abschnitt 3.3.2 bereits beschrieben. Das Resultat ist in Abbildung 5.3 zu sehen. Dieser *GraphViz*-Graph wird in das Programm *Dot* eingegeben, das die Platzierung der Elemente berechnet und ausgibt. Die Ausgabe des Programms ist im Folgenden jeweils grafisch dargestellt. Einzig die Schriftart wurde für die Erzeugung des Bilder angepasst, da die *PostScript*-Ausgabe des Programms *Dot* die verwendete Schriftart nicht unterstützt. Abbildung 5.4(a) zeigt somit die Darstellung der ersten Ebene des Zustands WaitAB in der *GraphViz*-Ansicht. In Abbildung 5.4(b) ist die zweite Region dargestellt.

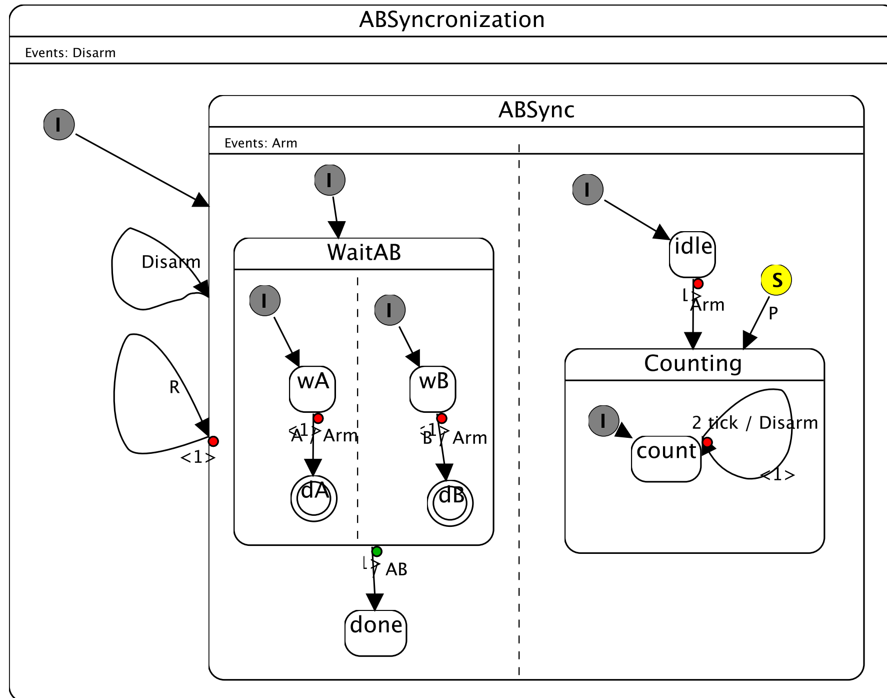


Abbildung 5.1.: Ausgangslage des Fallbeispiels

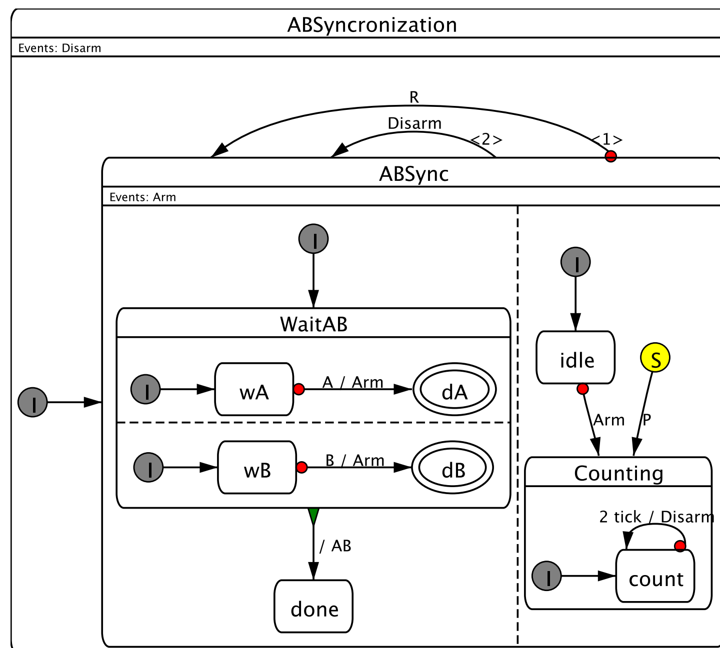
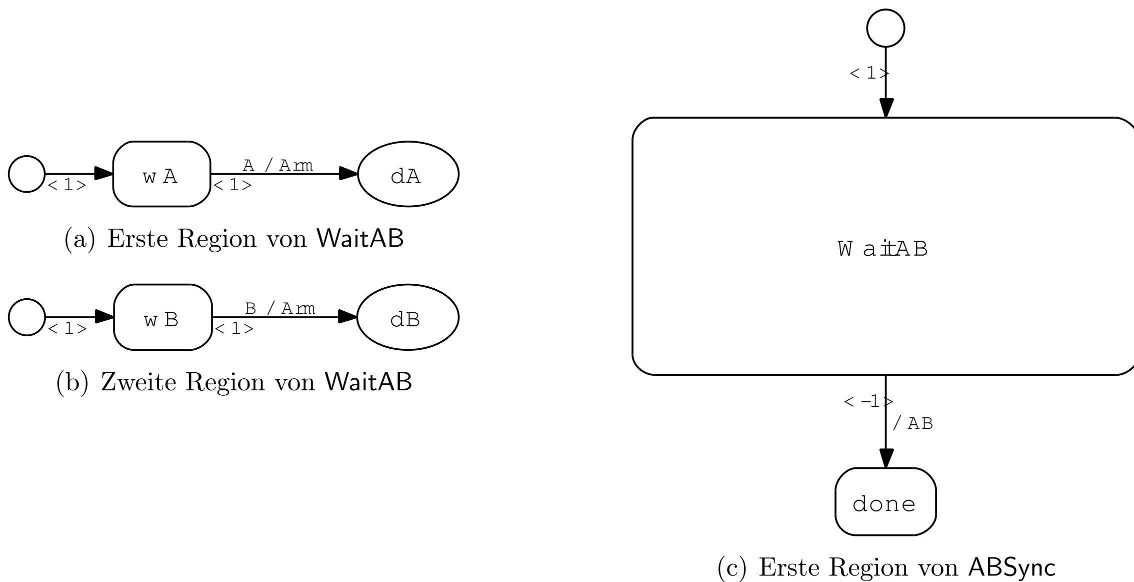


Abbildung 5.2.: Fallbeispiel im *Dot*-Layout

```
1  digraph wAdA {
2  graph [fontname=SansSerif, labeljust=c, rankdir=LR, dpi=72,
      labelloc=t, fontsize=14];
3  node [label="\N", fontname=SansSerif, fixedsize=true, fontsize=14,
      style=rounded];
4  edge [fontname=SansSerif, labelfontsize=11, labeldistance="1.5",
      fontsize=11, labelangle="-20", labelfontname=SansSerif];
5  "#9#init.34" [label="", width="0.28", height="0.28", shape=ellipse
      ];
6  "#11#state.70" [label=wA, width="0.75", height="0.50", shape=box];
7  "#13#state.72" [label=dA, width="0.78", height="0.50", shape=
      ellipse];
8  "#11#state.70" -> "#13#state.72" [label="A / Arm", taillabel="<1>",
       weight="1.0", comment="#23#strong_abortion.36"];
9  "#9#init.34" -> "#11#state.70" [taillabel="<1>", weight="2.0",
      comment="#15#initial.31"];
10 }
```

Abbildung 5.3.: Erste Region des Zustands WaitAB im *Dot*-Format

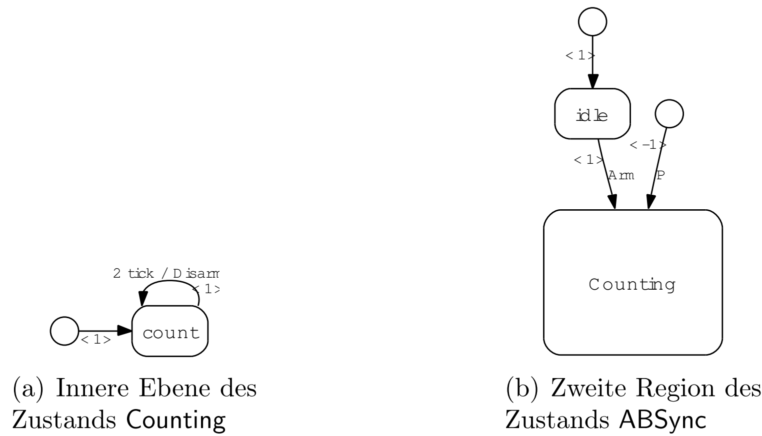


(a) Erste Region von WaitAB

(b) Zweite Region von WaitAB

(c) Erste Region von ABSync

Abbildung 5.4.: Erste Region des Zustands ABSync (komplett)



(a) Innere Ebene des Zustands Counting

(b) Zweite Region des Zustands ABSync

Abbildung 5.5.: Zweite Region des Zustands ABSync (komplett)

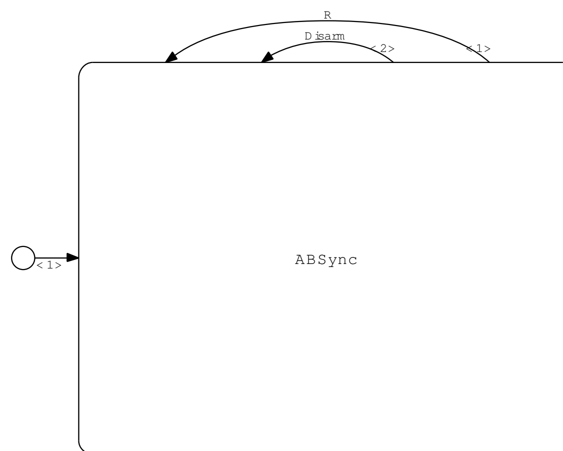


Abbildung 5.6.: Oberste Hierarchieebene des *Statecharts*

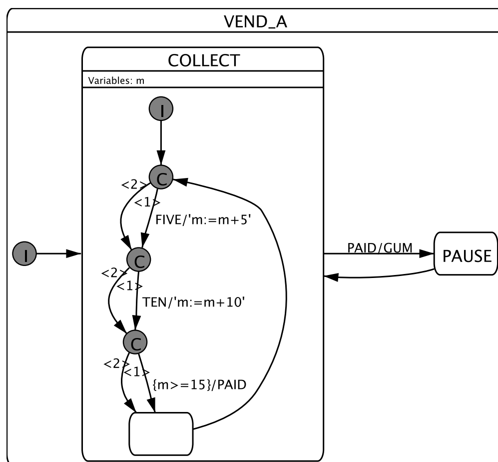
Nachdem die innere Ebene des Zustands WaitAB berechnet wurde, wird das Layout für die erste Region des Zustands ABSync berechnet. Das Resultat dieser Berechnung ist in Abbildung 5.4(c) dargestellt. Wie man sieht ist der Zustand WaitAB so groß gewählt, dass die zuvor berechneten Regionen (siehe Abbildung 5.4(a) und 5.4(b)) in ihm Platz finden.

Anschließend wird die zweite Region des Zustands ABSync auf gleiche Weise berechnet. Die Abbildungen 5.5(a) und 5.5(b) zeigen die jeweiligen Resultate. Abschließend wird das Layout für die oberste Hierarchieebene des *Statecharts* erstellt, wie Abbildung 5.6 zeigt, und somit die Abmessungen des *Statecharts* festgelegt.

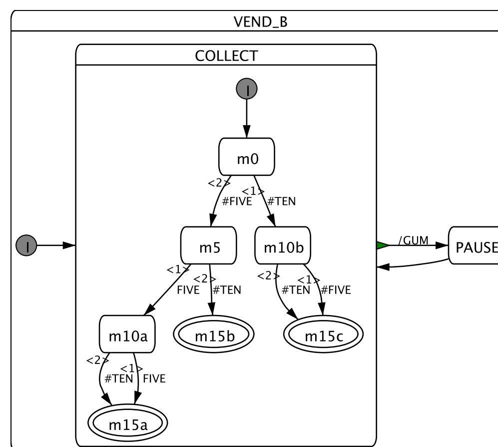
Die Tabelle 5.1 zeigt die Komplexitätsanalyse anhand des vorgestellten Fallbeispiels. Die Messwerte wurden auf einem Linux-PC (SuSE 9.2) mit einem Intel Pentium IV (1.7 GHz) Prozessor und 768 MB RAM ermittelt. Weitere Beispiele werden in den folgenden Abbildungen 5.7(a), 5.7(b), 5.7(c) und 5.7(d) unter Angabe der benötigten Rechenzeit dargestellt. Nach dieser grafischen Demonstration der Arbeits-

	Zustände	Transitionen	*Dot*	*KIEL*	*Dot* vs. *KIEL*
wAdA	3	2	2 ms	8 ms	0.25
wBdB	3	2	3 ms	9 ms	0.3
WaitABdone	3	2	3 ms	10 ms	0.33
count	2	2	4 ms	11 ms	0.36
idleCounting	4	3	6 ms	12 ms	0.5
ABSync	2	3	2 ms	17 ms	0.12
Summen	17	14	20 ms	67 ms	0.3
gesamt				75 ms	

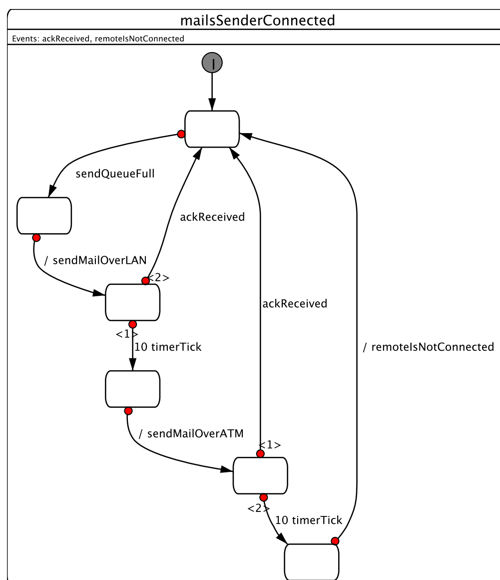
Tabelle 5.1.: Komplexitätsanalyse des Fallbeispiels



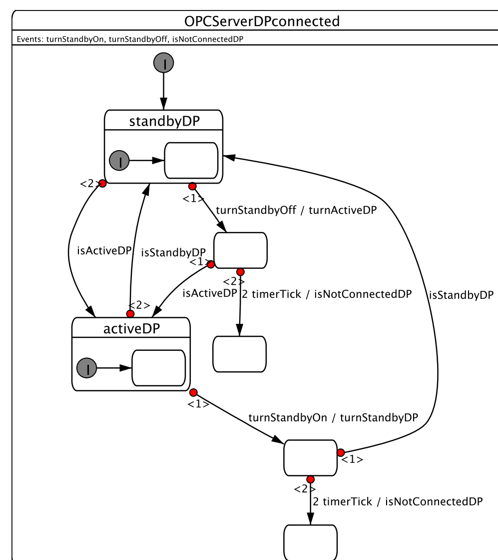
(a) Beispiel 1 (51 ms)



(b) Beispiel 2 (45 ms)



(c) Beispiel 3 (37 ms)



(d) Beispiel 4 (69 ms)

Abbildung 5.7.: Weitere Beispiele

weise werden im kommenden Abschnitt die erreichten Zeile noch einmal schriftlich zusammengefasst.

5.2. Erreichte Ziele

Nachdem die ereichten Ziele anhand des eben gezeigten Fallbeispiels grafisch dargestellt wurden, werden in diesem Abschnitt die wichtigsten Punkte noch einmal in Worten zusammengefasst.

Entwicklung des Layout-Moduls

Grundlage der gesamten Arbeit war die Entwicklung des Layout-Moduls für das Projekt *KIEL*. Das Modul stellt eine breite Basis für die Einbettung weiterer Layout-Verfahren bereit. Da die internen Klassen gegenüber den anderen Modulen des Projekts stark gekapselt sind, lassen sich Erweiterungen ohne Beeinträchtigung der anderen Module vornehmen. Darüber hinaus werden die Einstellmöglichkeiten aller Layouter-Klassen an einer zentralen Stelle vorgehalten, so dass die Parameter zukünftiger Entwicklungen leicht integriert werden können.

Neben der grundlegenden Definition der Schnittstelle wurde eine erweiterte Layouter-Klasse (siehe Abschnitt 4.3.1) entwickelt, die ein generelles hierarchieebenenweises Vorgehen bereitstellt. Auf der Basis dieser Layouter-Klasse lassen sich Layout-Verfahren entwickeln, die ausschließlich das Layout für eine Hierarchieebene berechnen. Insbesondere im Umfeld des Graph-Zeichnens existiert eine Vielzahl an Algorithmen, die sich auf diese Weise leicht integrieren lassen.

Umsetzung des *Semantischen Zoom*

Des Weiteren setzt die eben beschriebene Layouter-Klasse das Konzept des *Semantischen Zoom* um und bildet somit die Grundlage für die Erweiterung von *Statecharts* um dynamische Ansichten. Da alle in dieser Arbeit vorgestellten Layouter-Klassen von dieser Klasse abgeleitet sind, lassen sich mit allen Layout-Verfahren dynamische Ansichten generieren.

Anbindung eines Graph-Zeichen-Werkzeugs

Das Hauptziel dieser Arbeit war die Anbindung eines existierenden Graph-Zeichen-Werkzeugs an das Projekt *KIEL*. Nach der Evaluation diverser Werkzeuge stellte sich das Projekt *GraphViz* als geeignet heraus. Dieses Projekt zeichnet sich durch seine einfache Benutzbarkeit und eine allgemein gehaltene Herangehensweise aus.

Mit Hilfe des *Dot*-Layout-Verfahrens werden gut sturkturierte und somit übersichtliche *Statechart*-Ansichten generiert, die für den Betrachter leichter lesbar sind. In Zusammenhang mit der Bereitstellung von dynamischen Ansichten

während der Simualtion wird auf diese Weise das intuitive Verständnis des Systemverhaltens erhöht.

Einbettung des *HVLayouters*

Im Rahmen dieser Arbeit konnte das Layout-Verfahren aus einer vorherigen Arbeit in das Projekt *KIEL* integriert werden. Neben der Anpassung an die Datenstruktur des Projekts *KIEL* wurde das Verfahren um die Verarbeitung von Prioritäten erweitert. Durch die Einbettung in die entwickelte Klassenstruktur lassen sich auch für dieses Verfahren dynamische Ansichten generieren.

5.3. Offene Probleme

Wie der vorherige Abschnitt zeigt konnten viele Ziele erreicht werden, allerdings sind noch einige wenige Probleme offen geblieben, die an dieser Stelle diskutiert werden.

Hierarchieübergreifende Transitionen

Alle vorgestellten *Statechart*-Layouter schließen *Statecharts* aus, die hierarchieübergreifende Transitionen beinhalten. Derartige Transitionen werden von herkömmlichen Graph-Zeichen-Verfahren nicht berücksichtigt, weil die Knoten eines Graphen keine Unterkonten enthalten und somit auch keine Kanten zwischen Knoten und Unterknoten existieren können.

Ein *Statechart*-Layouter, der *Statecharts* mit hierarchieübergreifenden Transitionen verarbeitet, müsste den Kantenverlauf eigenständig berechnen. Damit die Anzahl der Kantenkreuzungen zwischen hierarchieübergreifenden und „normalen“ Transitionen minimiert wird, müsste der Kantenverlauf auch in der Positionierung der Knoten berücksichtigt werden. Folglich müsste ein Layout-Verfahren eigenständig entwickelt werden, wobei sich dieses an bestehende Graph-Zeichen-Verfahren anlehnen könnte.

Da sich das Projekt *KIEL* in erster Hinsicht um Konformität zu *Esterel Studio* bemüht, ist die Beschränkung auf *Statecharts* ohne hierarchieübergreifende Transitionen vertretbar.

GraphViz-Anbindung über Bibliothekszugriff

Die Anbindung des Projekts *GraphViz* über den direkten Bibliothekszugriff erzeugt Probleme in der Darstellung der Transitionen. Dabei liegen die Anfangs- und Endpunkte einer Eigentransition nicht immer auf dem Rand des entsprechenden Zustands, sondern oftmals in der Mitte des Zustands. Dieses nichtdeterministische Verhalten ist ausschließlich unter dem Betriebssystem *Linux* zu beobachten. Ein solches fehlerhaftes Beispiel ist in Abbildung 5.8 zu sehen, in dem die mit R und Disarm markierten Transitionen das beschriebene Verhalten aufweisen.

Weitere Nachforschungen konnten das Problem nur auf die Verwendung des *JNI* unter dem Betriebssystem *Linux* zurückführen. Dafür wurde ein *C++*-Programm zum Test der *GraphViz*-Bibliothek entwickelt sowie ein gleichwertiges *Java*-Programm, das mittels des *JNI* auf die Bibliothek zugriff. Obwohl

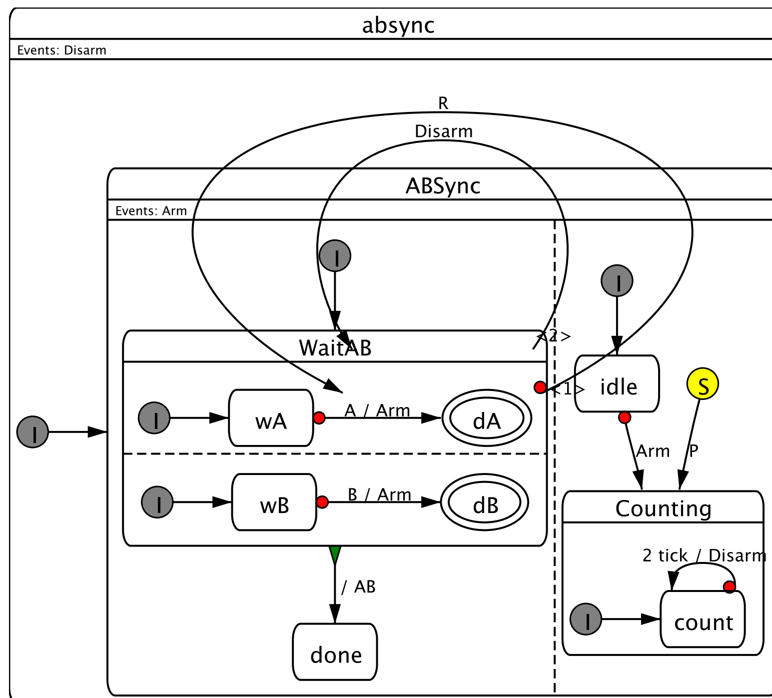


Abbildung 5.8.: Fehlerhaftes Layout mit dem `DotLibLayouter`

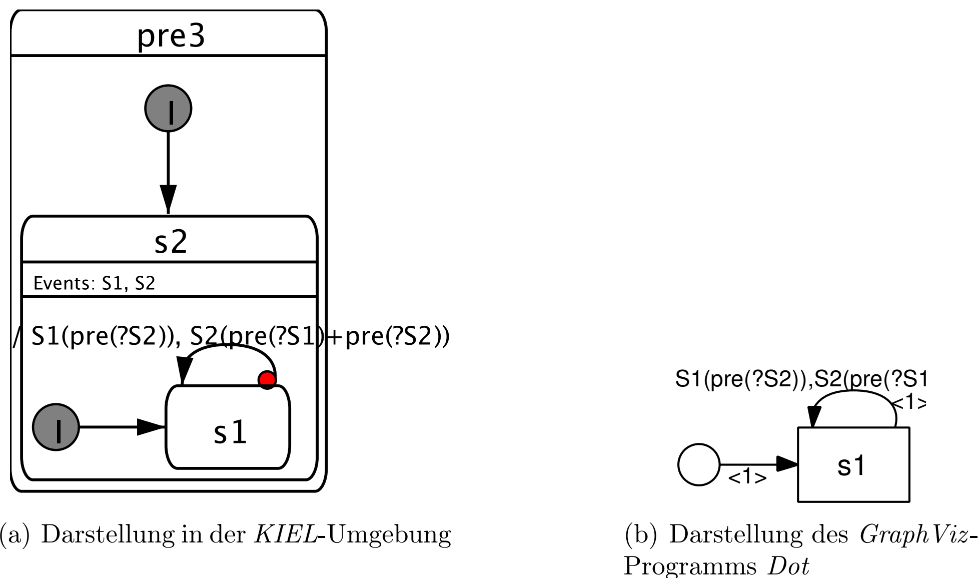
sämtliche Bibliotheksfunktionen von beiden Programmen in derselben Reihenfolge aufgerufen wurden, unterschieden sich die Ergebnisse der Berechnung. Der gleiche Versuch unter dem Betriebssystem *Microsoft Windows* lieferte richtige Ergebnisse.

Von weiteren Nachforschungen wurde aus zeitlichen Gründen abgesehen. Die weitere Analyse des Problems erscheint lohnenswert, weil die Anbindung des Projekts *GraphViz* über die mitgelieferte Bibliothek mehr Möglichkeiten bereithält als der Zugriff über den separaten Programmaufruf. Da diese Möglichkeiten im Rahmen dieser Arbeit allerdings nicht genutzt wurden, stellt der Zugriff über den Programmaufruf zur Zeit ein gleichwertiges Layout-Verfahren bereit.

Transitions-*Label* ragt aus Oberzustand

In manchen *Statechart*-Darstellungen kommt es vor, dass ein Transitions-*Label* aus seinem umgebenden Oberzustand herausragt. Dieser Fehler tritt unter folgenden Bedingungen auf:

- Die Layout-Richtung der Ebene ist von links nach rechts.
- Das *Label* gehört zu einer Eigentransition.
- Der Text des *Labels* ist länger als der zugehörige Zustand breit.



(a) Darstellung in der *KIEL*-Umgebung

(b) Darstellung des *GraphViz*-Programms *Dot*

Abbildung 5.9.: Transitions-*Label* ragt aus Oberzustand

Die Ursache dieses Problems liegt in der Berechnung des umgebenden Rechtecks, das von *GraphViz* unter den oben genannten Umständen falsch berechnet wurde. Abbildung 5.9 zeigt, dass das *Label* in der *GraphViz*-Ansicht (5.9(b)) in der Berechnung des umgebenden Rechtecks nicht berücksichtigt wird. Dieses Fehlverhalten wurde dem Projekt *GraphViz* [23] gemeldet und dort unter der Fehlernummer *669* registriert.

Von den eben beschriebenen Problemen stellt einzig der letztgenannte Punkt eine Beeinträchtigung der *KIEL*-Applikation dar. Dieser Punkt wird allerdings mit einer der kommenden *GraphViz*-Versionen behoben. Die anderen beiden Punkte beschreiben Einschränkungen, die von der *KIEL*-Applikation zur Zeit nicht genutzt werden. Abschließend wird im folgenden Abschnitt ein Ausblick auf mögliche Erweiterungen bzw. Verbesserungen gegeben.

5.4. Ausblicke

Der gegenwärtige Stand des Moduls `Layouter` berechnet gut lesbare Layouts in einer angemessenen Zeit, allerdings lassen sich Feinheiten der erstellten Layouts noch verbessern. An dieser Stelle wird ein kurzer Überblick über mögliche Veränderungen gegeben.

Platzsparenderes Anordnen der Regionen eines parallelen Zustands

In der gegenwärtigen Implementierung übernimmt die Klasse `LayerwiseLayouter` die Anordnung der Regionen eines parallelen Zustands. Das genutzte Verfahren richtet zunächst den „Inhalt“ aller Regionen in derselben Richtung

aus, anschließend werden die Regionen in der orthogonalen Richtung nebeneinander angeordnet.

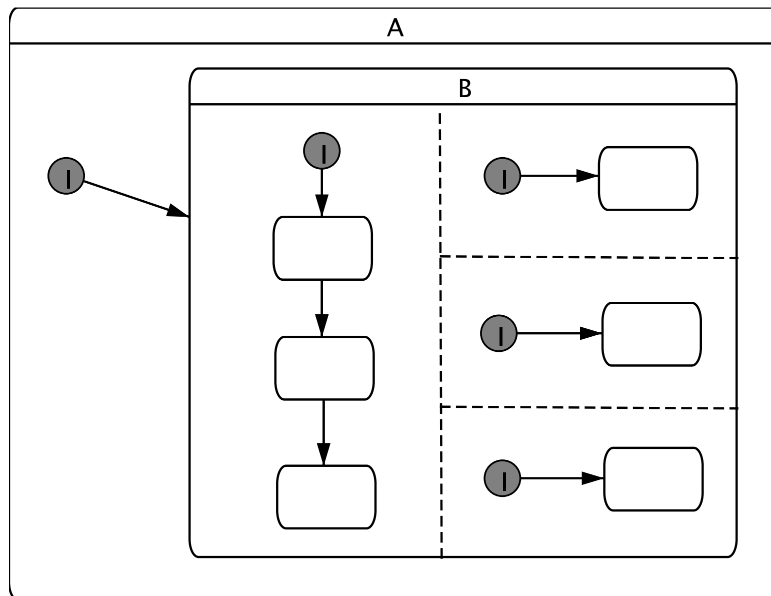


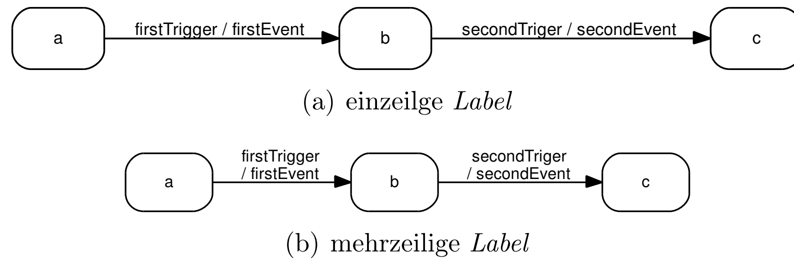
Abbildung 5.10.: Platzsparendere Anordnung der Regionen eines parallelen Zustands

Eine andere Möglichkeit der Anordnung wäre, die Regionen möglichst kompakt zu platzieren. In diesem Fall würden die Regionen nicht in einer Reihe nebeneinander platziert, sondern sowohl neben- als auch übereinander. Abbildung 5.10 zeigt ein *Statechart* der *KIEL*-Applikation, bei dem die Regionen des Zustands B nach dem beschriebenen Verfahren platziert wurden.

Mehrzeilige Transitions-*Label*

Neben der kompakteren Anordnung der Regionen eines parallelen Zustands wäre auch die kompaktere Darstellung der *Label* einer Transition denkbar. *Label* die entlang ihrer Transition verlaufen, beeinflussen deren Länge direkt. Im entgegengesetzten Fall, wenn sie orthogonal zu ihrer Transition verlaufen, beeinflussen sie unter Umständen die Länge anderer Transitionen. Da die Länge einer Transition in vielen Layoutverfahren als ein Ästhetikkriterium bewertet wird, ist diese kurz zu halten. Somit ist die Kompaktheit eines *Labels* als ein indirektes Ästetikkriterium zu betrachten.

Aus internen technischen Gründen des Projekts *KIEL* sind die mehrzeiligen Transitions-*Label* bis zur Fertigstellung dieser Arbeit noch nicht in dem Modul `Layouter` implementiert worden. Da das Projekt *GraphViz* bereits mehrzeilige *Label* unterstützt, wie Abbildung 5.11 zeigt, sollte eine diesbezügliche Erweiterung der *GraphViz*-basierten Layouter-Klasse (`GraphvizLayouter`) leicht möglich sein.



(a) einzeilge *Label*

(b) mehrzeilige *Label*

Abbildung 5.11.: Nutzung mehrzeiliger Transitions-*Label*

Variable Elementgrößen

Um das globale Systemverständnis zu erhöhen, unterstützt das Layout-Modul den *Semantischen Zoom*. Das Ausblenden ganzer Hierarchieebenen könnte durch ein Verkleinern der enthaltenen Elemente ersetzt werden. Somit würde die darin enthaltene Information bei der Betrachtung lediglich eine geringere Gewichtung erhalten, jedoch nicht völlig verschwinden.

In einem ersten Ansatz könnten die Schriftgrößen der einzelnen Elemente verringert werden. Eine Anpassung der Zustandsgrößen und Transitionslängen wäre ein weiterführender Schritt.

Ausblenden von weiteren Informationsdetails

Desweiteren sind Methoden denkbar die weitere Informationen ausblenden. Um einen groben Überblick über ein großes System zu erhalten, sind die Angaben der verwendeten Signale und Variablen der einzelnen Hierarchieebenen von sekundärer Bedeutung. Das Ausblenden dieser Informationen ermöglicht eine kompaktere Darstellung des *Statecharts*, die ein erhöhtes intuitive Systemverständnis ermöglicht.

Nach diesem Überblick über mögliche Erweiterungen folgt im Anschluss eine ausführliche Beschreibung der diversen Einstellmöglichkeiten des Moduls `Layouter`. Im Anhang B ist das vollständige *Code*-Beispiel aus Abschnitt 3.3.2 zu finden, abschließend werden in Anhang C die Quellen der Layouter-Klassen aufgelistet.

6. Literaturverzeichnis

[1] ArgoUML. The ArgoUML project. URL `http://argouml.tigris.org/`. An open source UML design tool.

[2] Auber David—University Bordeaux. Tulip Homepage, 2005. URL `http://tulip-software.org/`.

[3] Carlo Batini, Enrico Nardelli und Roberto Tamassia. A Layout algorithm for data flow diagrams. *IEEE Transactions on Software Engineering*, 12(4):538–546, 1986. ISSN 0098-5589.

[4] Giuseppe Di Battista, Walter Didimo, Maurizio Patrignani und Maurizio Pizzonia. Orthogonal and Quasi-upward Drawings with Vertices of Prescribed Size. In *GD '99: Proceedings of the 7th International Symposium on Graph Drawing*, Seiten 297–310, London, UK, 1999. Springer-Verlag. ISBN 3-540-66904-3.

[5] Franz-Josef Brandenburg, Michael Himsholt und Christoph Rohrer. An Experimental Comparison of Force-Directed and Randomized Graph Drawing Algorithms. In *GD '95: Proceedings of the Symposium on Graph Drawing*, Seiten 76–87, London, UK, 1996. Springer-Verlag. ISBN 3-540-60723-4.

[6] Christoph Buchheim, Michael Jünger und Sebastian Leipert. Improving Walker's Algorithm to Run in Linear Time. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, Seiten 344–353, London, UK, 2002. Springer-Verlag. ISBN 3-540-00158-1.

[7] Cooperation of groups in Halle, Köln, Saarbrücken, and Wien. Algorithms for Graph Drawing Homepage, 2003. URL `http://www.ads.tuwien.ac.at/AGD/`.

[8] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42: 149–160, 1984. ISSN 0384-9864.

[9] Peter Eades. Drawing free trees. In *Bulletin of the Institute for Combinatorics and its Applications*, volume 5, Seiten 10–36, 1992.

[10] Peter Eades, Robert F. Cohen und Mao Lin Huang. Online Animated Graph Drawing for Web Navigation. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, Seiten 330–335, London, UK, 1997. Springer-Verlag. ISBN 3-540-63938-1.

[11] Peter Eades, Brendan D. McKay und Nicholas C. Wormald. On an edge crossing problem. In *9th Australian Computer Science Conference*, Seiten 327–334. ACA, 1986.

[12] Peter Eades und Nicholas C. Wormald. The median heuristic for drawing 2-layered networks. Technical Report 69, University of Queensland, Department of Computer Science, 1986.

[13] Esterel Technologies. Company homepage. `http://www.esterel-technologies.com`.

[14] Ulrich Fößmeier und Michael Kaufmann. Drawing High Degree Graphs with Low Bend Numbers. In *GD '95: Proceedings of the Symposium on Graph Drawing*, Seiten 254–266, London, UK, 1996. Springer-Verlag. ISBN 3-540-60723-4.

[15] Arne Frick, Andreas Ludwig und Heiko Mehldau. A Fast Adaptive Layout Algorithm for Undirected Graphs. In *GD '94: Proceedings of the DIMACS International Workshop on Graph Drawing*, Seiten 388–403, London, UK, 1995. Springer-Verlag. ISBN 3-540-58950-3.

[16] Thomas M. J. Fruchterman und Edward M. Reingold. Graph drawing by force-directed placement. *Software – Practice and Experience*, 21(11):1129–1164, 1991. ISSN 0038-0644.

[17] Emden Gansner, Eleftherios Koutsofios und Stephen North. Drawing graphs with `dot`. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, Februar 2002. URL `http://www.research.att.com/sw/tools/graphviz/dotguide.pdf`.

[18] Emden R. Gansner. Drawing graphs with GraphViz. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, November 2004. URL `http://www.research.att.com/sw/tools/graphviz/libguide.pdf`.

[19] Emden R. Gansner, Yehuda Koren und Stephen C. North. Graph Drawing by Stress Majorization. In *Graph Drawing*, Seiten 239–250, 2004. URL `http://springerlink.metapress.com/openurl.asp?genre=article{\&}issn=0302-9743{\&}volume=3383{\&}spage=239`.

[20] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North und Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3): 214–230, 1993. URL `citeseer.nj.nec.com/gansner93technique.html`.

[21] Michael R. Garey und David S. Johnson. *Computers and Intractibility.* Bell Telephone Laboratories, Inc., Murray Hill, NJ, 1979.

[22] GeomNet. Graph Drawing Server Hompage, 2004. URL `http://loki.cs.brown.edu:8081/graphserver/gds/gds-home.shtml`.

[23] GraphViz. Graphviz—graph drawing tools. URL `http://graphviz.org/`. Graph Visualization Software.

[24] Grappa. Grappa Homepage, 2003. URL `http://www.research.att.com/~john/Grappa/`.

[25] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, Juni 1987.

[26] Michael Jünger und Petra Mutzel. *Graph Drawing Software*. Springer, Oktober 2003. ISBN 3540008810.

[27] Tomihisa Kamada und Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989. ISSN 0020-0190. URL `http://dx.doi.org/10.1016/0020-0190(89)90102-6`.

[28] Tobias Kloss. Flexibles und Automatisiertes Layout von Statecharts. Studienarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, Juli 2003.

[29] Florian Lüpke. Implementierung eines Statechart-Editors mit layoutbasierten Bearbeitungshilfen. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, Juni 2005. URL `http://www.informatik.uni-kiel.de/~rt-kiel/kiel/documents/papers/editor/paper.pdf`.

[30] *Stateflow and Stateflow Coder for use with Simulink — User's Guide*. Mathworks Inc., 6 edition, 2004.

[31] Networks/Pajek—Program for Large Network Analysis. Pajek Homepage, 2005. URL `http://vlado.fmf.uni-lj.si/pub/networks/pajek/`.

[32] Stephen C. North. Agraph tutorial. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, August 2002. URL `http://www.research.att.com/sw/tools/graphviz/Agraph.pdf`.

[33] Stephen C. North. Drawing graphs with NEATO. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, April 2004. URL `http://www.research.att.com/sw/tools/graphviz/neatoguide.pdf`.

[34] Object Management Group. OMG Unified Modeling Language Specification, Version 1.5, März 2003. `http://www.omg.org/cgi-bin/doc?formal/03-03-01`.

[35] André Ohlhoff. Simulating the Behavior of Synccharts. Studienarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, November 2004. URL `http://www.informatik.uni-kiel.de/~rt-kiel/kiel/documents/papers/simulator-estudio/paper.pdf`.

[36] OREAS GmbH—Optimization Research And Software. GoVisual, 2004. URL `http://www.oreas.com/libraries.php`.

[37] Steffen Prochnow und Reinhard von Hanxleden. Visualisierung komplexer reaktiver Systeme – Annotierte Bibliographie. Technischer Bericht 0406, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, Juni 2004.

[38] E. M. Reingold und J. S. Tilford. Tidier drawing of trees. *IEEE Transactions on Software Engineering*, 7:223–228, mar 1981.

[39] Lawrence Rowe, Michael Davis, Eli Messinger, Carl Meyer, Charles Spirakis und Allen Tuan. A browser for directed graphs. *Software – Practice and Experience*, 17(1):61–76, Januar 1987.

[40] Janet M. Six und Ioannis G. Tollis. Circular drawings of biconnected graphs. *Lecture Notes in Computer Science*, 1619:57–73, 1999. ISSN 0302-9743.

[41] Janet M. Six und Ioannis G. Tollis. A framework for circular drawings of networks. *Lecture Notes in Computer Science*, 1731:107ff, 2000. ISSN 0302-9743. URL `http://link.springer-ny.com/link/service/series/0558/bibs/1731/17310107.htm;http://link.springer-ny.com/link/service/series/0558/papers/1731/17310107.pdf`.

[42] Software Systems Engineering Research Group - BTU Cottbus. CrocoCosmos Homepage, 2003. URL `http://software-systemtechnik.de/crococosmos/`.

[43] Kozo Sugiyama und Kazuo Misue. A Simple and Unified Method for Drawing Graphs: Magnetic-Spring Algorithm. In *GD '94: Proceedings of the DIMACS International Workshop on Graph Drawing*, Seiten 364–375, London, UK, 1995. Springer-Verlag. ISBN 3-540-58950-3.

[44] Kozo Sugiyama, Shojiro Tagawa und Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, Februar 1981.

[45] Sun Microsystems, Inc. Java Native Interface, 2005. URL `http://java.sun.com/docs/books/tutorial/native1.1/`.

[46] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987. ISSN 0097-5397.

[47] Roberto Tamassia, Giuseppe Di Battista und Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, 18(1): 61–79, 1988. ISSN 0018-9472. URL `http://dx.doi.org/10.1109/21.87055`.

[48] Jan Täubrich. Untersuchung der nicht-interaktiven Simulation von Stateflow-Statecharts. Praktikumsbericht, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, April 2005. URL http://www.informatik.uni-kiel.de/~rt-kiel/kiel/documents/papers/simulator-matlab-evaluation/paper.pdf.

[49] The KIEL Project. Project API Documentation, . URL http://www.informatik.uni-kiel.de/~rt-kiel/kiel/kiel/doc/. Kiel Integrated Environment for Layout.

[50] The KIEL Project. Project Homepage, . URL http://www.informatik.uni-kiel.de/~rt-kiel/. Kiel Integrated Environment for Layout.

[51] The Visone Project. Visone Homepage, 2004. URL http://www.visone.de/.

[52] Tim Dwyer and Peter Eckersley. WilmaScpoe Homepage, 2003. URL http://www.wilmascope.org/.

[53] Università degli Studi Roma Tre. DBdraw Homepage, 2000. URL http://www.dia.uniroma3.it/~dbdraw/.

[54] Università degli Studi Roma Tre. Polyphemus Homepage, 2001. URL http://www.dia.uniroma3.it/~polyph/.

[55] Università degli Studi Roma Tre. Hermes Homepage, 2003. URL http://www.dia.uniroma3.it/~hermes/.

[56] University Passau. BioPath Homepage, 2001. URL http://biopath.fmi.uni-passau.de/.

[57] ViSta—Visulaizing Statecharts. ViSta Homepage, 2003. URL http://www.utdallas.edu/~rmili/lab/vistaProjectOrganization.htm.

[58] John Q. Walker, II. A node-positioning algorithm for general trees. *Software – Practice and Experience*, 20(7):685–705, 1990. ISSN 0038-0644.

[59] John Warfield. Crossing Theory and Hierarchy Mapping. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-7(7):502–523, 1977.

[60] Thomas Willhalm. Software packages. In Michael Kaufmann und Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*, Kapitel A, Seiten 274–281. Springer-Verlag, Berlin, Germany, 2001. ISBN 3-540-42062-2.

[61] Graham J. Wills. NicheWorks—Interactive Visualization of Very Large Graphs. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, Seiten 403–414, London, UK, 1997. Springer-Verlag. ISBN 3-540-63938-1.

[62] Wine. Wine Homepage, 2005. URL http://www.winehq.com/. Wine Is Not an Emulator.

[63] Mirko Wischer. Ein FileInterface für das KIEL Projekt. Praktikumsbericht, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, 2005. URL http://www.informatik.uni-kiel.de/~rt-kiel/kiel/documents/papers/fileInterface/paper.pdf.

[64] yWorks. yFiles Homepage, 2005. URL http://www.yworks.com/ger/products_yfiles_about.htm.

A. Einstellungen

In diesem Kapitel werden die verschiedenen Einstellungen des Moduls `Layouter` detailliert beschrieben und deren Auswirkungen mit Beispielen verdeutlicht.

layouter.defaultLayouter

Dieser Parameter legt die Standard-Layouter-Klasse fest. Dieser Layouter berechnet zum Beispiel die grafischen Informationen eingelesener *Esterel*-Dateien, die keine grafischen Informationen enthalten.

Standardwert: `DotBinLayouter`

mögliche Werte: Alle Layouter-Klassennamen

layouter.logLevel

Jede Layouter-Klasse hat die Möglichkeit Informationen in eine Protokolldatei zu schreiben. Dabei wird jede Information mit einem *Log-Level* versehen, das die Relevanz dieser Information beschreibt. Mit diesem Parameter lässt sich die Ausgabe der Informationen steuern. Dabei gilt je kleiner die Zahl, desto mehr Informationen werden in die Protokolldatei geschrieben.

Standardwert: 3 (Fehler und Prozessinformationen)

mögliche Werte: ganze Zahlen von 0 – 10

layouter.semanticZoom

Die Layouter-Klasse `LayerwiseLayouter` 4.3.1 unterstützt das Merkmal des *Semantischen Zoom*. Mit diesem Parameter kann dieses Merkmal ein- bzw. abgeschaltet werden.

Standardwert: `true`

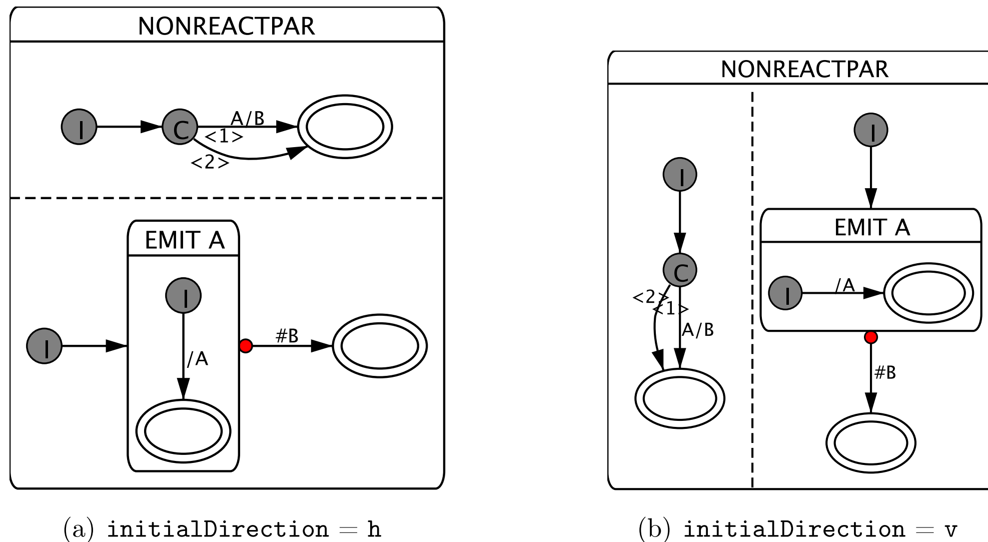
mögliche Werte: `true` und `false`

layouter.layerbased.initialDirection

Mit diesem Parameter lässt sich festlegen, in welcher Layout-Richtung die oberste Hierarchieebene des *Statecharts* ausgelegt werden soll. Mit der Einstellung `auto` werden zunächst sowohl die horizontale als auch die vertikale Richtung berechnet und die kompaktere Variante gewählt. Abbildung A.1 zeigt ein *Statechart* in den beiden möglichen Ausrichtungen.

Standardwert: `auto`

mögliche Werte: `h`, `v`, `auto`



(a) `initialDirection` = `h` (b) `initialDirection` = `v`

Abbildung A.1.: Veränderung der initialen Ausrichtung

layouter.layerbased.alternateDirection

Die Layout-Richtung der einzelnen Hierarchieebenen kann mit diesem Parameter alternierend gewechselt werden. Falls die Layout-Richtungen alternieren sollen und die erste Ebene in horizontaler Richtung ausgelegt wird, wird die zweite in vertikaler und die dritte wieder in horizontaler Richtung ausgelegt (u. s. w.). Die *Statechart*-Diagramme der Abbildung A.1 weisen alternierende Layout-Richtungen auf. In Abbildung A.2 wird das gleiche *Statechart* noch einmal mit gleichbleibender horizontaler Ausrichtung gezeigt.

Standardwert: `true`

mögliche Werte: `true` und `false`

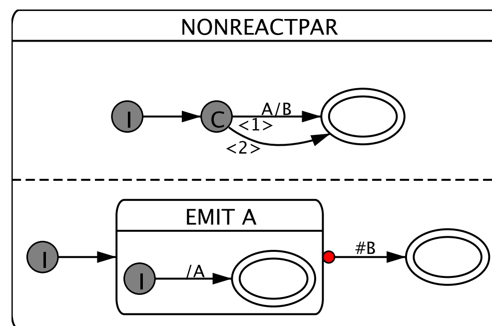


Abbildung A.2.: *Statechart*-Diagramm in konstanter Layout-Richtung

layouter.layerbased.layerAlignment.horizontal

Mit diesem Parameter lässt sich die horizontale Ausrichtung des „Inhalts“ eines Oberzustands (`CompositeState`) beeinflussen.

Standardwert: `c`

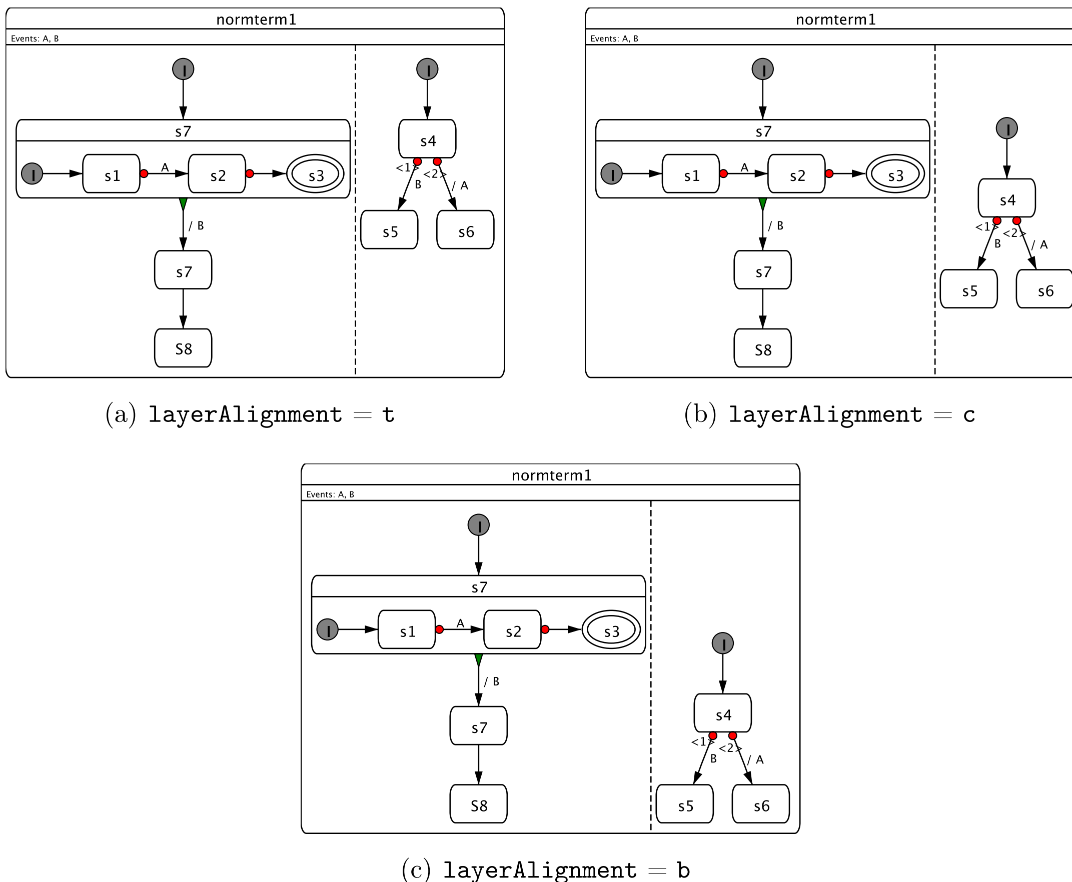
mögliche Werte: `[l]eft`, `[c]enter`, `[r]ight`

layouter.layerbased.layerAlignment.vertical

Dieser Parameter ist das vertikal Pendant zum eben beschriebenen Parameter. Abbildung A.3 zeigt drei *Statechart*-Diagramme mit unterschiedlicher vertikaler Ausrichtung.

Standardwert: `c`

mögliche Werte: `[t]op`, `[c]enter` und `[b]ottom`



(a) `layerAlignment` = `t`

(b) `layerAlignment` = `c`

(c) `layerAlignment` = `b`

Abbildung A.3.: Veränderung der vertikalen Ausrichtung

layouter.graphviz.mode

Für die Anbindung des Projekts *GraphViz* stehen der Zugriff über ein separat laufendes Programm (`bin`) und der Aufruf der Bibliotheksfunktionen (`lib`) bereit. Die Auswirkungen einer Veränderung dieses Parameters werden erst mit einem Neustart der *KIEL*-Applikation wirksam.

Standardwert: `bin`

mögliche Werte: `bin`, `lib` und `both`

layouter.graphviz.winCommand.dot

Das separat laufenden Programm *Dot* wird auf diese Weise unter dem Betriebssystem *Micorsoft Windows* gestartet.

Standardwert: `dot.exe -Tdot`

mögliche Werte: jeder gültige Kommandozeilenbefehl

layouter.graphviz.winCommand.neato

Das separat laufenden Programm *Neato* wird auf diese Weise unter dem Betriebssystem *Micorsoft Windows* gestartet.

Standardwert: `neato.exe -Tdot`

mögliche Werte: jeder gültige Kommandozeilenbefehl

layouter.graphviz.winCommand.circo

Das separat laufenden Programm *Circo* wird auf diese Weise unter dem Betriebssystem *Micorsoft Windows* gestartet.

Standardwert: `circo.exe -Tdot`

mögliche Werte: jeder gültige Kommandozeilenbefehl

layouter.graphviz.winCommand.twopi

Das separat laufenden Programm *Twopi* wird auf diese Weise unter dem Betriebssystem *Micorsoft Windows* gestartet.

Standardwert: `twopi.exe -Tdot`

mögliche Werte: jeder gültige Kommandozeilenbefehl

layouter.graphviz.linuxCommand.dot

Das separat laufenden Programm *Dot* wird auf diese Weise unter dem Betriebssystem *Linux* gestartet.

Standardwert: `dot -Tdot`

mögliche Werte: jeder gültige Kommandozeilenbefehl

layouter.graphviz.linuxCommand.neato

Das separat laufenden Programm *Neato* wird auf diese Weise unter dem Betriebssystem *Linux* gestartet.

Standardwert: `neato -Tdot`

mögliche Werte: jeder gültige Kommandozeilenbefehl

layouter.graphviz.linuxCommand.circo

Das separat laufenden Programm *Circo* wird auf diese Weise unter dem Betriebssystem *Linux* gestartet.

Standardwert: `circo -Tdot`

mögliche Werte: jeder gültige Kommandozeilenbefehl

layouter.graphviz.linuxCommand.twopi

Das separat laufenden Programm *Twopi* wird auf diese Weise unter dem Betriebssystem *Linux* gestartet.

Standardwert: `twopi -Tdot`

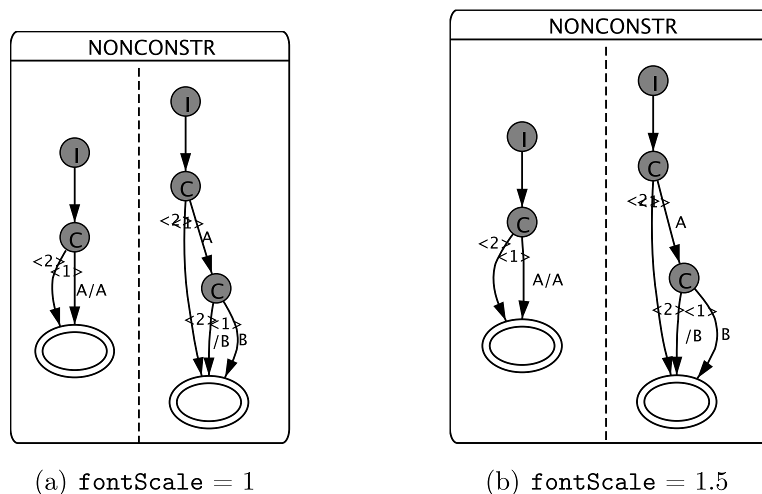
mögliche Werte: jeder gültige Kommandozeilenbefehl

layouter.graphviz.fontsizeScale

Mit diesem Skalierungsfaktor wird die Schriftgröße eines Transitions-*Labels* innerhalb der *GraphViz*-Berechnung versehen. Wenn der Wert größer 1 gewählt wird, lässt sich auf diese Weise mehr Platz für ein *Label* innerhalb der *KIEL*-Applikation schaffen. Siehe hierzu Abbildung A.4.

Standardwert: 1.1

mögliche Werte: ein beliebiger Gleitkommawert



(a) `fontScale` = 1 (b) `fontScale` = 1.5

Abbildung A.4.: Veränderung der Schriftgröße

layouter.graphviz.initialTransitionWeightGain

Initiale Transitionen können mit einem höheren Gewicht versehen werden, damit diese Transition möglichst kurz und geradlinig ausfällt. Somit werden der initiale Zustand und sein Folgezustand dicht bei einander platziert.

Standardwert: 1

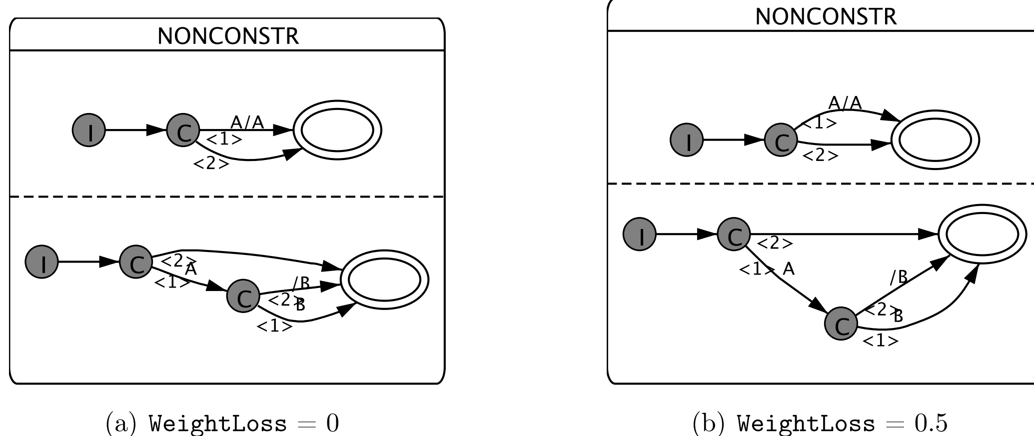
mögliche Werte: ein beliebiger Gleitkommawert

layouter.graphviz.labeledTransitionWeightLoss

Dieser Parameter ist das Pendant zu der zuvor genannten Einstellung. Hiermit lässt sich das Gewicht von Transitionen mit *Label* verringern, so dass eine solche Transition weniger „straff“ ausfällt als eine Transition ohne *Label*. Im Allgemeinen verschlechtert sich jedoch das Resultat derart, dass eine Verringerung des Gewichtes nicht empfehlenswert ist. Abbildung A.5 zeigt die Auswirkungen dieses Parameters.

Standardwert: 0

mögliche Werte: ein beliebiger Gleitkommawert



(a) `WeightLoss` = 0 (b) `WeightLoss` = 0.5

Abbildung A.5.: Auswirkungen der Gewichtsreduktion

layouter.graphviz.rankdir.vertical

Das Projekt *GraphViz* bietet mehrere mögliche „Laufrichtungen“ eines Diagramms an. In vertikaler Richtung ist eine Ausrichtung sowohl von oben nach unten (`TB`) als auch umgekehrt (`BT`) möglich. Der Möglichkeiten dieses Parameters sind in Abbildung A.6 dargestellt.

Standardwert: `TB`

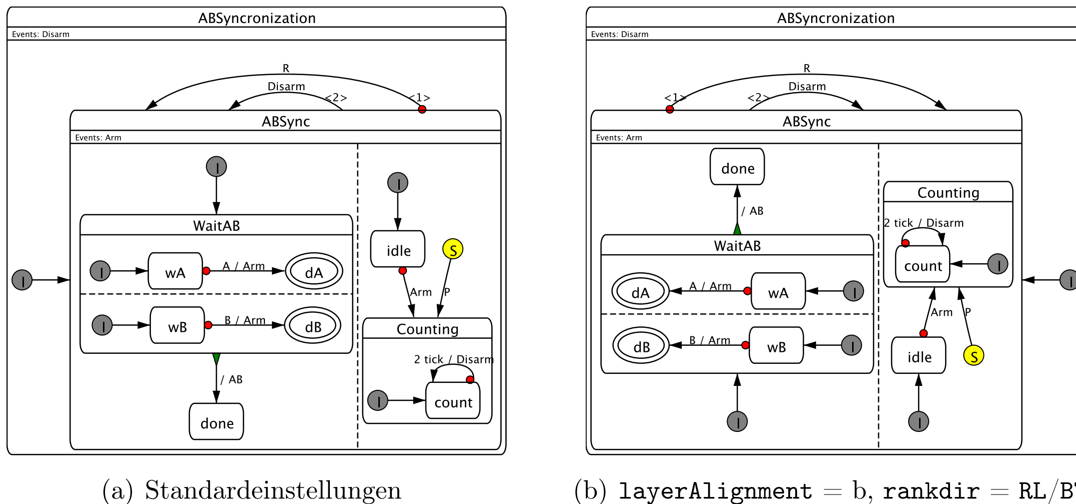
mögliche Werte: `TB` und `BT`

layouter.graphviz.rankdir.horizontal

Dieser Parameter ist das horizontale Pendant zum vorher beschriebenen. Hier stehen die Ausrichtungen von links nach rechts (`LR`) als auch von rechts nach links (`RL`) zur Verfügung.

Standardwert: `LR`

mögliche Werte: `LR` und `RL`



(a) Standardeinstellungen

(b) `layerAlignment` = b, `rankdir` = `RL`/`BT`

Abbildung A.6.: Ausrichtung und Layout-Verlauf

layouter.graphviz.emptyLabelReplacement.vertical

Um den Abstand zwischen parallel verlaufenden Transitionen zu vergrößern, kann innerhalb der *GraphViz*-Berechnung ein *Label* hinzugefügt werden. Dieses *Label* ist innerhalb der *KIEL*-Darstellung nicht sichtbar.

Standardwert: `M`

mögliche Werte: eine beliebige Zeichenfolge

layouter.graphviz.emptyLabelReplacement.horizontal

Dies ist das horizontale Pendant zum vorher beschriebenen Parameter. Jedoch ist der Einsatz für diese Ausrichtung nicht so gut geeignet, wie für die vertikale Ausrichtung, da die Transitionen in diesem Fall länger werden. Die Auswirkungen der *Label*-Ersetzung sind in Abbildung A.7 exemplarisch dargestellt.

Standardwert: <*leer*>

mögliche Werte: eine beliebige Zeichenfolge

layouter.graphviz.priorityDistance

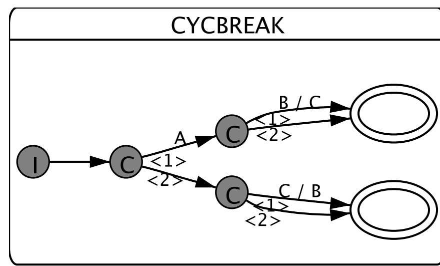
Mit diesem Parameter lässt sich der Abstand der Priorität vom Startpunkt der Transition bestimmen. Je größer der Wert ist, desto weiter entfernt liegt er von diesem Punkt.

Standardwert: 1.5

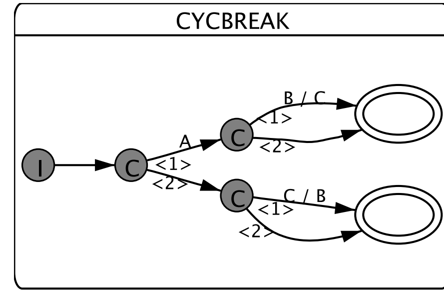
mögliche Werte: ein beliebiger Gleitkommawert

layouter.graphviz.priorityAngle.lr

Dieser Parameter legt den Drehwinkel für die Prioritätsposition in der Links-Rechts-Layout-Richtung fest.



(a) keine Ersetzung



(b) `labelReplacement` = `M`

Abbildung A.7.: Hinzufügen virtueller Transitions-*Label*

Standardwert: −20

mögliche Werte: ein beliebiger Gleitkommawert

layouter.graphviz.priorityAngle.rl

Dieser Parameter legt den Drehwinkel für die Prioritätsposition in der Rechts-Links-Layout-Richtung fest.

Standardwert: −20

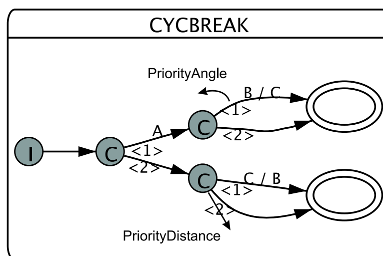
mögliche Werte: ein beliebiger Gleitkommawert

layouter.graphviz.priorityAngle.tb

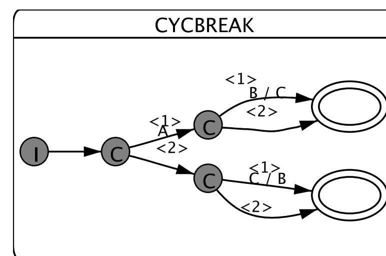
Dieser Parameter legt den Drehwinkel für die Prioritätsposition in der Oben-Unten-Layout-Richtung fest.

Standardwert: −30

mögliche Werte: ein beliebiger Gleitkommawert



(a) Standard



(b) `priorityDistance` = 2, `labelAngle` = 20

Abbildung A.8.: Veränderung Prioritätseinstellungen

layouter.graphviz.priorityAngle.bt

Dieser Parameter legt den Drehwinkel für die Prioritätsposition in der Unten-Oben-Layout-Richtung fest. Die Auswirkungen der auf die Prioritäten bezogenen Parameter sind in Abbildung A.8 dargestellt.

Standardwert: −30

mögliche Werte: ein beliebiger Gleitkommawert

layouter.graphviz.dpi

GraphViz arbeitet mit auf Zoll (*Inch*) bezogenen Höhen- und Breitenangaben. Die *KIEL*-Applikation verwendet hierfür Punktangaben (*Pixel*). Für die Umrechnung dieser Angaben wird eine Angabe der Punkte pro Zoll (*dpi*) benötigt.

Standardwert: 72

mögliche Werte: eine beliebige Ganzzahl

layouter.graphviz.stateShape

Damit eine möglichst gute Übereinstimmung zwischen der *GraphViz*- und er *KIEL*-Darstellung gewährleistet werden kann, müssen sich die grundlegenden Formen der Zustände ähneln. Dieser Parameter gibt die Form eines „normalen“ Zustands an.

Standardwert: `box`

mögliche Werte: eine beliebige *GraphViz*-Form

layouter.graphviz.finalStateShape

Damit eine möglichst gute Übereinstimmung zwischen der *GraphViz*- und er *KIEL*-Darstellung gewährleistet werden kann, müssen sich die grundlegenden Formen der Zustände ähneln. Dieser Parameter gibt die Form eines finalen Zustands an.

Standardwert: `ellipse`

mögliche Werte: eine beliebige *GraphViz*-Form

layouter.graphviz.pseudoStateShape

Damit eine möglichst gute Übereinstimmung zwischen der *GraphViz*- und er *KIEL*-Darstellung gewährleistet werden kann, müssen sich die grundlegenden Formen der Zustände ähneln. Dieser Parameter gibt die Form eines Pseudozustands an.

Standardwert: `ellipse`

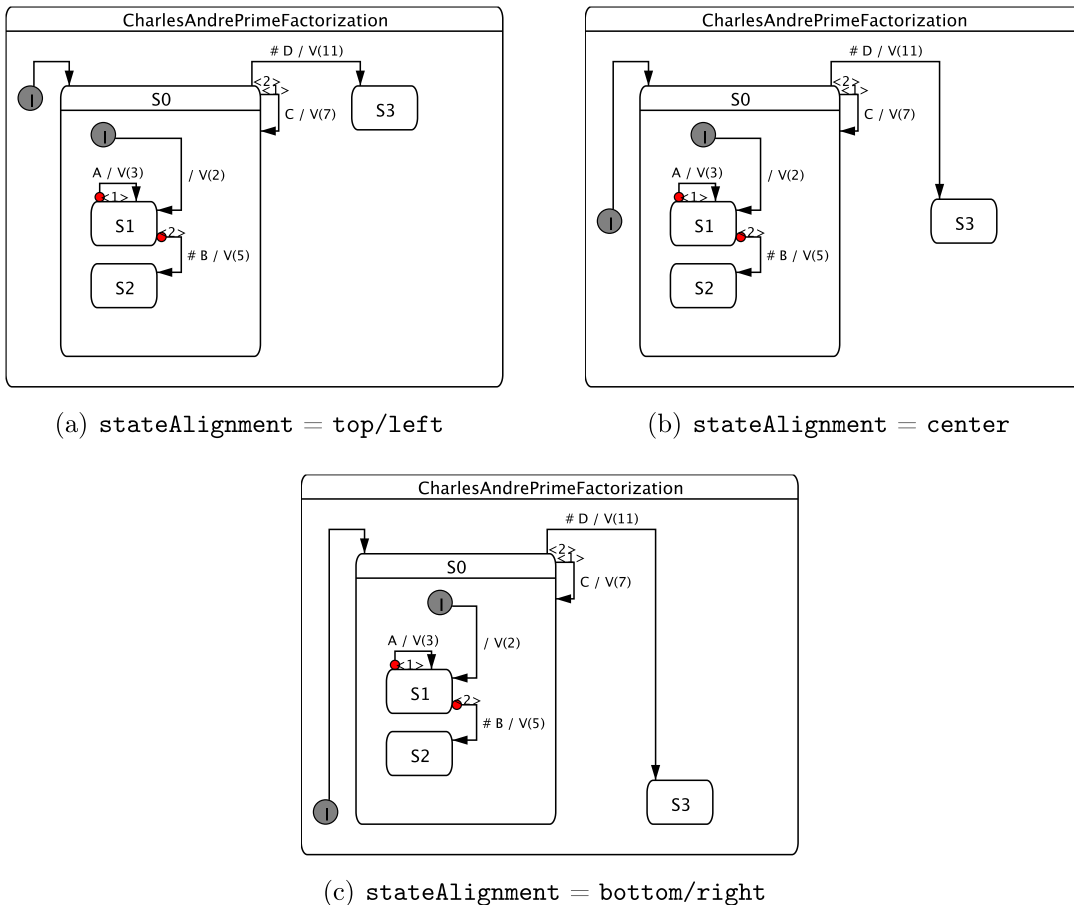
mögliche Werte: eine beliebige *GraphViz*-Form

layouter.hv.stateAlignment

Dieser Parameter beeinflusst die Ausrichtung der Platzierung der Zustände unter Verwendung des `HVLayouters`. Die Möglichkeiten dieses Parameters zeigt Abbildung A.9 an einem Beispiel.

Standardwert: `center`

mögliche Werte: `top/left`, `center` und `bottom/right`



(a) `stateAlignment` = `top/left`

(b) `stateAlignment` = `center`

(c) `stateAlignment` = `bottom/right`

Abbildung A.9.: Ausrichtungsmöglichkeiten der Zustände

layouter.hv.labelBorder

Dieser Parameter definiert den Abstand an der Ober- und Unterseite eines Transitions-*Labels*.

Standardwert: 5

mögliche Werte: eine beliebige Ganzzahl

layouter.hv.stateBorder

Dieser Parameter definiert den Abstand um einen Zustand. Dieser und weitere Parameter sind in Abbildung A.10 wiederzufinden.

Standardwert: 15

mögliche Werte: eine beliebige Ganzzahl

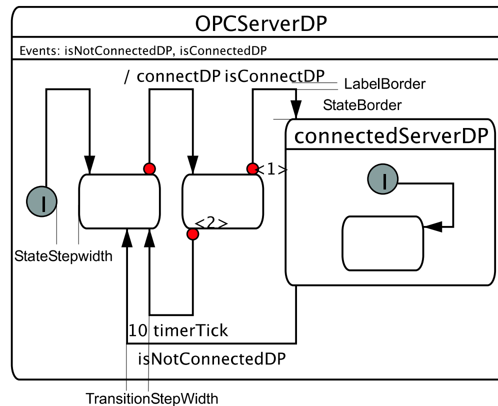


Abbildung A.10.: Einstellungen des `HVLayouter`

layouter.hv.horizontalStateStepWidth

Dieser Parameter definiert den horizontalen Abstand zwischen zwei Zuständen.

Standardwert: 30

mögliche Werte: eine beliebige Ganzzahl

layouter.hv.horizontalTransitionStepWidth

Dieser Parameter definiert den horizontalen Abstand zwischen zwei Transitionen.

Standardwert: 15

mögliche Werte: eine beliebige Ganzzahl

layouter.hv.verticalStateStepWidth

Dieser Parameter definiert den vertikalen Abstand zwischen zwei Zuständen.

Standardwert: 30

mögliche Werte: eine beliebige Ganzzahl

layouter.hv.verticalTransitionStepWidth

Dieser Parameter definiert den vertikalen Abstand zwischen zwei Transitionen.

Standardwert: 15

mögliche Werte: eine beliebige Ganzzahl

B. *Code*-Beispiel für die Benutzung der *GraphViz*-Bibliothek

B.1. Beschreibung

In Abschnitt 3.3.2 wurde der Zugriff auf das Projekt *GraphViz* über die mitgelieferte *C++*-Bibliothek beschrieben. In diesem Zusammenhang wurden einzelne *Code*-Abschnitte vorgestellt. Das gesamte Programm ist im folgenden Abschnitt aufgelistet. Ziel dieses Beispielprogramms ist die Erstellung eines Graphen mit anschließender Layout-Berechnung und Ausgabe der Größe des Graphen. Das Beispiel ist in Abbildung 3.8 grafisch dargestellt worden.

B.2. Der *Code*

```
#include <dotneato.h>

int main() {
    Agraph_t *graph;
    Agnode_t *a;
    Agnode_t *b;
    Agnode_t *c;
    Agedge_t *ab;
    Agedge_t *bc;
    Agedge_t *ca;

    // initializing library
    aginit();

    // create graph
    graph = agopen("dotsample", AGDIGRAPH);

    // setting global graph attributes
    agraphattr(graph, "rankdir", "LR");
    // setting global node attributes
    agnodeattr(graph, "label", "");
    agnodeattr(graph, "width", "0.75");
    agnodeattr(graph, "height", "0.5");
    agnodeattr(graph, "shape", "ellipse");
```

```
    // setting global edge attributes
    agedgeattr(graph, "label", "");
    agedgeattr(graph, "taillabel", "");
    agedgeattr(graph, "labelangle", "0");
    agedgeattr(graph, "labeldistance", "1");

    // adding nodes
    a = agnode(graph, "A");
    agset(a, "label", "Node A");
    agset(a, "width", "1.50");
    agset(a, "height", "1.00");
    agset(a, "shape", "box");
    b = agnode(graph, "B");
    agset(b, "label", "Node B");
    agset(b, "width", "1.20");
    agset(b, "height", "0.80");
    agset(b, "shape", "ellipse");
    c = agnode(graph, "C");
    agset(c, "label", "Node C");
    agset(c, "width", "1.40");
    agset(c, "height", "0.75");
    agset(c, "shape", "hexagon");

    // adding edges
    ab = agedge(graph, a, b);
    agset(ab, "label", "A -> B");
    bc = agedge(graph, b, c);
    agset(bc, "label", "B -> C");
    ca = agedge(graph, c, a);
    agset(ca, "label", "C -> A");
    agset(ca, "taillabel", "taillabel");
    agset(ca, "labelangle", "25");
    agset(ca, "labeldistance", "2.5");

    // layouting
    dot_layout(graph);

    // read layout informations
    attach_attrs(graph);
    printf("Bounding Box: %s\n", agget(graph, "bb"));

    // cleanup and close graph
    dot_cleanup(graph);
    close(graph);

    return 0;
}
```

C. Der Quell-*Code* des Moduls `Layouter`

In dem vorleigenden Anhangskapitel wird der Quelle-*Code* aufgelistet, der im Rahmen dieser Arbeit erstellt bzw. angepasst wurde. Vor den Auflistungen in den Abschnitten C.2 und C.3 wird eine kurze Beschreibung der Programmteile gegeben. Eine detaillierte Beschreibung ist in Kapitel 4 zu finden.

C.1. Beschreibung

Allgemeine Layouter-Klassen

Zu den allgemeinen Layouter-Klassen, die im Paket `kiel.layouter` vereint sind, zählen:

- `Handler`,
- `LayerwiseLayouter`,
- `Layouter`,
- `LayouterProperties` und
- `PseudoLayouter`.

Dieses Klassen stellen sowohl die Basis für Neuentwicklungen, als auch die Schnittstelle für die anderen Module des Projekts *KIEL* bereit.

Layouter-Klassen für die Anbindung des Projekts *GraphViz*

Das Paket `kiel.layouter.graphviz` vereint die Klassen zur Anbindung des Projekts *GraphViz*. In diesem Paket befinden sich die Klassen:

- `GraphvizLayouter`,
- `GraphvizBinaryLayouter`,
- `CircoBinLayouter`,
- `DotBinLayouter`,
- `NeatoBinLayouter`,
- `TwopiBinLayouter`,
- `GraphvizLibraryLayouter`,

- GraphvizAPI,
- CircoLibLayouter,
- DotLibLayouter,
- NeatoLibLayouter und
- TwopiLibLayouter.

Die Klassen des HVLayouters

Der *Code* des Layouter-Klasse `HVLayouter` wird aus Gründen der Vollständigkeit an dieser Stelle mit aufgelistet, obwohl das Verfahren nicht im Rahmen dieser Arbeit entwickelt, sondern nur angepasst wurde. Folgende Klassen umfasst das Paket `kiel.layouter.hvlayouter`:

- HVLayouter,
- HVLayoutedLayer,
- HorizontalLayoutedLayer,
- VerticalLayoutedLayer,
- ConversionLookupTable,
- HVLayoutedEdge,
- HVLayoutedNode,
- Level,
- StateLevel,
- TransitionLevel,
- PropertyComparator,
- InitialStateComparator,
- NumberOfLevelsComparator,
- TransitionCrossingComparator,
- WellBalancedLevelsComparator und
- PermutationIterator.

Der *C++-Code* der Anbindung über das *JNI*

In Abschnitt C.3 sind die beiden *C++*-Dateien aufgelistet, die die Umsetzung der *Java*-Methoden in die entsprechenden *C++*-Bibliotheksfunktionen übernehmen. Um bei der Auflistung des *Codes* Platz zu sparen, sind die folgenden Abschnitte zweispaltig und im Querformat gedruckt worden.

C.2. Der *Java-Code*

C.2.1. kiel.layouter.Handler

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/25 09:08:18 $
 */
package kiel.layouter;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;

import kiel.layouter.graphviz.CircoBinLayouter;
import kiel.layouter.graphviz.CircoLibLayouter;
import kiel.layouter.graphviz.DotBinLayouter;
import kiel.layouter.graphviz.DotLibLayouter;
import kiel.layouter.graphviz.NeatoBinLayouter;
import kiel.layouter.graphviz.NeatoLibLayouter;
import kiel.layouter.graphviz.TwopiBinLayouter;
import kiel.layouter.graphviz.TwopiLibLayouter;
import kiel.layouter.hvlayouter.HVLayouter;
import kiel.util.LogFile;


/**
 * <p>
 * Description: This class loads all layouters. The designated layouter can
 *     be
 * accesed by its name.
 * </p>
 *
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 *
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss</a>
 * @version $Revision: 1.23 $
 */
public final class Handler {
    /**
     * This is the instance of <code>Handler</code>.
     */
    private static Handler instance = null;

    /**
     * The <code>Handler</code> class uses the disign pattern singleton.
     This
     * method will return the instance.
     *
     * @return instance of <code>Handler</code>
     */
    public static Handler instance() {
        if (instance == null) {
            instance = new Handler();
            if (!LayouterProperties.load()) {
                instance.logError("Error while loading properties.");
            }
            instance.resetLogLevel();
            instance.loadLayouters();
            instance.resetDefaultLayouter();
        }

        return instance;
    }

    /**
     * This is the default layouter.
     */
    private Layouter defaultLayouter;

    /**
     * This is the log file.
     */
    private LogFile logFile;

    /**
     * Maps name -> layouter.
     */
    private HashMap nameLayouterMap;

    /**
     * This is the pseudo layouter.
     */
    private PseudoLayouter pseudoLayouter;

    /**
     * This is the constructor.
     */
    private Handler() {
        IOException ioe;

        // create map
        nameLayouterMap = new HashMap();

        // open log file
```

```
            ioe = null;
            try {
                logFile = new LogFile(new FileWriter(
                        System.getProperty("user.home") + File.separator + ".
    kiel"
                        + File.separator + "layouter.log"), "Layouter");
            } catch (IOException e) {
                logFile = new LogFile(new PrintWriter(System.out), "Layouter");
110             ioe = e;
            }

            logFile.enableLog();
            logFile.setLogLevel(LogFile.ALL);
            if (ioe != null) {
                logError("Could not open log file at "
                    + System.getProperty("user.home") + File.separator
                    + ".kiel" + File.separator + "layouter.log (" + ioe + ")");
            }
120     }

        /**
         * @see Object#finalize()
         */
        protected void finalize() throws Throwable {
            logFile.closeLog();
            super.finalize();
        }

130     /**
         * Returns all layouters as <code>Layouter</code> object.
         *
         * @return array of all layouters
         * @deprecated use {@link #getLayouterNames()}
         *             and {@link #getLayouterNamed(String)} instead
         */
        public Layouter[] getAllLayouters() {
            return (Layouter[]) nameLayouterMap.values().toArray(new Layouter
     [0]);
        }
140
        /**
         * This method returns the default layouter.
         *
         * @return the default layouter
         */
        public Layouter getDefaultLayouter() {
            return defaultLayouter;
        }

150     /**
         * This method returns the layouter object referenced by its name.
         *
         * @param name name of the layouter
         *
         * @return <code>Layouter</code> with name <code>name</code>
         */
        public Layouter getLayouterNamed(final String name) {
            return (Layouter) nameLayouterMap.get(name);
        }
160
        /**
         * This method returns the names of all available layouters.
         *
         * @return collection of layouter names
         */
        public Collection getLayouterNames() {
            ArrayList names = new ArrayList();
            Iterator iter;

170         iter = nameLayouterMap.keySet().iterator();

            while (iter.hasNext()) {
                names.add(iter.next());
            }

            Collections.sort(names);

            return names;
        }
180
        /**
         * This method returns the pseudo layouter.
         *
         * @return the pseudo layouter
         */
        public PseudoLayouter getPseudoLayouter() {
            return pseudoLayouter;
        }

190     /**
         * Loads the layouters specified in <code>LAYOUTERS</code>.
         */
        private void loadLayouters() {
            pseudoLayouter = new PseudoLayouter();

            nameLayouterMap.put("OriginalLayout", pseudoLayouter);
            nameLayouterMap.put("HVLayouter", new HVLayouter());
            if (LayouterProperties.getMode().equalsIgnoreCase("bin")) {
                nameLayouterMap.put("DotLayouter", new DotBinLayouter());
200             nameLayouterMap.put("CircoLayouter", new CircoBinLayouter());
                nameLayouterMap.put("NeatoLayouter", new NeatoBinLayouter());
                nameLayouterMap.put("TwopiLayouter", new TwopiBinLayouter());
            } else if (LayouterProperties.getMode().equalsIgnoreCase("lib")) {
                nameLayouterMap.put("DotLayouter", new DotLibLayouter());
                nameLayouterMap.put("CircoLayouter", new CircoLibLayouter());
                nameLayouterMap.put("NeatoLayouter", new NeatoLibLayouter());
                nameLayouterMap.put("TwopiLayouter", new TwopiLibLayouter());
            } else {
                nameLayouterMap.put("DotBinLayouter", new DotBinLayouter());
210             nameLayouterMap.put("CircoBinLayouter", new CircoBinLayouter())
     ;
                nameLayouterMap.put("NeatoBinLayouter", new NeatoBinLayouter())
     ;
                nameLayouterMap.put("TwopiBinLayouter", new TwopiBinLayouter())
     ;
                nameLayouterMap.put("DotLibLayouter", new DotLibLayouter());
                nameLayouterMap.put("CircoLibLayouter", new CircoLibLayouter())
     ;
                nameLayouterMap.put("NeatoLibLayouter", new NeatoLibLayouter())
     ;
```

```
                nameLayouterMap.put("TwopiLibLayouter", new TwopiLibLayouter())
        ;
            }
        }

220     /**
         * Writes a debug message to the log file.
         *
         * @param msg a message
         */
        public void logDebug(final String msg) {
            logFile.log(LogFile.DEBUG, msg);
        }

        /**
230      * Writes an error message to the log file.
         *
         * @param msg a message
         */
        public void logError(final String msg) {
            logFile.log(LogFile.ERROR, msg);
        }

        /**
         * Writes an information message to the log file.
240      *
         * @param msg a message
         */
        public void logInfo(final String msg) {
            logFile.log(LogFile.INFO, msg);
        }

        /**
         * Sets the default layouter as specified in
         * <code>LayouterProperties</code>.
250      */
        protected void resetDefaultLayouter() {
            Iterator iter = nameLayouterMap.values().iterator();
            defaultLayouter = null;

            while (iter.hasNext()) {
                Layouter layouter = (Layouter) iter.next();
                if (layouter.getClass().getName()
                        .endsWith(LayouterProperties.getDefaultLayouter())) {
                    defaultLayouter = layouter;
260                 break;
                }
            }

            if (defaultLayouter == null) {
                defaultLayouter = getLayouterNamed(
                        (String) getLayouterNames().iterator().next());
                logInfo("Default layouter \""
                        + LayouterProperties.getDefaultLayouter()
                        + "\" could not be found. Setting \""
270                     + defaultLayouter.getClass().getName()
                        + "\" as default layouter.");
            }
        }

        /**
         * Sets the log level as specified in <code>LayouterProperties</code>.
         */
        protected void resetLogLevel() {
            logFile.setLogLevel(LayouterProperties.getLogLevel());
280     }
    }
```

C.2.2. kiel.layouter.LayerwiseLayouter

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/05/25 09:34:32 $
 */
package kiel.layouter;

import java.awt.Dimension;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Vector;

import kiel.configMngr.Configuration;
import kiel.dataStructure.ANDState;
import kiel.dataStructure.CompositeState;
import kiel.dataStructure.DelimiterLine;
import kiel.dataStructure.Edge;
import kiel.dataStructure.Node;
import kiel.dataStructure.ORState;
import kiel.dataStructure.Region;
import kiel.dataStructure.StateChart;
import kiel.dataStructure.Transition;
import kiel.graphicalInformations.CompositeStateLayoutInformation;
import kiel.graphicalInformations.DelimiterLineLayoutInformation;
import kiel.graphicalInformations.EdgeLayoutInformation;
import kiel.graphicalInformations.LabelLayoutInformation;
import kiel.graphicalInformations.NodeLayoutInformation;
import kiel.graphicalInformations.Point;
import kiel.graphicalInformations.Properties;
import kiel.graphicalInformations.View;

/**
 * <p>
 * Description: This abstract layouter divides a given statechart into
 *    separate
 * layers. Derived laouters just layout a single layer using its specific
 * layout algorithm.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.20 $
 */
public abstract class LayerwiseLayouter extends Layouter {

    /**
     * Checks whether the given layer has interlevel transitions.
     *
     * @param nodes the nodes of a layer
     * @return true if the given statechart has interlevel transitions
     */
    private static boolean hasInterlayerEdges(final Collection nodes) {
        Iterator nodeIter;
        boolean overlapingEdgeDetected = false;

        nodeIter = nodes.iterator();
        while (nodeIter.hasNext()) {
            Iterator edgeIter = ((Node) nodeIter.next())
                                    .getOutgoingTransitions().iterator();

            while (edgeIter.hasNext()) {
                Edge edge = (Edge) edgeIter.next();

                if (!nodes.contains(edge.getTarget())) {
                    overlapingEdgeDetected = true;
                    break;
                }
            }
        }

        return overlapingEdgeDetected;
    }

    /**
     * Checks whether the given statechart has interlevel transitions.
     *
     * @param statechart a statechart
     * @return true if the given statechart has interlevel transitions
     */
    private static boolean hasInterlayerEdges(final StateChart statechart)
     {
        return hasInterlayerEdges(statechart.getRootNode().getSubnodes());
    }

    /**
     * If two configurations collapse the same states they have got the
     same
     * view.
     */
    private HashMap collapsedStatesToViewMap = new HashMap();

    /**
     * @see Layouter#getConfigurationView(Configuration)
     */
    public final View getConfigurationView(final Configuration config) {
        View view;

        if (LayouterProperties.useSemanticZoom()) {
            Vector statesToCollapse;
            Iterator iter;

            statesToCollapse = getStatesToCollapse(config);
            view = null;
            iter = collapsedStatesToViewMap.keySet().iterator();
            while (iter.hasNext()) {
                Vector v = (Vector) iter.next();
```

```java
                if (v.equals(statesToCollapse)) {
                    view = (View) collapsedStatesToViewMap.get(v);
                    break;
                }
            }

            if (view == null) {
                view = View.getInitializedView(getCurrentStatechart());
                view.setInvalid();

                iter = statesToCollapse.iterator();
                while (iter.hasNext()) {
                    ((CompositeStateLayoutInformation) view
                        .getLayoutInformation((CompositeState) iter.next())
)
                        .setCollapsed(true);
                }

                collapsedStatesToViewMap.put(statesToCollapse, view);
            }
        } else {
            view = getStaticView();
        }

        return view;
    }

    /**
     * A state could be collapsed if itself is inactive and all of its
     * substates are inactive too.
     *
     * @param configuration a configuration
     * @return a list of all states that could be collapsed
     */
    private Vector getStatesToCollapse(final Configuration configuration) {
        Vector statesToCollapse = new Vector();
        Vector statesToCheck = new Vector();
        Node[] activeNodes = configuration.getNodes();

        // collect all composite states
        statesToCheck.add(getCurrentStatechart().getRootNode());
        while (!statesToCheck.isEmpty()) {
            CompositeState state;
            Iterator iter;

            state = (CompositeState) statesToCheck.remove(0);
            iter = state.getSubnodes().iterator();
            while (iter.hasNext()) {
                Node node = (Node) iter.next();

                if (node instanceof CompositeState) {
                    statesToCollapse.add(node);
                    statesToCheck.add(node);
                }
            }
        }

        // remove active composite states from list
        for (int i = 0; i < activeNodes.length; i++) {
            CompositeState parent;

            parent = (CompositeState) activeNodes[i].getParent();
            while (parent != null) {
                statesToCollapse.remove(parent);
                parent = (CompositeState) parent.getParent();
            }
        }

        return statesToCollapse;
    }

    /**
     * @see Layouter#invalidateAllKnownViews()
     */
    public final void invalidateAllKnownViews() {
        Iterator iter;

        iter = collapsedStatesToViewMap.values().iterator();
        while (iter.hasNext()) {
            ((View) iter.next()).setInvalid();
        }
    }

    /**
     * @see Layouter#layoutAllKnownViews()
     */
    public final void layoutAllKnownViews() {
        Iterator iter;

        iter = collapsedStatesToViewMap.values().iterator();
        while (iter.hasNext()) {
            layoutView((View) iter.next());
        }
    }

    /**
     * @see Layouter#layoutAllViews(Configuration[])
     */
    public final void layoutAllViews(final Configuration[] configs) {
        for (int i = 0; i < configs.length; i++) {
            layoutView(getConfigurationView(configs[i]));
        }
    }

    /**
     * Layouts an <code>ANDState</code>.
     *
     * @param node
     *            AndState to be layouted
     * @param view
     *            the view
     * @param horizontalLayout
     *            layout of this layer is done horizontal
     */
    private void layoutANDState(final ANDState node,
            final View view, final boolean horizontalLayout) {
        NodeLayoutInformation nodeLayout = view.getLayoutInformation(node);
        Vector regions = new Vector(node.getSubnodes());
        Iterator iter;
```

```
        Iterator delLineIter;
        Dimension maxDim = new Dimension(0, 0);
        Point p;
        Dimension innerLayerDimension;

        // layout all regions and get maximum width/height
        iter = regions.iterator();
        while (iter.hasNext()) {
            Region region = (Region) iter.next();
            int regionWidth;
            int regionHeight;

            layoutNode(region, view, horizontalLayout);

            regionWidth = view.getLayoutInformation(region).getWidth();
            regionHeight = view.getLayoutInformation(region).getHeight();

            maxDim.width = Math.max(regionWidth, maxDim.width);
            maxDim.height = Math.max(regionHeight, maxDim.height);
        }

        // place regions
        p = Properties.getUpperLeftOfContentArea(node);
        if (horizontalLayout) {
            innerLayerDimension = new Dimension(maxDim.width, 0);
            nodeLayout.setWidth(Properties.getMinimumWidth(node,
                    innerLayerDimension.width));
        } else {
            innerLayerDimension = new Dimension(0, maxDim.height);
            nodeLayout.setHeight(Properties.getMinimumHeight(node,
                    innerLayerDimension.height));
        }

        delLineIter = node.getDelimiterLines().iterator();
        iter = regions.iterator();
        while (iter.hasNext()) {
            Region region = (Region) iter.next();
            NodeLayoutInformation regionLayout = view
                    .getLayoutInformation(region);

            regionLayout.setUpperLeftPoint(new Point(p.getX(), p.getY()));

            if (horizontalLayout) {
                p.setPosition(p.getX(),
                        p.getY() + regionLayout.getHeight());
                if (LayouterProperties.getHorizontalLayerAlignment()
                        .equalsIgnoreCase("l")) {
                    // otherwise this would be an empty statement
                    moveLayer(region.getSubnodes(), view, new Point(0, 0));
                } else if (LayouterProperties.getHorizontalLayerAlignment()
                        .equalsIgnoreCase("r")) {
                    moveLayer(region.getSubnodes(), view,
                            new Point(innerLayerDimension.width
                                    - regionLayout.getWidth(), 0));
                } else {
                    moveLayer(region.getSubnodes(), view,
                            new Point((innerLayerDimension.width
                                    - regionLayout.getWidth()) / 2, 0));
                }
                regionLayout.setWidth(innerLayerDimension.width);
                innerLayerDimension.height += regionLayout.getHeight();
            } else {
                p.setPosition(p.getX() + regionLayout.getWidth(),
                              p.getY());
                if (LayouterProperties.getVerticalLayerAlignment()
                        .equalsIgnoreCase("t")) {
                    // otherwise this would be an empty statement
                    moveLayer(region.getSubnodes(), view, new Point(0, 0));
                } else if (LayouterProperties.getVerticalLayerAlignment()
                        .equalsIgnoreCase("b")) {
                    moveLayer(region.getSubnodes(), view, new Point(0,
                            innerLayerDimension.height
                            - regionLayout.getHeight()));
                } else {
                    moveLayer(region.getSubnodes(), view, new Point(0,
                            (innerLayerDimension.height
                                    - regionLayout.getHeight()) / 2));
                }
                regionLayout.setHeight(innerLayerDimension.height);
                innerLayerDimension.width += regionLayout.getWidth();
            }

            // place delimiter line
            if (iter.hasNext()) {
                DelimiterLine delLine;
                DelimiterLineLayoutInformation delLineLayout;
                Point startPoint;
                Point endPoint;

                delLine = (DelimiterLine) delLineIter.next();
                delLineLayout = view.getLayoutInformation(delLine);

                if (horizontalLayout) {
                    startPoint = new Point(0, p.getY());
                    endPoint = p.add(new Point(
                            nodeLayout.getWidth()
                            - Properties.getRightOffset(node),
                            0));
                } else {
                    startPoint = new Point(p.getX(), p.getY());
                    endPoint = p.add(new Point(0,
                            innerLayerDimension.height
                            + Properties.getLowerOffset(node)));
                }
                delLineLayout.setStart(startPoint);
                delLineLayout.setEnd(endPoint);
            }
        }

        // set dimension
        if (horizontalLayout) {
            nodeLayout.setHeight(Properties.getMinimumHeight(node,
                    innerLayerDimension.height));
        } else {
            nodeLayout.setWidth(Properties.getMinimumWidth(node,
                    innerLayerDimension.width));
        }
    }

    /**
```

```java
 * Layouts a <code>CompositeState</code>.
 *
 * @param node
 *            composite state to be layouted
 * @param view
 *            the view
 * @param horizontalLayout
 *            layout of this layer is done horizontal
 */
private void layoutCompositeState(final CompositeState node,
        final View view, final boolean horizontalLayout) {
    NodeLayoutInformation nodeLayout = view.getLayoutInformation(node);

    if ((node instanceof ORState) || (node instanceof Region)) {
        Dimension innerLayerDimension;
        Iterator nodeIter;
        String layerAlignment;

        // layout inner nodes
        nodeIter = ((CompositeState) node).getSubnodes().iterator();
        while (nodeIter.hasNext()) {
            if (LayouterProperties.doAlternateLayoutDirection()) {
                layoutNode((Node) nodeIter.next(), view, !
horizontalLayout);
            } else {
                layoutNode((Node) nodeIter.next(), view,
horizontalLayout);
            }
        }

        // layout inner layer
        innerLayerDimension = layoutLayer(node.getSubnodes(), view,
                horizontalLayout);

        // set dimension
        nodeLayout.setHeight(Properties.getMinimumHeight(
                node, innerLayerDimension.height));
        nodeLayout.setWidth(Properties.getMinimumWidth(
                node, innerLayerDimension.width));

        // place inner layer
        layerAlignment = LayouterProperties.getHorizontalLayerAlignment
();
        if (layerAlignment.equalsIgnoreCase("l")) {
            moveLayer(node.getSubnodes(), view,
                    Properties.getUpperLeftOfContentArea(node));
        } else if (layerAlignment.equalsIgnoreCase("r")) {
            moveLayer(node.getSubnodes(), view,
                    Properties.getUpperLeftOfContentArea(node)
                            .add(new Point(nodeLayout.getWidth()
                                    - Properties.getLeftOffset(node)
                                    - Properties.getRightOffset(node)
                                    - innerLayerDimension.width, 0)));
        } else {
            moveLayer(node.getSubnodes(), view,
                    Properties.getUpperLeftOfContentArea(node)
                            .add(new Point((nodeLayout.getWidth()
                                    - Properties.getLeftOffset(node)
                                    - Properties.getRightOffset(node)
                                    - innerLayerDimension.width) / 2,
 0)));
        }

    } else if (node instanceof ANDState) {
        layoutANDState((ANDState) node, view, horizontalLayout);
    }
}

/**
 * Does the layout for a single layer.
 *
 * @param subnodes
 *            nodes in the layer
 * @param view
 *            the view
 * @param horizontalLayout
 *            is layout direction horizontal
 * @return bounding box of the layer
 */
protected abstract Dimension layoutLayer(final Collection subnodes,
        final View view, final boolean horizontalLayout);

/**
 * Does the layout for a single node. For non-collapsed composite
 states
 * the layout of the inner layer is invoked.
 *
 * @param node
 *            node to be layouted
 * @param view
 *            the view
 * @param horizontalLayout
 *            layout of this layer is done horizontal
 */
private void layoutNode(final Node node, final View view,
        final boolean horizontalLayout) {
    NodeLayoutInformation nodeLayout = view.getLayoutInformation(node);

    if (node instanceof CompositeState) {
        if (((CompositeStateLayoutInformation) nodeLayout).isCollapsed
()) {
            // do not layout inner layer
            // set dimension
            nodeLayout.setHeight(Properties.getMinimumHeight(node));
            nodeLayout.setWidth(Properties.getMinimumWidth(node));
        } else {
            // layout inner layer first
            layoutCompositeState((CompositeState) node, view,
                    horizontalLayout);
        }
    } else {
        // set dimension
        nodeLayout.setHeight(Properties.getMinimumHeight(node));
        nodeLayout.setWidth(Properties.getMinimumWidth(node));
    }
}

/**
 * @see Layouter#layoutSingleView(View)
```

```
         */
        protected final void layoutSingleView(final View view) {
            if (LayouterProperties.getInitialLayoutDirection()
                    .equalsIgnoreCase("h")) {
                // horizontal layout direction
                layoutNode(getCurrentStatechart().getRootNode(), view, true);
            } else if (LayouterProperties.getInitialLayoutDirection()
                    .equalsIgnoreCase("v")) {
470             // vertical layout direction
                layoutNode(getCurrentStatechart().getRootNode(), view, false);
            } else {
                int vertWidth;
                int vertHeight;

                // best layout direction
                layoutNode(getCurrentStatechart().getRootNode(), view, false);
                vertWidth = view.getLayoutInformation(
                        getCurrentStatechart().getRootNode()).getWidth();
480             vertHeight = view.getLayoutInformation(
                        getCurrentStatechart().getRootNode()).getHeight();
                layoutNode(getCurrentStatechart().getRootNode(), view, true);
                if (vertHeight * vertWidth
                        < view.getLayoutInformation(
                                getCurrentStatechart().getRootNode()).getWidth
         ()
                        * view.getLayoutInformation(
                                getCurrentStatechart().getRootNode()).getHeight
         ()) {
                    layoutNode(getCurrentStatechart().getRootNode(), view,
         false);
                }
490         }
        }

        /**
         * Moves all components in a layer by <code>offset</code>.
         *
         * @param nodes
         *            all nodes in the layer
         * @param view
         *            the view layout is done for
500      * @param offset
         *            all layer components are moved by this offset
         */
        private void moveLayer(final Collection nodes, final View view,
                final Point offset) {
            Iterator nodeIter;
            nodeIter = nodes.iterator();
            while (nodeIter.hasNext()) {
                Node node = (Node) nodeIter.next();
510             NodeLayoutInformation nodeLayout = view.getLayoutInformation(
         node);
                Iterator edgeIter;

                nodeLayout.move(offset);
                edgeIter = node.getOutgoingTransitions().iterator();
                while (edgeIter.hasNext()) {
                    Edge edge = (Edge) edgeIter.next();
                    EdgeLayoutInformation edgeLayout = view
                            .getLayoutInformation(edge);

520                 edgeLayout.move(offset);
                    if (edge instanceof Transition) {
                        LabelLayoutInformation labelLayout = view
                                .getLayoutInformation(((Transition) edge)
                                        .getLabel());
                        LabelLayoutInformation prioLayout = view
                                .getLayoutInformation(((Transition) edge)
                                        .getPriority());

                        labelLayout.move(offset);
530                     prioLayout.move(offset);
                    }
                }
            }
        }

        /**
         * @see Layouter#setStatechart(StateChart)
         */
        public void setStatechart(final StateChart statechart)
540         throws LayoutException {
            super.setStatechart(statechart);
            setStaticView(View.getInitializedView(getCurrentStatechart()));
            collapsedStatesToViewMap.clear();
            collapsedStatesToViewMap.put(new Vector(), getStaticView());

            if (hasInterlayerEdges(statechart)) {
                throw new LayoutException(
                    "Statecharts with interlevel transitions are not supported
         .");
            }
550     }
    }
```

C.2.3. kiel.layouter.Layouter

```java
/*
 * $Author: tkl $
 *
 * $Date: 2005/05/25 09:34:32 $
 */
package kiel.layouter;

import kiel.configMngr.Configuration;
import kiel.dataStructure.StateChart;
import kiel.graphicalInformations.View;


/**
 * <p>
 * Description: This class represents the basic layouter. You can only use
 *     a
 * layouter through this abstract layouter class.
 * </p>
 *
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 *
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss</a>
 * @version $Revision: 1.19 $
 */
public abstract class Layouter {
    /**
     * This is the statechart currently layouted.
     */
    private StateChart currentStatechart;

    /**
     * This is the static view with all subnodes visable.
     */
    private View staticView;

    /**
     * This is the constructor.
     */
    protected Layouter() {
    }

    /**
     * Call this method to get the <code>View</code> maped to the given
     * <code>Configuration</code>.
     *
     * @param config configuration to retrieve view for
     *
     * @return the view
     */
    public abstract View getConfigurationView(final Configuration config);

    /**
     * Call this method to get the current statechart.
     *
     * @return the current statechart
     */
    protected final StateChart getCurrentStatechart() {
        return currentStatechart;
    }

    /**
     * Call this method to get the static <code>View</code> of the
     * statechart.
     *
     * @return the <code>staticView</code>
     */
    public final View getStaticView() {
        return staticView;
    }

    /**
     * Invalidates all currently known views.
     */
    public abstract void invalidateAllKnownViews();

    /**
     * Call this method if you want to layout all currently known
     * views of the current statechart.
     */
    public abstract void layoutAllKnownViews();

    /**
     * Call this method if you want to layout all <code>Views</code> of the
     * current statechart with the given configurations.
     *
     * @param configs array of all configurations
     */
    public abstract void layoutAllViews(final Configuration[] configs);

    /**
     * This is the layouter speciifc method to layout a single
     * <code>View</code> of the current statechart with the given
     * configurations.
     *
     * @param view a view to be layouted
     */
    protected abstract void layoutSingleView(final View view);

    /**
     * Call this method if you want to layout a single <code>View</code> of
     * the current statechart with the given configurations.
     *
     * @param view a view to be layouted
     */
    public final void layoutView(final View view) {
        if (getCurrentStatechart() == null) {
```

```
                Handler.instance().logError("Unable to layout. No Statechart
          set.");
                return;
            }

            if (LayouterProperties.changed()) {
                LayouterProperties.reload();
                invalidateAllKnownViews();
            }
120
            if (!view.isValid()) {
                long t0, t1;

                t0 = System.currentTimeMillis();
                layoutSingleView(view);
                view.setValid();
                t1 = System.currentTimeMillis();

                Handler.instance().logInfo("time elapsed: " + (t1 - t0) + " ms
          ");
130         }
        }

        /**
         * Call this method to set the current statechart.
         *
         * @param statechart statechart to be layouted
         * @throws LayoutException thrown if an unsupported feature is
         requested
         */
        public void setStatechart(final StateChart statechart)
140         throws LayoutException {
            currentStatechart = statechart;
            staticView = null;
        }

        /**
         * Call this method to set the static view.
         *
         * @param view a view
         */
150     protected final void setStaticView(final View view) {
            staticView = view;
        }
    }
```

C.2.4. `kiel.layouter.LayouterProperties`

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/05/25 09:34:32 $
 */
package kiel.layouter;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;


/**
 * <p>
 * Description: Used to load and store user defineable properties.
 * </p>
 *
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 *
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.40 $
 */
public final class LayouterProperties {

    /**
     * This is a part of a property key.
     * Each property key is coposed of these parts.
     */
    private static final String ALTERNATEDIRECTION = "alternateDirection";

    /**
     * This is a part of a property key.
     * Each property key is coposed of these parts.
     */
    private static final String BOTTOMTOP = "bt";

    /**
     * This is a part of a property key.
     * Each property key is coposed of these parts.
     */
    private static final String CIRCO = "circo";

    /**
     * This is a part of a property key.
     * Each property key is coposed of these parts.
     */
    private static final String DEFAULTLAYOUTER = "defaultLayouter";

    /**
     * This the key for the resouce file containing the default values.
     */
    private static final String DEFAULTRESOURCE =
        "layouter.properties";

    /**
     * This is the key for property key delimiter.
     */
    private static final String DELIMITER = ".";

    /**
     * This is a part of a property key.
     * Each property key is coposed of these parts.
     */
    private static final String DOT = "dot";

    /**
     * This is a part of a property key.
     * Each property key is coposed of these parts.
     */
    private static final String DPI = "dpi";

    /**
     * This is a part of a property key.
     * Each property key is coposed of these parts.
     */
    private static final String EMPTYLABELREPLACEMENT = "
     emptyLabelReplacement";

    /**
     * This is a part of a property key.
     * Each property key is coposed of these parts.
     */
    private static final String FINALSTATESHAPE = "finalStateShape";

    /**
     * This is a part of a property key.
     * Each property key is coposed of these parts.
     */
    private static final String FONTSIZESCALE = "fontsizeScale";

    /**
     * This is a part of a property key.
     * Each property key is coposed of these parts.
     */
    private static final String GRAPHVIZ = "graphviz";

    /**
     * This is a part of a property key.
     * Each property key is coposed of these parts.
     */
    private static final String HORIZONTAL = "horizontal";

    /**
```

```
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String HORIZONTALSTATESTEPWIDTH =
    "horizontalTransitionStepWidth";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String HORIZONTALTRANSITIONSTEPWIDTH =
    "horizontalTransitionStepWidth";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String HV = "hv";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String INITIALDIRECTION = "initialDirection";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String INITIALTRANSITION_WEIGHTGAIN =
    "initialTransitionWeightGain";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String LABELBORDER = "labelBorder";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String LABELEDTRANSITION_WEIGHTLOSS =
    "labeledTransitionWeightLoss";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String LAYERALIGNMENT = "layerAlignment";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String LAYERBASED = "layerbased";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String LAYOUTER = "layouter";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String LEFTRIGHT = "lr";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String LINUXCOMMAND = "linuxCommand";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String LOGLEVEL = "logLevel";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String MODE = "mode";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String NEATO = "neato";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String PRIORITYANGLE = "priorityAngle";

/**
 * This is a part of a property key.
 * Each property key is coposed of these parts.
 */
private static final String PRIORITYDISTANCE = "priorityDistance";

/**
 * These are the internal properties.
 */
private static Properties properties;

/**
 * This is the key for user specific file.
 */
private static final String PROPERTIESFILE =
    System.getProperty("user.home")
    + File.separator
    + ".kiel"
    + File.separator
    + "layouter.properties";
```

```
        /**
         * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
        private static final String PSEUDOSTATESHAPE = "pseudoStateShape";

        /**
         * This is a part of a property key.
240      * Each property key is coposed of these parts.
         */
        private static final String RANKDIR = "rankdir";

        /**
         * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
        private static final String RIGHTLEFT = "rl";

250     /**
         * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
        private static final String SEMANTICZOOM = "semanticZoom";

        /**
         * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
260     private static final String STATEALIGNMENT = "stateAlignment";

        /**
         * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
        private static final String STATEBORDER = "stateBorder";

        /**
         * This is a part of a property key.
270      * Each property key is coposed of these parts.
         */
        private static final String STATESHAPE = "stateShape";

        /**
         * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
        private static final String TOPBOTTOM = "tb";

280     /**
         * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
        private static final String TWOPI = "twopi";

        /**
         * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
290     private static final String VERTICAL = "vertical";

        /**
         * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
        private static final String VERTICALSTATESTEPWIDTH =
            "verticalTransitionStepWidth";

        /**
300      * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
        private static final String VERTICALTRANSITIONSTEPWIDTH =
            "verticalTransitionStepWidth";

        /**
         * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
310     private static final String WINCOMMAND = "winCommand";

        /**
         * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
        private static final String SYSPROPWINLAYOUTERPATH =
            "layouter.path.windows";

        /**
320      * This is a part of a property key.
         * Each property key is coposed of these parts.
         */
        private static final String SYSPROPOTHERLAYOUTERPATH =
            "layouter.path.other";

        /**
         * Copies properties to user specific file.
         *
         * @return true, if copy was successful
330      */
        private static boolean copyDefaults() {
            InputStream is;
            FileWriter fw;
            boolean success = true;

            try {
                is = LayouterProperties.class.getResourceAsStream(
         DEFAULTRESOURCE);
                fw = new FileWriter(PROPERTIESFILE);

340             int c = is.read();
                while (c >= 0) {
                    fw.write(c);
                    c = is.read();
                }
                fw.flush();

                is.close();
                fw.close();
            } catch (IOException storeExeception) {
350             success = false;
```

```java
    }

    return success;
}

/**
 * Should layout direction alternate form layer to layer.
 *
 * @return true, if layout direction should alternate
 */
public static boolean doAlternateLayoutDirection() {
    return properties.getProperty(LAYOUTER + DELIMITER + LAYERBASED
            + DELIMITER + ALTERNATEDIRECTION).equalsIgnoreCase("true");
}

/**
 * Returns the system specific command line to start the circo process.
 *
 * @return the circo command
 */
public static String getCircoCommand() {
    String key;
    String path;

    key = LAYOUTER + DELIMITER + GRAPHVIZ + DELIMITER;

    if (System.getProperty("os.name").startsWith("Windows")) {
        key += WINCOMMAND;
        path = System.getProperty(SYSPROPWINLAYOUTERPATH);
    } else {
        key += LINUXCOMMAND;
        path = System.getProperty(SYSPROPOTHERLAYOUTERPATH);
    }
    key += DELIMITER + CIRCO;

    if (path == null) {
        path = "";
    } else {
        path += File.separator;
    }

    if (System.getProperty("os.name").startsWith("Windows")) {
        return path + properties.getProperty(key);
    } else {
        return "/bin/sh " + path + properties.getProperty(key)
            + " " + path;
    }
}

/**
 * Returns the name of the default layouter.
 *
 * @return the name of the default layouter
 */
public static String getDefaultLayouter() {
    return properties.getProperty(LAYOUTER + DELIMITER +
 DEFAULTLAYOUTER);
}

/**
 * Loads the defaults form resource file.
 *
 * @return the default values
 */
private static Properties getDefaults() {
    Properties defaults = new Properties();

    try {
        defaults.load(LayouterProperties.class
                .getResourceAsStream(DEFAULTRESOURCE));
    } catch (IOException e) {
        defaults.clear();
    }

    return defaults;
}

/**
 * Returns the system specific command line to start the dot process.
 *
 * @return the dot command
 */
public static String getDotCommand() {
    String key;
    String path;

    key = LAYOUTER + DELIMITER + GRAPHVIZ + DELIMITER;

    if (System.getProperty("os.name").startsWith("Windows")) {
        key += WINCOMMAND;
        path = System.getProperty(SYSPROPWINLAYOUTERPATH);
    } else {
        key += LINUXCOMMAND;
        path = System.getProperty(SYSPROPOTHERLAYOUTERPATH);
    }
    key += DELIMITER + DOT;

    if (path == null) {
        path = "";
    } else {
        path += File.separator;
    }

    if (System.getProperty("os.name").startsWith("Windows")) {
        return path + properties.getProperty(key);
    } else {
        return "/bin/sh " + path + properties.getProperty(key)
            + " " + path;
    }
}

/**
 * Returns the resolution of dot graph.
 *
 * @return the resultion in dpi
 */
public static Integer getDPI() {
       return new Integer(properties.getProperty(LAYOUTER + DELIMITER
               + GRAPHVIZ + DELIMITER + DPI));
}
```

```
470
    /**
     * State shape for final states.
     *
     * @return shape as string (dot language)
     */
    public static String getFinalStateShape() {
        return properties.getProperty(LAYOUTER + DELIMITER + GRAPHVIZ
                + DELIMITER + FINALSTATESHAPE);
    }
480
    /**
     * You may use a bigger font while calling dot.
     *
     * @return the font scale
     */
    public static float getFontSizeScale() {
           return Float.parseFloat(properties.getProperty(LAYOUTER +
     DELIMITER
                   + GRAPHVIZ + DELIMITER + FONTSIZESCALE));
    }
490
    /**
     * Returns the horizontal alignment.
     *
     * @return "l" -> left
     *         "c" -> center
     *         "r" -> right
     */
    public static String getHorizontalLayerAlignment() {
        return properties.getProperty(LAYOUTER + DELIMITER + LAYERBASED
500             + DELIMITER + LAYERALIGNMENT + DELIMITER + HORIZONTAL);
    }

    /**
     * The <code>HVLayouter</code> defines a step width for placing states.
     *
     * @return the step width
     */
    public static int getHorizontalStateStepWidth() {
        return Integer.parseInt(properties.getProperty(LAYOUTER + DELIMITER
510                 + HV + DELIMITER + HORIZONTALSTATESTEPWIDTH));
    }

    /**
     * The <code>HVLayouter</code> defines a step width for placing
     transition
     * ports.
     *
     * @return the step width
     */
    public static int getHorizontalTransitionStepWidth() {
520     return Integer.parseInt(properties.getProperty(LAYOUTER + DELIMITER
                    + HV + DELIMITER + HORIZONTALTRANSITIONSTEPWIDTH));
    }

    /**
     * Returns the initial layout direction.
     *
     * @return "auto" -> best layout
     *         "horizontal" -> horizontal layout
     *         "vertical" -> vertical layout
530  */
    public static String getInitialLayoutDirection() {
        return properties.getProperty(LAYOUTER + DELIMITER + LAYERBASED
                + DELIMITER + INITIALDIRECTION);
      }

    /**
     * Transitions from the initial state may have a higher weight as
     others.
     *
     * @return the weight gain (0, if weight equals other transitions)
540  */
    public static float getInitialTransitionWeightGain() {
           return Float.parseFloat(properties.getProperty(LAYOUTER +
     DELIMITER
                   + GRAPHVIZ + DELIMITER + INITIALTRANSITION_WEIGHTGAIN));
    }

    /**
     * The <code>HVLayouter</code> defines a border around any label.
     *
     * @return the border
550  */
    public static int getLabelBorder() {
        return Integer.parseInt(properties.getProperty(LAYOUTER + DELIMITER
                   + HV + DELIMITER + LABELBORDER));
    }

    /**
     * Labeled transitions may have a lower weight as non-labeled.
     *
     * @return the weight loss (0, if weight equals non-labeled transtions)
560  */
    public static float getLabeledTransitionWeightLoss() {
           return Float.parseFloat(properties.getProperty(LAYOUTER +
     DELIMITER
                   + GRAPHVIZ + DELIMITER + LABELEDTRANSITION_WEIGHTLOSS));
    }

    /**
     * Returns the replacement for empty transition labels which have
     parallel
     * transitions.
     *
570  * @param horizontalLayout true, if layout direction is from horizontal
     * @return the replacement
     */
    public static String getLabelReplacement(final boolean horizontalLayout
     ) {
        String key = LAYOUTER + DELIMITER + GRAPHVIZ + DELIMITER
        + EMPTYLABELREPLACEMENT + DELIMITER;

        if (horizontalLayout) {
            key += HORIZONTAL;
        } else {
580         key += VERTICAL;
        }
```

```
        return properties.getProperty(key);
    }

    /**
     * Returns the log level.
     *
     * @return the log level
     */
    public static int getLogLevel() {
        return Integer.parseInt(properties.getProperty(LAYOUTER + DELIMITER
                + LOGLEVEL));
    }

    /**
     * Returns the mode of graphviz usage.
     *
     * @return "bin" -> binary mode
     *         "lib" -> library mode
     *         "both" -> both modes
     */
    public static String getMode() {
        return properties.getProperty(LAYOUTER + DELIMITER + GRAPHVIZ
                + DELIMITER + MODE);
    }

    /**
     * Returns the system specific command line to start the neato process.
     *
     * @return the neato command
     */
    public static String getNeatoCommand() {
        String key;
        String path;

        key = LAYOUTER + DELIMITER + GRAPHVIZ + DELIMITER;

        if (System.getProperty("os.name").startsWith("Windows")) {
            key += WINCOMMAND;
            path = System.getProperty(SYSPROPWINLAYOUTERPATH);
        } else {
            key += LINUXCOMMAND;
            path = System.getProperty(SYSPROPOTHERLAYOUTERPATH);
        }
        key += DELIMITER + NEATO;

        if (path == null) {
            path = "";
        } else {
            path += File.separator;
        }

        if (System.getProperty("os.name").startsWith("Windows")) {
            return path + properties.getProperty(key);
        } else {
            return "/bin/sh " + path + properties.getProperty(key)
                + " " + path;
        }
    }

    /**
     * Returns the priority angle depending an the specified layout
     direction.
     *
     * @param horizontalLayout true, if currentlay has horizontal layout
     * @return the angle of the priority
     */
    public static Integer getPriorityAngle(final boolean horizontalLayout)
     {
        String key = LAYOUTER + DELIMITER + GRAPHVIZ + DELIMITER +
     PRIORITYANGLE
                          + DELIMITER;
        String rankdir = getRankdir(horizontalLayout);

        if (rankdir.equalsIgnoreCase("LR")) {
            key += LEFTRIGHT;
        } else if (rankdir.equalsIgnoreCase("TB")) {
            key += TOPBOTTOM;
        } else if (rankdir.equalsIgnoreCase("RL")) {
            key += RIGHTLEFT;
        } else {
            key += BOTTOMTOP;
        }

        return new Integer(properties.getProperty(key));
    }

    /**
     * Returns the distance of properties form source port.
     *
     * @return the distance
     */
    public static Float getPriorityDistance() {
           return new Float(properties.getProperty(LAYOUTER + DELIMITER
                   + GRAPHVIZ + DELIMITER + PRIORITYDISTANCE));
    }

    /**
     * Returns the state shape for pseudo states.
     *
     * @return shape as string (dot language)
     */
    public static String getPseudoStateShape() {
        return properties.getProperty(LAYOUTER + DELIMITER + GRAPHVIZ
                + DELIMITER + PSEUDOSTATESHAPE);
    }

    /**
     * Returns the rankdirection.
     *
     * @param horizontalLayout true, if layout direction is from horizontal
     * @return "LR" -> left to right
     *         "RL" -> right to left
     *         "TB" -> top to bottom
     *         "BT" -> bottom to top
     */
    public static String getRankdir(final boolean horizontalLayout) {
        String key = LAYOUTER + DELIMITER + GRAPHVIZ + DELIMITER
                                       + RANKDIR + DELIMITER;

        if (horizontalLayout) {
```

```
700             key += HORIZONTAL;
            } else {
                key += VERTICAL;
            }

            return properties.getProperty(key);
        }

        /**
         * The <code>HVLayouter</code> places states using this alignment.
710      *
         *
         * @return "top/left" -> place states at top/left
         *         "center" -> centers sattes
         *         "bottom/right" -> place states at bottom/right
         */
        public static String getStateAlignment() {
            return properties.getProperty(LAYOUTER + DELIMITER
                    + HV + DELIMITER + STATEALIGNMENT);
        }
720
        /**
         * The <code>HVLayouter</code> defines a border around any state.
         *
         * @return the border
         */
        public static int getStateBorder() {
            return Integer.parseInt(properties.getProperty(LAYOUTER + DELIMITER
                    + HV + DELIMITER + STATEBORDER));
        }
730
        /**
         * Returns the state shape for normal states.
         *
         * @return shape as string (dot language)
         */
        public static String getStateShape() {
            return properties.getProperty(LAYOUTER + DELIMITER + GRAPHVIZ
                  + DELIMITER + STATESHAPE);
        }
740
        /**
         * Returns the system specific command line to start the twopi process.
         *
         * @return the twopi command
         */
        public static String getTwopiCommand() {
            String key;
            String path;

750         key = LAYOUTER + DELIMITER + GRAPHVIZ + DELIMITER;

            if (System.getProperty("os.name").startsWith("Windows")) {
                key += WINCOMMAND;
                path = System.getProperty(SYSPROPWINLAYOUTERPATH);
            } else {
                key += LINUXCOMMAND;
                path = System.getProperty(SYSPROPOTHERLAYOUTERPATH);
            }
            key += DELIMITER + TWOPI;
760
            if (path == null) {
                path = "";
            } else {
                path += File.separator;
            }

            if (System.getProperty("os.name").startsWith("Windows")) {
                return path + properties.getProperty(key);
            } else {
770             return "/bin/sh " + path + properties.getProperty(key)
                    + " " + path;
            }
        }

        /**
         * Returns the vertical alignment.
         *
         * @return "t" -> top
         *         "c" -> center
780      *         "b" -> bottom
         */
        public static String getVerticalLayerAlignment() {
            return properties.getProperty(LAYOUTER + DELIMITER + LAYERBASED
                    + DELIMITER + LAYERALIGNMENT + DELIMITER + VERTICAL);
        }

        /**
         * The <code>HVLayouter</code> defines a step width for placing states.
         *
790      * @return the step width
         */
        public static int getVerticalStateStepWidth() {
            return Integer.parseInt(properties.getProperty(LAYOUTER + DELIMITER
                    + HV + DELIMITER + VERTICALSTATESTEPWIDTH));
        }

        /**
         * The <code>HVLayouter</code> defines a step width for placing
         transition
         * ports.
800      *
         * @return the step width
         */
        public static int getVerticalTransitionStepWidth() {
            return Integer.parseInt(properties.getProperty(LAYOUTER + DELIMITER
                    + HV + DELIMITER + VERTICALTRANSITIONSTEPWIDTH));
        }
        /**
         * This method will load the properties from local file. If this file
         does
         * not exist, the defaults are written to local file.
810      *
         * @return true, if loading was successful
         */
        protected static boolean load()  {
            boolean success;

            properties = new Properties(getDefaults());
```

```
            if (!loadProperties()) {
                success = copyDefaults();
820         } else {
                success = true;
            }

            return success;
        }

        /**
         * Loads properties from user specific file.
         *
830      * @return true, if loading was successful
         */
        private static boolean loadProperties() {
            boolean success = true;

            try {
                properties.load(new FileInputStream(PROPERTIESFILE));
            } catch (IOException loadException) {
                Handler.instance().logError(PROPERTIESFILE
                        + " could not be found. (" + loadException + ")");
840             success = false;
            }

            return success;
        }

        /**
         * This method checks whether properties changed.
         *
         * @return true, if properties changed
850      */
        protected static boolean changed() {
            boolean result = false;

            try {
                Properties newProps = new Properties(getDefaults());
                newProps.load(new FileInputStream(PROPERTIESFILE));
                if (!newProps.equals(properties)) {
                    result = true;
860             }
            } catch (IOException loadException) {
                Handler.instance().logError(PROPERTIESFILE
                        + " could not be found. (" + loadException + ")");
            }

            return result;
        }

        /**
870      * Reloads the properties file.
         */
        protected static void reload() {
            loadProperties();
            Handler.instance().resetDefaultLayouter();
            Handler.instance().resetLogLevel();
        }

        /**
         * Is the feature <b>semantic zoom</b> active.
880      *
         * @return true if feature is active
         */
        public static boolean useSemanticZoom() {
            return properties.getProperty(LAYOUTER + DELIMITER + SEMANTICZOOM)
                    .equalsIgnoreCase("true");
        }

        /**
         * This is the construtor.
890      */
        private LayouterProperties() {
        }
    }
```

C.2.5. kiel.layouter.PseudoLayouter

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/05/20 11:45:48 $
 */
package kiel.layouter;

import kiel.configMngr.Configuration;
import kiel.dataStructure.StateChart;

import kiel.graphicalInformations.View;


/**
 * <p>
 * Description: The <code>PseudoLayouter</code> is not a real layouter. It
 *    is
 * used to have the original layout in store. The original layout will be
 * loaded from file or created by the editor.
 * </p>
 *
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 *
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.19 $
 */
public final class PseudoLayouter extends Layouter {
    /**
     * This is the constructor.
     */
    public PseudoLayouter() {
    }

    /**
     * @see Layouter#getConfigurationView(Configuration)
     */
    public View getConfigurationView(final Configuration config) {
        return getStaticView();
    }

    /**
     * @see Layouter#invalidateAllKnownViews()
     */
    public void invalidateAllKnownViews() {
        getStaticView().setInvalid();
    }

    /**
     * @see Layouter#layoutAllKnownViews()
     */
    public void layoutAllKnownViews() {
        layoutView(getStaticView());
    }

    /**
     * @see Layouter#layoutAllViews(Configuration[])
     */
    public void layoutAllViews(final Configuration[] configs) {
        layoutView(getStaticView());
    }

    /**
     * @see Layouter#layoutSingleView(View)
     */
    protected void layoutSingleView(final View view) {
        // nothing to do
        // the PseudoLayouter does no layout
    }

    /**
     * @see Layouter#setStatechart(StateChart)
     */
    public void setStatechart(final StateChart statechart) {
        try {
            super.setStatechart(statechart);
        } catch (LayoutException e) {
            // super class will never throw an exception
            Handler.instance().logError(
                    "LayoutException caught in PseudoLayouter. (" + e + ")
 ");
        }
    }

    /**
     * Call this method to set the static view.
     *
     * @param view a view
     */
    public void setView(final View view) {
        setStaticView(view);
    }
}
```

C.2.6. kiel.layouter.graphviz.GraphvizLayouter

```java
/*
 * $Author: tkl $
 *
 * $Date: 2005/05/02 10:14:39 $
 */
package kiel.layouter.graphviz;

import java.awt.Dimension;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Vector;

import kiel.dataStructure.Edge;
import kiel.dataStructure.FinalSimpleState;
import kiel.dataStructure.FinalState;
import kiel.dataStructure.InitialState;
import kiel.dataStructure.Node;
import kiel.dataStructure.Priority;
import kiel.dataStructure.PseudoState;
import kiel.dataStructure.Transition;
import kiel.dataStructure.TransitionLabel;
import kiel.graphicalInformations.NodeLayoutInformation;
import kiel.graphicalInformations.Properties;
import kiel.graphicalInformations.View;
import kiel.layouter.LayerwiseLayouter;
import kiel.layouter.LayouterProperties;

/**
 * <p>
 * Description: This is a basic class to uses the GraphViz tools
 * for layouting a single layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.23 $
 */
public abstract class GraphvizLayouter extends LayerwiseLayouter {
    /**
     * This is the constructor.
     */
    protected GraphvizLayouter() {
    }

    /**
     * Adds all edges.
     *
     * @param edges all edges in the graph
     * @param horizontalLayout true, if horizontal layout
     */
    private void addAllEdges(final Collection edges,
            final boolean horizontalLayout) {
        Iterator edgeIter;
        ArrayList edgesWithLabels = new ArrayList();
        ArrayList edgesWithoutLabels = new ArrayList();

        // sort edges by labels and priorities
        edgeIter = edges.iterator();
        while (edgeIter.hasNext()) {
            Edge edge = (Edge) edgeIter.next();

            if (edge instanceof Transition) {
                if (((Transition) edge).getLabel().toString().length() > 0)
  {
                    int i = 0;

                    for (i = 0; i < edgesWithLabels.size(); i++) {
                        if (((Transition) edgesWithLabels.get(i))
                                .getPriority().getValue()
                            < ((Transition) edge).getPriority().getValue())
  {
                            edgesWithLabels.add(i, edge);
                            break;
                        }
                    }
                    if (i == edgesWithLabels.size()) {
                        edgesWithLabels.add(edge);
                    }
                } else {
                    int i = 0;

                    for (i = 0; i < edgesWithoutLabels.size(); i++) {
                        if (((Transition) edgesWithoutLabels.get(i))
                                .getPriority().getValue()
                            < ((Transition) edge).getPriority().getValue())
  {
                            edgesWithoutLabels.add(i, edge);
                            break;
                        }
                    }
                    if (i == edgesWithoutLabels.size()) {
                        edgesWithoutLabels.add(edge);
                    }
                }
            } else {
                edgesWithoutLabels.add(edge);
            }
        }

        // if horizontal layout add Edges with labels first
        // if vertical layout Edges without labels first
        if (horizontalLayout) {
            edgeIter = edgesWithLabels.iterator();
        } else {
```

```
            edgeIter = edgesWithoutLabels.iterator();
        }
        while (edgeIter.hasNext()) {
            Edge edge = (Edge) edgeIter.next();
            addSingleEdge(edge, hasParallelEdge(edge, edges),
 horizontalLayout);
        }

        if (horizontalLayout) {
            edgeIter = edgesWithoutLabels.iterator();
        } else {
            edgeIter = edgesWithLabels.iterator();
        }
        while (edgeIter.hasNext()) {
            Edge edge = (Edge) edgeIter.next();
            addSingleEdge(edge, hasParallelEdge(edge, edges),
 horizontalLayout);
        }
    }

    /**
     * Adds all nodes.
     *
     * @param nodes the nodes
     * @param view the current view
     */
    private void addAllNodes(final Collection nodes, final View view) {
        Iterator nodeIter;
        Vector nodeBuffer = new Vector();
        boolean initialStateAdded = false;

        // add initial state first
        nodeIter = nodes.iterator();
        while (nodeIter.hasNext()) {
            Node node = (Node) nodeIter.next();
            NodeLayoutInformation nodeLayout;

            if (initialStateAdded) {
                nodeLayout = view.getLayoutInformation(node);
                addSingleNode(node, nodeLayout);
            } else {
                if (node instanceof InitialState) {
                    Iterator bufferIter;

                    nodeLayout = view.getLayoutInformation(node);
                    addSingleNode(node, nodeLayout);
                    bufferIter = nodeBuffer.iterator();
                    while (bufferIter.hasNext()) {
                        Node bufferedNode = (Node) bufferIter.next();

                        nodeLayout = view.getLayoutInformation(bufferedNode
 );

                        addSingleNode(bufferedNode, nodeLayout);
                    }
                    initialStateAdded = true;
                } else {
                    nodeBuffer.add(node);
                }
            }
        }

        // statechart has no initial state
        if (!initialStateAdded) {
            Iterator bufferIter;

            bufferIter = nodeBuffer.iterator();
            while (bufferIter.hasNext()) {
                Node node = (Node) bufferIter.next();
                NodeLayoutInformation nodeLayout = view
                        .getLayoutInformation(node);

                addSingleNode(node, nodeLayout);
            }
        }
    }

    /**
     * Adds a single edge to the graph.
     *
     * @param edge the edge
     * @param attributes the edge's attributes
     */
    protected abstract void addEdgeToGraph(final Edge edge,
            final Map attributes);

    /**
     * Adds a single node to the graph.
     *
     * @param node the node
     * @param attributes the node's attributes
     */
    protected abstract void addNodeToGraph(final Node node,
            final Map attributes);

    /**
     * Prepares the edge's attributes and adds it to the graph.
     *
     * @param edge the edge
     * @param hasParallelEdge true, if there are more edges with the same
     *                        direction
     * @param horizontalLayout true, if horizontal layout
     */
    private void addSingleEdge(final Edge edge, final boolean
     hasParallelEdge,
            final boolean horizontalLayout) {
        HashMap attributes = new HashMap();

        // set local edge attributes
        if (edge instanceof Transition) {
            TransitionLabel label = ((Transition) edge).getLabel();
            Priority prio = ((Transition) edge).getPriority();
            float transitionWeight = 1;
            String labelString;

            attributes.put("taillabel", "<" + prio.getValue() + ">");

            if (label.toString().length() > 0) {
                labelString = label.toString();
                transitionWeight -= LayouterProperties
                    .getLabeledTransitionWeightLoss();
```

```java
            } else if (hasParallelEdge) {
                labelString = LayouterProperties
                                .getLabelReplacement(horizontalLayout);
            } else {
                labelString = "";
            }

            if (((Transition) edge).getSource() instanceof InitialState) {
                transitionWeight += LayouterProperties
                                .getInitialTransitionWeightGain();
            }

            attributes.put("label", labelString);
            attributes.put("weight", Float.toString(transitionWeight));
        }

        addEdgeToGraph(edge, attributes);
    }

    /**
     * Prepares the nodes attributes and adds the node to the graph.
     *
     * @param node the node
     * @param nodeLayout the node's layout information
     */
    private void addSingleNode(final Node node,
            final NodeLayoutInformation nodeLayout) {
        String shape;
        float widthInInch;
        float heightInInch;
        int dpi;
        HashMap attributes;

        dpi = LayouterProperties.getDPI().intValue();
        widthInInch = (float) nodeLayout.getWidth() / (float) dpi;
        heightInInch = (float) nodeLayout.getHeight() / (float) dpi;
        if (node instanceof PseudoState) {
            shape = LayouterProperties.getPseudoStateShape();
        } else if ((node instanceof FinalState)
                || (node instanceof FinalSimpleState)) {
            shape = LayouterProperties.getFinalStateShape();
        } else {
            shape = LayouterProperties.getStateShape();
        }

        // set local node attributes
        attributes = new HashMap();
        attributes.put("label", node.getName());
        attributes.put("width", Float.toString(widthInInch));
        attributes.put("height", Float.toString(heightInInch));
        attributes.put("shape", shape);

        addNodeToGraph(node, attributes);
    }

    /**
     * Returns the graph's bounding box.
     *
     * @return the bounding box dimension
     */
    protected abstract Dimension getBoundingBox();

    /**
     * Returns true if <code>edge</code> has parallel edges.
     *
     * @param edge the edge
     * @param edges all edges
     * @return true, if <code>edge</code> has parallel edges
     */
    private boolean hasParallelEdge(final Edge edge, final Collection edges
     ) {
        Iterator iter = edges.iterator();
        boolean result = false;

        while (iter.hasNext()) {
            Edge anotherEdge = (Edge) iter.next();
            boolean sameDir =
                    (edge.getSource().getID()
                        .equals(anotherEdge.getSource().getID()))
                  && (edge.getTarget().getID()
                        .equals(anotherEdge.getTarget().getID()));
            boolean oppositeDir =
                    (edge.getSource().getID()
                        .equals(anotherEdge.getTarget().getID()))
                 && (edge.getTarget().getID()
                        .equals(anotherEdge.getSource().getID()));

            if (!edge.getID().equals(anotherEdge.getID())
                && (sameDir || oppositeDir)) {
                result = true;
                break;
            }
        }

        return result;
    }

    /**
     * Layouts the graph.
     */
    protected abstract void layoutGraph();

    /**
     * @see LayerwiseLayouter#layoutLayer(Collection, View, boolean)
     */
    protected final Dimension layoutLayer(final Collection subnodes,
            final View view, final boolean horizontalLayout) {
        String graphName;
        Iterator nodeIter;
        HashMap attributes;
        int fontsize;
        ArrayList edges;

        // build graph name and collect all edges
        graphName = "";
        edges = new ArrayList();
        nodeIter = subnodes.iterator();
        while (nodeIter.hasNext()) {
            Node node = (Node) nodeIter.next();
            graphName += node.getName();
```

```
            edges.addAll(node.getOutgoingTransitions());
        }

        // calculate fontsize
        fontsize = Math.round(Properties.getLabelFontSize()
350                 * LayouterProperties.getFontSizeScale());

        newGraph(graphName);

        // set graph attributes
        attributes = new HashMap();
        attributes.put("rankdir", LayouterProperties
                .getRankdir(horizontalLayout));
        attributes.put("fontname", Properties.getLabelFontFamily());
        attributes.put("fontsize", Integer.toString(
360             Properties.getDescriptionFontSize()));
        attributes.put("label", "");
        attributes.put("labelloc", "t");
        attributes.put("labeljust", "c");
        attributes.put("dpi", LayouterProperties.getDPI()
                .toString());
        setGraphAttributes(attributes);

        // set global node attributes
        attributes.clear();
370     attributes.put("fixedsize", "true");
        attributes.put("fontname", Properties.getLabelFontFamily());
        attributes.put("fontsize", Integer.toString(
                Properties.getDescriptionFontSize()));
        attributes.put("label", "\\N");
        attributes.put("style", "rounded");
        setGlobalNodeAttributes(attributes);

        // set global edge attributes
        attributes.clear();
380     attributes.put("labelangle", LayouterProperties
                .getPriorityAngle(horizontalLayout).toString());
        attributes.put("fontname", Properties.getLabelFontFamily());
        attributes.put("fontsize", Integer.toString(fontsize));
        attributes.put("labelfontname", Properties.getLabelFontFamily());
        attributes.put("labelfontsize", Integer.toString(fontsize));
        attributes.put("labeldistance", LayouterProperties
                .getPriorityDistance().toString());
        attributes.put("label", "");
        setGlobalEdgeAttributes(attributes);
390
        addAllNodes(subnodes, view);
        addAllEdges(edges, horizontalLayout);
        layoutGraph();
        retrieveLayoutInformations(subnodes, view);

        return getBoundingBox();
    }

    /**
400  * Creates a new graph.
     *
     * @param name the graph name
     */
    protected abstract void newGraph(final String name);

    /**
     * Retrieves the layout information for all components.
     *
     * @param nodes all graph nodes
410  * @param view the curretn view
     */
    protected abstract void retrieveLayoutInformations(final Collection
     nodes,
            final View view);

    /**
     * Sets the global edge attributes.
     *
     * @param attributes attribute <-> value pair
     */
420 protected abstract void setGlobalEdgeAttributes(final Map attributes);

    /**
     * Sets the global node attributes.
     *
     * @param attributes attribute <-> value pair
     */
    protected abstract void setGlobalNodeAttributes(final Map attributes);

    /**
430  * Sets the graph attributes.
     *
     * @param attributes attribute <-> value pair
     */
    protected abstract void setGraphAttributes(final Map attributes);
}
```

C.2.7. kiel.layouter.graphviz.GraphvizBinaryLayouter

```java
/*
 * $Author: tkl $
 *
 * $Date: 2005/05/25 08:33:18 $
 */
package kiel.layouter.graphviz;

import java.awt.Dimension;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.StringTokenizer;

import kiel.dataStructure.Edge;
import kiel.dataStructure.Node;
import kiel.dataStructure.Transition;
import kiel.graphicalInformations.CurvetoPath;
import kiel.graphicalInformations.EdgeLayoutInformation;
import kiel.graphicalInformations.LabelLayoutInformation;
import kiel.graphicalInformations.LinetoPath;
import kiel.graphicalInformations.MovetoPath;
import kiel.graphicalInformations.NodeLayoutInformation;
import kiel.graphicalInformations.Point;
import kiel.graphicalInformations.View;
import kiel.layouter.Handler;

/**
 * <p>
 * Description: This is a basic class to uses the GraphViz tools via the
 * command line for layouting a single layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.8 $
 */
public abstract class GraphvizBinaryLayouter extends GraphvizLayouter {
    /**
     * This is the dimension of bounding box from last read graph.
     */
    private Dimension boundingBox;

    /**
     * This is the stdin of dot process.
     */
    private PrintWriter dotInput;

    /**
     * This is the stdout of dot process.
     */
    private BufferedReader dotOutput;

    /**
     * This is the stderr of dot process.
     */
    private BufferedReader errorStream;
    /**
     * This is the dot process.
     */
    private Process graphvizProcess;

    /**
     * Maps the object id to the object itself.
     */
    private HashMap idToObject;

    /**
     * This is the constructor.
     */
    protected GraphvizBinaryLayouter() {
        idToObject = new HashMap();
        boundingBox = new Dimension();
        startProcess();
    }

    /**
     * @see GraphvizLayouter#addEdgeToGraph(Edge, Map)
     */
    protected final void addEdgeToGraph(final Edge edge, final Map
     attributes) {
        Iterator iter = attributes.keySet().iterator();

        dotInput.print("\"" + edge.getSource().getID() + "\" -> \""
              + edge.getTarget().getID() + "\" [");

        while (iter.hasNext()) {
            String key = (String) iter.next();

            dotInput.print(key + "=\"" + attributes.get(key) + "\"");
            if (iter.hasNext()) {
                dotInput.print(", ");
            }
        }

        dotInput.println(", comment=\"" + edge.getID() + "\"];");

        idToObject.put(edge.getID(), edge);
    }

    /**
     * @see GraphvizLayouter#addNodeToGraph(Node, Map)
```

```
          */
         protected final void addNodeToGraph(final Node node, final Map
          attributes) {
             Iterator iter = attributes.keySet().iterator();

             dotInput.print("\"" + node.getID() + "\" [");

             while (iter.hasNext()) {
                 String key = (String) iter.next();
120
                 dotInput.print(key + "=\"" + attributes.get(key) + "\"");
                 if (iter.hasNext()) {
                     dotInput.print(", ");
                 }
             }

             dotInput.println("];");

             idToObject.put(node.getID(), node);
130      }

         /**
          * @see Object#finalize()
          */
         protected final void finalize() throws Throwable {
             // kill dot process
             graphvizProcess.destroy();
             super.finalize();
         }
140
         /**
          * @see GraphvizLayouter#getBoundingBox()
          */
         protected final Dimension getBoundingBox() {
             return boundingBox;
         }

         /**
          * Returns the system specific command line of this layouter.
150       *
          * @return the commad line
          */
         protected abstract String getCommandLine();

         /**
          * Parses the edge pos attribute an returns a svg path.
          *
          * @param pos
          *            the pos attribute of an edge
160       * @return SVG Path of <code>edge</code>
          */
         private ArrayList getEdgePath(final String pos) {
             ArrayList path = new ArrayList();
             String posString;
             StringTokenizer posTokenizer;
             Point endPoint = null;
             Point startPoint = null;
             Point moveToPoint = null;

170          posTokenizer = new StringTokenizer(pos, " ");
```

```
             while (moveToPoint == null) {
                 String token = posTokenizer.nextToken();
                 StringTokenizer pointTokenizer = new StringTokenizer(token,
         ",");

                 if (pointTokenizer.countTokens() > 2) {
                     if (pointTokenizer.nextToken().equalsIgnoreCase("e")) {
                         endPoint = new Point(Integer.parseInt(
                                 pointTokenizer.nextToken()),
                                 boundingBox.height
180                              - Integer.parseInt(pointTokenizer.nextToken()))
         ;
                     } else {
                         startPoint = new Point(
                                 Integer.parseInt(pointTokenizer.nextToken()),
                                 boundingBox.height
                                 - Integer.parseInt(pointTokenizer.nextToken()))
         ;
                     }
                 } else {
                     moveToPoint = new Point(Integer.parseInt(
                             pointTokenizer.nextToken()),
190                          boundingBox.height
                             - Integer.parseInt(pointTokenizer.nextToken()));
                     path.add(new MovetoPath(moveToPoint));
                     if (startPoint != null) {
                         path.add(new LinetoPath(startPoint));
                         path.add(new MovetoPath(moveToPoint));
                     }
                 }
             }

200          while (posTokenizer.hasMoreTokens()) {
                 String token;
                 StringTokenizer pointTokenizer;

                 token = posTokenizer.nextToken();
                 pointTokenizer = new StringTokenizer(token, ",");
                 Point cubic1 = new Point(
                         Integer.parseInt(pointTokenizer.nextToken()),
                         boundingBox.height - Integer.parseInt(
                                 pointTokenizer.nextToken()));
210
                 token = posTokenizer.nextToken();
                 pointTokenizer = new StringTokenizer(token, ",");
                 Point cubic2 = new Point(
                         Integer.parseInt(pointTokenizer.nextToken()),
                         boundingBox.height - Integer.parseInt(
                                 pointTokenizer.nextToken()));

                 token = posTokenizer.nextToken();
                 pointTokenizer = new StringTokenizer(token, ",");
220              Point cubic3 = new Point(
                         Integer.parseInt(pointTokenizer.nextToken()),
                         boundingBox.height - Integer.parseInt(
                                 pointTokenizer.nextToken()));

                 path.add(new CurvetoPath(cubic1, cubic2, cubic3));
             }
```

```
        if (endPoint != null) {
            path.add(new LinetoPath(endPoint));
        }

        return path;
    }

    /**
     * @see GraphvizLayouter#layoutGraph()
     */
    protected final void layoutGraph() {
        dotInput.println("}");
        dotInput.flush();
    }

    /**
     * @see GraphvizLayouter#newGraph(String)
     */
    protected final void newGraph(final String name) {
        idToObject.clear();
        dotInput.println("digraph \"" + name + "\" {");
    }

    /**
     * Parses the given attribute string.
     *
     * @param line line to be parsed
     * @return a map of attribute <-> value pairs
     */
    private Map readAttributes(final String line) {
        HashMap map = new HashMap();
        String attr = null;
        String value = null;
        int beginIndex = 0;
        int i = 0;
        boolean markedValue;

        while (i < line.length()) {
            attr = null;
            value = null;

            while ((line.charAt(i) == ',') || (line.charAt(i) == ' ')) {
                i++;
            }
            beginIndex = i;

            // looking for attribute
            while (attr == null) {
                if (line.charAt(i) == '=') {
                    attr = line.substring(beginIndex, i);
                    beginIndex = i + 1;
                }
                i++;
            }

            // looking for value
            markedValue = false;
            while (value == null) {
                switch (line.charAt(i)) {
                    case '\\':
                        // escape char, ignore next
                        i++;
                        break;
                    case '\"':
                        if (markedValue) {
                            value = line.substring(beginIndex, i);
                        } else {
                            beginIndex++;
                            markedValue = true;
                        }
                        break;
                    case ',':
                        if (!markedValue) {
                            value = line.substring(beginIndex, i);
                        }
                        break;
                    default:
                        // all other chars are element of the attribute
                }

                i++;
                if ((i >= line.length()) && (value == null)) {
                    value = line.substring(beginIndex, i);
                }
            }

            map.put(attr, value);
        }

        return map;
    }

    /**
     * Returns str with leading spaces omitted.
     *
     * @param str a string
     * @return str without leading spaces
     */
    private String removeLeadingSpaces(final String str) {
        int i;

        for (i = 0; i < str.length(); i++) {
            if (!Character.isWhitespace(str.charAt(i))) {
                break;
            }
        }

        return str.substring(i);
    }

    /**
     * @see GraphvizLayouter#retrieveLayoutInformations(Collection, View)
     */
    protected final void retrieveLayoutInformations(final Collection nodes,
            final View view) {
        boolean endOfGraph;

        // read error stream
        try {
            while (errorStream.ready()) {
```

```
                Handler.instance().logError("GraphvizBinary: "
                        + errorStream.readLine());
            }
        } catch (IOException e) {
            Handler.instance().logError(
                    "error while reading dot error stream. (" + e + ")");
        }

        // read output stream
        try {
            endOfGraph = false;
            while (!endOfGraph) {
                String line;
                boolean endOfLine;

                line = "";
                endOfLine = false;
                while (!endOfLine) {
                    String read = dotOutput.readLine();
                    Handler.instance().logDebug("read: " + read);
                    if (read.endsWith("\\")) {
                        line += read.substring(0, read.length() - 1);
                    } else {
                        line += read;
                        endOfLine = true;
                    }
                }

                if (line.matches(".*->.*")) {
                    // edge found
                    Map attributes;
                    Edge edge;
                    EdgeLayoutInformation edgeLayout;

                    attributes = readAttributes(line.substring(
                            line.indexOf('[') + 1, line.lastIndexOf(']')));
                    edge = (Edge) idToObject.get(attributes.get("comment"))
;
                    edgeLayout = view.getLayoutInformation(edge);
                    edgeLayout.setPath(getEdgePath(
                            (String) attributes.get("pos")));

                    if (edge instanceof Transition) {
                        LabelLayoutInformation labelLayout;
                        StringTokenizer posTokenizer;

                        if (((Transition) edge).getLabel().toString()
                                .length() > 0) {
                            labelLayout = view.getLayoutInformation(
                                    ((Transition) edge).getLabel());
                            labelLayout.setDimension(
                                    ((Transition) edge).getLabel().toString
());
                            posTokenizer = new StringTokenizer(
                                    (String) attributes.get("lp"), ",");
                            labelLayout.setCenterPoint(new Point(
                                    Integer.parseInt(posTokenizer.nextToken
()),
                                            boundingBox.height - Integer.parseInt(
                                                    posTokenizer.nextToken())));
                        }

                        labelLayout = view.getLayoutInformation(
                                ((Transition) edge).getPriority());
                        labelLayout.setDimension(
                                (String) attributes.get("taillabel"));
                        posTokenizer = new StringTokenizer(
                                (String) attributes.get("tail_lp"), ",");
                        labelLayout.setCenterPoint(new Point(
                                Integer.parseInt(posTokenizer.nextToken()),
                                boundingBox.height
                                - Integer.parseInt(posTokenizer.nextToken()
)));
                    }
                } else if (line.matches(".*\".*\" \\[.*")) {
                    // node found
                    StringTokenizer st = new StringTokenizer(line);
                    Node node;
                    NodeLayoutInformation layoutInfo;
                    Map attributes;

                    st.nextToken("\""); // \t
                    node = (Node) idToObject.get(st.nextToken("\""));
                    layoutInfo = view.getLayoutInformation(node);
                    st.nextToken("\"["); // " [
                    attributes = readAttributes(st.nextToken("[]"));
                    st = new StringTokenizer((String) attributes.get("pos")
,
                            ",");
                    layoutInfo.setCenterPoint(new Point(
                            Integer.parseInt(st.nextToken()),
                            boundingBox.height
                            - Integer.parseInt(st.nextToken())));
                } else if (line.matches(".*graph \\[bb.*")) {
                    // bounding box found
                    StringTokenizer st = new StringTokenizer(line, "=,\"");

                    st.nextToken(); // graph [bb
                    st.nextToken(); // 0
                    st.nextToken(); // 0
                    boundingBox.width = Integer.parseInt(st.nextToken());
                    boundingBox.height = Integer.parseInt(st.nextToken());
                } else if (line.startsWith("}")) {
                    endOfGraph = true;
                }
            }
        } catch (IOException e) {
            Handler.instance().logError(
                    "Error while reading dot output stream. (" + e + ")");
        }
    }

    /**
     * @see GraphvizLayouter#setGlobalEdgeAttributes(Map)
     */
    protected final void setGlobalEdgeAttributes(final Map attributes) {
        Iterator iter = attributes.keySet().iterator();

        dotInput.print("edge [");
```

```
            while (iter.hasNext()) {
                String key = (String) iter.next();

                dotInput.print(key + "=\"" + attributes.get(key) + "\"");
                if (iter.hasNext()) {
                    dotInput.print(", ");
                }
470         }

            dotInput.println("];");
        }

        /**
         * @see GraphvizLayouter#setGlobalNodeAttributes(Map)
         */
        protected final void setGlobalNodeAttributes(final Map attributes) {
            Iterator iter = attributes.keySet().iterator();
480
            dotInput.print("node [");

            while (iter.hasNext()) {
                String key = (String) iter.next();

                dotInput.print(key + "=\"" + attributes.get(key) + "\"");
                if (iter.hasNext()) {
                    dotInput.print(", ");
                }
490         }

            dotInput.println("];");
        }

        /**
         * @see GraphvizLayouter#setGraphAttributes(Map)
         */
        protected final void setGraphAttributes(final Map attributes) {
            Iterator iter = attributes.keySet().iterator();
500
            dotInput.print("graph [");

            while (iter.hasNext()) {
                String key = (String) iter.next();

                dotInput.print(key + "=\"" + attributes.get(key) + "\"");
                if (iter.hasNext()) {
                    dotInput.print(", ");
                }
510         }

            dotInput.println("];");
        }

        /**
         * Starts the dot prosess.
         *
         * @return returns true, if the dot process could be started
         */
520     private boolean startProcess() {
            boolean succsess = true;

            try {
                graphvizProcess = Runtime.getRuntime().exec(getCommandLine());
                dotOutput = new BufferedReader(new InputStreamReader(
                        graphvizProcess.getInputStream()));
                errorStream = new BufferedReader(new InputStreamReader(
                        graphvizProcess.getErrorStream()));
                dotInput = new PrintWriter(graphvizProcess.getOutputStream());
530         } catch (IOException e) {
                Handler.instance().logError("Unable to start process. (" + e +
         ")");
                succsess = false;
            }

            return succsess;
        }
    }
```

C.2.8. kiel.layouter.graphviz.CircoBinLayouter

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/11 18:18:28 $
 */
package kiel.layouter.graphviz;

import kiel.layouter.LayouterProperties;

/**
 * <p>
 * Description: This layouter uses the GraphViz circo algorithm via the
 * command line to layout a given layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.6 $
 */
public final class CircoBinLayouter extends GraphvizBinaryLayouter {
    /**
     * @see GraphvizBinaryLayouter#getCommandLine()
     */
    protected String getCommandLine() {
        return LayouterProperties.getCircoCommand();
    }
}
```

C.2.9. `kiel.layouter.graphviz.DotBinLayouter`

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/11 18:18:28 $
 */
package kiel.layouter.graphviz;

import kiel.layouter.LayouterProperties;

/**
 * <p>
 * Description: This layouter uses the GraphViz dot algorithm via the
 * command line to layout a given layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.6 $
 */
public final class DotBinLayouter extends GraphvizBinaryLayouter {
    /**
     * @see GraphvizBinaryLayouter#getCommandLine()
     */
    protected String getCommandLine() {
        return LayouterProperties.getDotCommand();
    }
}
```

C.2.10. kiel.layouter.graphviz.NeatoBinLayouter

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/11 18:18:28 $
 */
package kiel.layouter.graphviz;

import kiel.layouter.LayouterProperties;

/**
 * <p>
 * Description: This layouter uses the GraphViz neato algorithm via the
 * command line to layout a given layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.6 $
 */
public final class NeatoBinLayouter extends GraphvizBinaryLayouter {
    /**
     * @see GraphvizBinaryLayouter#getCommandLine()
     */
    protected String getCommandLine() {
        return LayouterProperties.getNeatoCommand();
    }
}
```

C.2.11. kiel.layouter.graphviz.TwopiBinLayouter

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/11 18:18:28 $
 */
package kiel.layouter.graphviz;

import kiel.layouter.LayouterProperties;

/**
 * <p>
 * Description: This layouter uses the GraphViz twopi algorithm via the
 * command line to layout a given layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.6 $
 */
public final class TwopiBinLayouter extends GraphvizBinaryLayouter {
    /**
     * @see GraphvizBinaryLayouter#getCommandLine()
     */
    protected String getCommandLine() {
        return LayouterProperties.getTwopiCommand();
    }
}
```

C.2.12. kiel.layouter.graphviz.GraphvizLibraryLayouter

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/11 18:18:28 $
 */
package kiel.layouter.graphviz;

import java.awt.Dimension;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import kiel.dataStructure.Edge;
import kiel.dataStructure.Node;
import kiel.dataStructure.Priority;
import kiel.dataStructure.Transition;
import kiel.dataStructure.TransitionLabel;
import kiel.graphicalInformations.EdgeLayoutInformation;
import kiel.graphicalInformations.LabelLayoutInformation;
import kiel.graphicalInformations.NodeLayoutInformation;
import kiel.graphicalInformations.Point;
import kiel.graphicalInformations.View;
import kiel.layouter.Handler;


/**
 * <p>
 * Description: This is the basic layouter for all layouters which uses any
 * GraphViz algorithm via the graphviz library to layout a given layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.7 $
 */
public abstract class GraphvizLibraryLayouter extends GraphvizLayouter {
    /**
     * Make sure the library is initialized only once.
     */
    private static boolean initialized = false;

    /**
     * This is the dimension of bounding box from last read graph.
     */
    private Dimension boundingBox;

    /**
     * Maps from edge to C pointer.
     */
    private HashMap edgeToPointerMap;

    /**
     * This is the value of the graphviz library graph pointer.
     */
    private int graphPointer;

    /**
     * Maps from node to C pointer.
     */
    private HashMap nodeToPointerMap;

    /**
     * This is the constructor.
     */
    protected GraphvizLibraryLayouter() {
        graphPointer = 0;
        nodeToPointerMap = new HashMap();
        edgeToPointerMap = new HashMap();
        boundingBox = new Dimension();

        if (!initialized) {
            GraphvizAPI.initialize();
            initialized = true;
        }
        Handler.instance().logInfo("Using GraphViz version: "
                + GraphvizAPI.getVersionString());
    }

    /**
     * @see GraphvizLayouter#addEdgeToGraph(Edge, Map)
     */
    protected final void addEdgeToGraph(final Edge edge, final Map
 attributes) {
        int edgePointer;
        int sourcePointer;
        int targetPointer;
        Iterator iter;

        sourcePointer = ((Integer) nodeToPointerMap.get(edge.getSource()))
                                   .intValue();
        targetPointer = ((Integer) nodeToPointerMap.get(edge.getTarget()))
                                   .intValue();

        edgePointer = GraphvizAPI.createEdge(graphPointer, sourcePointer,
                                             targetPointer);
        edgeToPointerMap.put(edge, new Integer(edgePointer));

        // set local edge attributes
        iter = attributes.keySet().iterator();
        while (iter.hasNext()) {
            String key = (String) iter.next();

            GraphvizAPI.setLocalEdgeAttribute(graphPointer, edgePointer,
 key,
                    (String) attributes.get(key));
```

```
        }
    }

    /**
     * @see GraphvizLayouter#addNodeToGraph(Node, Map)
     */
    protected final void addNodeToGraph(final Node node, final Map
     attributes) {
        int nodePointer;
        Iterator iter;
120
        nodePointer = GraphvizAPI.createNode(graphPointer, node.getID());
        nodeToPointerMap.put(node, new Integer(nodePointer));

        // set local node attributes
        iter = attributes.keySet().iterator();
        while (iter.hasNext()) {
            String key = (String) iter.next();

            GraphvizAPI.setLocalNodeAttribute(graphPointer, nodePointer,
     key,
130                 (String) attributes.get(key));
        }
    }

    /**
     * Cleans up and closes.
     */
    private void closeGraph() {
        if (graphPointer != 0) {
            doCleanup(graphPointer);
140         GraphvizAPI.closeGraph(graphPointer);
        }
    }

    /**
     * Invokes the cleanup routine.
     *
     * @param gPointer
     *            pointer to the C graph object
     */
150 protected abstract void doCleanup(final int gPointer);

    /**
     * Invokes layout.
     *
     * @param gPointer
     *            pointer to the C graph object
     */
    protected abstract void doLayout(final int gPointer);

160 /**
     * @see Object#finalize()
     */
    protected final void finalize() throws Throwable {
        closeGraph();
        super.finalize();
    }

    /**
     * @see GraphvizLayouter#getBoundingBox()
170  */
    protected final Dimension getBoundingBox() {
        return boundingBox;
    }

    /**
     * @see GraphvizLayouter#layoutGraph()
     */
    protected final void layoutGraph() {
        // doing layout
180     doLayout(graphPointer);
        GraphvizAPI.attachAttributes(graphPointer);
    }

    /**
     * @see GraphvizLayouter#newGraph(String)
     */
    protected final void newGraph(final String name) {
        nodeToPointerMap.clear();
        edgeToPointerMap.clear();
190
        closeGraph();
        graphPointer = GraphvizAPI.createGraph(name);
    }

    /**
     * @see GraphvizLayouter#retrieveLayoutInformations(Collection, View)
     */
    protected final void retrieveLayoutInformations(final Collection nodes,
            final View view) {
200     Iterator nodeIter;

        // retrieve graph layout informations
        boundingBox = GraphvizAPI.getBoundingBox(graphPointer);

        nodeIter = nodes.iterator();
        while (nodeIter.hasNext()) {
            Node node = (Node) nodeIter.next();
            int nodePointer;
            Point nodePoint;
210         NodeLayoutInformation nodeLayout;
            Iterator edgeIter;

            // retrieve node layout informations
            nodePointer = ((Integer) nodeToPointerMap.get(node)).intValue()
     ;
            nodePoint = GraphvizAPI.getNodePos(nodePointer, boundingBox.
     height);
            nodeLayout = view.getLayoutInformation(node);
            nodeLayout.setCenterPoint(nodePoint);

            // retrieve edge layout informations
220         edgeIter = node.getOutgoingTransitions().iterator();
            while (edgeIter.hasNext()) {
                Edge edge = (Edge) edgeIter.next();
                int edgePointer;
                EdgeLayoutInformation edgeLayout;
                ArrayList pathElements;
```

```
                edgePointer = ((Integer) edgeToPointerMap.get(edge)).
     intValue();
                pathElements = GraphvizAPI.getEdgePath(edgePointer,
                        boundingBox.height);
230             edgeLayout = view.getLayoutInformation(edge);
                edgeLayout.setPath(pathElements);

                if (edge instanceof Transition) {
                    TransitionLabel label = ((Transition) edge).getLabel();
                    Priority prio = ((Transition) edge).getPriority();
                    Point labelPos;
                    Point prioPos;
                    LabelLayoutInformation labelLayout;
                    LabelLayoutInformation prioLayout;
240
                    // retrieve label layout informations
                    labelPos = GraphvizAPI.getLabelPos(edgePointer,
                            boundingBox.height);
                    labelLayout = view.getLayoutInformation(label);
                    labelLayout.setDimension(GraphvizAPI
                            .getLabelString(edgePointer));
                    labelLayout.setCenterPoint(labelPos);

                    // retrieve priority layout informations
250                 prioPos = GraphvizAPI.getPrioPos(edgePointer,
                            boundingBox.height);
                    prioLayout = view.getLayoutInformation(prio);
                    prioLayout.setDimension(GraphvizAPI
                            .getTailLabelString(edgePointer));
                    prioLayout.setCenterPoint(prioPos);
                }
            }
        }
    }
260
    /**
     * @see GraphvizLayouter#setGlobalEdgeAttributes(Map)
     */
    protected final void setGlobalEdgeAttributes(final Map attributes) {
        Iterator iter = attributes.keySet().iterator();

        while (iter.hasNext()) {
            String key = (String) iter.next();

270         GraphvizAPI.setGlobalEdgeAttribute(graphPointer, key,
                    (String) attributes.get(key));
        }
    }

    /**
     * @see GraphvizLayouter#setGlobalNodeAttributes(Map)
     */
    protected final void setGlobalNodeAttributes(final Map attributes) {
        Iterator iter = attributes.keySet().iterator();
280
        while (iter.hasNext()) {
            String key = (String) iter.next();

            GraphvizAPI.setGlobalNodeAttribute(graphPointer, key,
                    (String) attributes.get(key));
        }
    }

    /**
290  * @see GraphvizLayouter#setGraphAttributes(Map)
     */
    protected final void setGraphAttributes(final Map attributes) {
        Iterator iter = attributes.keySet().iterator();

        while (iter.hasNext()) {
            String key = (String) iter.next();

            GraphvizAPI.setGraphAttribute(graphPointer, key,
                    (String) attributes.get(key));
300     }
    }
}
```

C.2.13. `kiel.layouter.graphviz.GraphvizAPI`

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/14 15:06:42 $
 */
package kiel.layouter.graphviz;

import java.awt.Dimension;
import java.util.ArrayList;
import java.util.StringTokenizer;

import kiel.graphicalInformations.CurvetoPath;
import kiel.graphicalInformations.LinetoPath;
import kiel.graphicalInformations.MovetoPath;
import kiel.graphicalInformations.Point;


/**
 * <p>
 * Description: Defines the JNI interface to the graphviz library.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.15 $
 */
public final class GraphvizAPI {
    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_BOUNDINGBOX   = "bb";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_DPI           = "dpi";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_FIXEDSIZE     = "fixedsize";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_FONTNAME      = "fontname";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_FONTSIZE      = "fontsize";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_HEADLABEL     = "headlabel";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_HEIGHT        = "height";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_LABEL         = "label";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_LABELANGLE    = "labelangle";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_LABELDISTANCE = "labeldistance";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_LABELFONTNAME = "labelfontname";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_LABELFONTSIZE = "labelfontsize";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_LABELJUST     = "labeljust";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_LABELLOC      = "labelloc";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_LP            = "lp";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_POS           = "pos";
```

```
    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_RANKDIR       = "rankdir";

    /**
     * This is a graphviz attribute.
120  */
    protected static final String ATTR_SHAPE         = "shape";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_TAILLABEL     = "taillabel";

    /**
     * This is a graphviz attribute.
130  */
    protected static final String ATTR_TAILLP        = "tail_lp";

    /**
     * This is a graphviz attribute.
     */
    protected static final String ATTR_WEIGHT        = "weight";

    /**
     * This is a graphviz attribute.
140  */
    protected static final String ATTR_WIDTH         = "width";

    static {
        System.loadLibrary("GraphvizAPI");
    }

    /**
     * This is the native method to attach layout informations to graph.
     *
150  * @param graph
     *            pointer to the C graph object
     */
    protected static native void attachAttributes(final int graph);

    /**
     * This is the native method to invoke the circo cleanup routine.
     *
     * @param graph
     *            pointer to the C graph object
160  */
    protected static native void circoCleanup(final int graph);

    /**
     * This is the native method to close the graph.
     *
     * @param graph
     *            pointer to the C graph object
     */
    protected static native void closeGraph(final int graph);
170
    /**
     * This is the native method to create an edge.
     *
     * @param graph
     *            pointer to the C graph object
     * @param source
     *            pointer to the C node object of source node
     * @param target
     *            pointer to the C node object of target node
180  * @return pointer to the C edge object
     */
    protected static native int createEdge(final int graph,
            final int source, final int target);

    /**
     * This is the native method to create the graph.
     *
     * @param name
     *            graph's name
190  * @return pointer to the C graph object
     */
    protected static native int createGraph(final String name);

    /**
     * This is the native method to create a node.
     *
     * @param graph
     *            pointer to the C graph object
     * @param name
200  *            node's name
     * @return pointer to the C node object
     */
    protected static native int createNode(final int graph, final String
     name);

    /**
     * This is the native method to invoke the circo layout.
     *
     * @param graph
     *            pointer to the C graph object
210  */
    protected static native void doCircoLayout(final int graph);

    /**
     * This is the native method to invoke the dot layout.
     *
     * @param graph
     *            pointer to the C graph object
     */
    protected static native void doDotLayout(final int graph);
220
    /**
     * This is the native method to invoke the neato layout.
     *
     * @param graph
     *            pointer to the C graph object
     */
    protected static native void doNeatoLayout(final int graph);

    /**
230  * This is the native method to invoke the dot cleanup routine.
     *
```

```
         * @param graph
         *            pointer to the C graph object
         */
        protected static native void dotCleanup(final int graph);

        /**
         * This is the native method to invoke the twopi layout.
         *
240      * @param graph
         *            pointer to the C graph object
         */
        protected static native void doTwopiLayout(final int graph);

        /**
         * This is the native method to get an attribute.
         *
         * @param elem
         *            pointer to the C [graph|node|edge] object
250      * @param attr
         *            attribute's name (see constants)
         * @return attribute's value
         */
        protected static native String getAttribute(final int elem,
                final String attr);

        /**
         * Returns the boundling box of the layouted graph.
         *
260      * @param graph
         *            pointer to the C graph object
         * @return bounding box of the layouted graph
         */
        protected static Dimension getBoundingBox(final int graph) {
            StringTokenizer st;
            String attributeString;
            String widthString;
            String heightString;

270         attributeString = getAttribute(graph, ATTR_BOUNDINGBOX);
            st = new StringTokenizer(attributeString, ",");
            st.nextToken(); // can be ignored, always 0
            st.nextToken(); // can be ignored, always 0
            widthString = st.nextToken();
            heightString = st.nextToken();

            return new Dimension(Integer.parseInt(widthString), Integer
                    .parseInt(heightString));
        }
280
        /**
         * Returns the path of the edge.
         *
         * @param edge
         *            pointer to the C edge object
         * @param layerHeight
         *            the height of the bounding box
         *            (used to invert the y coordinate)
         * @return SVG Path of <code>edge</code>
290      */
        protected static ArrayList getEdgePath(final int edge,
                final int layerHeight) {
            ArrayList path = new ArrayList();
            String posString;
            StringTokenizer posTokenizer;
            Point endPoint = null;
            Point startPoint = null;
            Point moveToPoint = null;

300         posString = getAttribute(edge, ATTR_POS);
            posTokenizer = new StringTokenizer(posString, " ");
            while (moveToPoint == null) {
                String token = posTokenizer.nextToken();
                StringTokenizer pointTokenizer = new StringTokenizer(token,
    ",");

                if (pointTokenizer.countTokens() > 2) {
                    if (pointTokenizer.nextToken().equalsIgnoreCase("e")) {
                        endPoint = new Point(Integer.parseInt(
                                pointTokenizer.nextToken()),
310                             layerHeight
                                - Integer.parseInt(pointTokenizer.nextToken()))
    ;
                    } else {
                        startPoint = new Point(
                                Integer.parseInt(pointTokenizer.nextToken()),
                                layerHeight
                                - Integer.parseInt(pointTokenizer.nextToken()))
    ;
                    }
                } else {
                    moveToPoint = new Point(Integer.parseInt(
320                         pointTokenizer.nextToken()),
                            layerHeight
                            - Integer.parseInt(pointTokenizer.nextToken()));
                    path.add(new MovetoPath(moveToPoint));
                    if (startPoint != null) {
                        path.add(new LinetoPath(startPoint));
                        path.add(new MovetoPath(moveToPoint));
                    }
                }
            }
330
            while (posTokenizer.hasMoreTokens()) {
                String token;
                StringTokenizer pointTokenizer;

                token = posTokenizer.nextToken();
                pointTokenizer = new StringTokenizer(token, ",");
                Point cubic1 = new Point(
                        Integer.parseInt(pointTokenizer.nextToken()),
                        layerHeight - Integer.parseInt(pointTokenizer.nextToken
    ()));
340
                token = posTokenizer.nextToken();
                pointTokenizer = new StringTokenizer(token, ",");
                Point cubic2 = new Point(
                        Integer.parseInt(pointTokenizer.nextToken()),
                        layerHeight - Integer.parseInt(pointTokenizer.nextToken
    ()));
```

```
            token = posTokenizer.nextToken();
            pointTokenizer = new StringTokenizer(token, ",");
            Point cubic3 = new Point(
350                 Integer.parseInt(pointTokenizer.nextToken()),
                    layerHeight - Integer.parseInt(pointTokenizer.nextToken
    ()));

            path.add(new CurvetoPath(cubic1, cubic2, cubic3));
        }

        if (endPoint != null) {
            path.add(new LinetoPath(endPoint));
        }

360     return path;
    }

    /**
     * Returns the position of the label.
     *
     * @param edge
     *            pointer to the C edge object
     * @param layerHeight
     *            the height of the bounding box
370  *            (used to invert the y coordinate)
     * @return center point of edge's label
     */
    protected static Point getLabelPos(final int edge, final int
     layerHeight) {
        Point pos = new Point(-1, -1);
        String posString;
        StringTokenizer st;

        posString = getAttribute(edge, ATTR_LP);
        if ((posString != null) && (posString.length() > 0)) {
380         st = new StringTokenizer(posString, ",");
            pos = new Point(Integer.parseInt(st.nextToken()),
                    layerHeight - Integer.parseInt(st.nextToken()));
        } else {
            pos = new Point(0, 0);
        }

        return pos;
    }

390 /**
     * Returns the label as string.
     *
     * @param elem
     *            pointer to the C [graph|node|edge] object
     * @return the label string of the element
     */
    protected static String getLabelString(final int elem) {
        return getAttribute(elem, ATTR_LABEL);
    }
400
    /**
     * Returns position of the node.
     *
     * @param node
     *            pointer to the C node object
     * @param layerHeight
     *            the height of the bounding box
     *            (used to invert the y coordinate)
     * @return center point of <code>node</code>
410  */
    protected static Point getNodePos(final int node, final int layerHeight
     ) {
        Point pos = new Point(-1, -1);
        String posString;
        StringTokenizer st;

        posString = getAttribute(node, ATTR_POS);
        if (posString.length() > 0) {
            st = new StringTokenizer(posString, ",");
            pos = new Point(Integer.parseInt(st.nextToken()),
420                 layerHeight - Integer.parseInt(st.nextToken()));
        }

        return pos;
    }

    /**
     * Returns the position of the priority.
     *
     * @param edge
430  *            pointer to the C edge object
     * @param layerHeight
     *            the height of the bounding box
     *            (used to invert the y coordinate)
     * @return center point of edge's priority
     */
    protected static Point getPrioPos(final int edge, final int layerHeight
     ) {
        Point pos = new Point(-1, -1);
        String posString;
        StringTokenizer st;
440
        posString = getAttribute(edge, ATTR_TAILLP);
        if ((posString != null) && (posString.length() > 0)) {
            st = new StringTokenizer(posString, ",");
            pos = new Point(Integer.parseInt(st.nextToken()),
                    layerHeight - Integer.parseInt(st.nextToken()));
        } else {
            pos = new Point(0, 0);
        }

450     return pos;
    }

    /**
     * Returns the priority as string.
     *
     * @param edge
     *            pointer to the C edge object
     * @return the label string of the element
     */
460 protected static String getTailLabelString(final int edge) {
        return getAttribute(edge, ATTR_TAILLABEL);
    }
```

```
    /**
     * This is the native method to get the used graphviz version number.
     *
     * @return a string containing the graphviz version number
     */
    protected static native String getVersionString();
470
    /**
     * This is the native method to initialize the library.
     */
    protected static native void initialize();

    /**
     * This is the native method to invoke the neato cleanup routine.
     *
     * @param graph
480  *            pointer to the C graph object
     */
    protected static native void neatoCleanup(final int graph);

    /**
     * This is the native method to set a global edge attribute.
     *
     * @param graph
     *            pointer to the C graph object
     * @param attr
490  *            attribute's name (see constants)
     * @param value
     *            attribute's value
     */
    protected static native void setGlobalEdgeAttribute(final int graph,
            final String attr, final String value);

    /**
     * This is the native method to set a global node attribute.
     *
500  * @param graph
     *            pointer to the C graph object
     * @param attr
     *            attribute's name (see constants)
     * @param value
     *            attribute's value
     */
    protected static native void setGlobalNodeAttribute(final int graph,
            final String attr, final String value);

510 /**
     * This is the native method to set a graph attribute.
     *
     * @param graph
     *            pointer to the C graph object
     * @param attr
     *            attribute's name (see constants)
     * @param value
     *            attribute's value
     */
520 protected static native void setGraphAttribute(final int graph,
            final String attr, final String value);

    /**
     * This is the native method to set a local edge attribute.
     *
     * @param graph
     *            pointer to the C graph object
     * @param edge
     *            pointer to the C edge object
530  * @param attr
     *            attribute's name (see constants)
     * @param value
     *            attribute's value
     */
    protected static native void setLocalEdgeAttribute(final int graph,
            final int edge, final String attr, final String value);

    /**
     * This is the native method to set a local node attribute.
540  *
     * @param graph
     *            pointer to the C graph object
     * @param node
     *            pointer to the C node object
     * @param attr
     *            attribute's name (see constants)
     * @param value
     *            attribute's value
     */
550 protected static native void setLocalNodeAttribute(final int graph,
            final int node, final String attr, final String value);

    /**
     * This is the native method to invoke the twopi cleanup routine.
     *
     * @param graph
     *            pointer to the C graph object
     */
    protected static native void twopiCleanup(final int graph);
560
    /**
     * This is the native method to write graph into dot file.
     *
     * @param graph
     *            pointer to the C graph object
     * @param filename
     *            the name of the dot file
     */
    protected static native void writeDOT(final int graph,
570         final String filename);


    /**
     * This is the costructor.
     */
    private GraphvizAPI() {
    }
}
```

C.2.14. kiel.layouter.graphviz.CircoLibLayouter

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/11 18:18:28 $
 */
package kiel.layouter.graphviz;


/**
 * <p>
 * Description: This layouter uses the GraphViz circo algorithm via the
 * graphviz library to layout a given layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.5 $
 */
public final class CircoLibLayouter extends GraphvizLibraryLayouter {
    /**
     * @see GraphvizLibraryLayouter#doCleanup(int)
     */
    protected void doCleanup(final int graphPointer) {
        GraphvizAPI.circoCleanup(graphPointer);
    }

    /**
     * @see GraphvizLibraryLayouter#doLayout(int)
     */
    protected void doLayout(final int graphPointer) {
        GraphvizAPI.doCircoLayout(graphPointer);
    }
}
```

C.2.15. kiel.layouter.graphviz.DotLibLayouter

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/11 18:18:28 $
 */
package kiel.layouter.graphviz;


/**
 * <p>
 * Title: DotLayouter.java
 * </p>
 * <p>
 * Description: This layouter uses the GraphViz dot algorithm via the
 * graphviz library to layout a given layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.5 $
 */
public final class DotLibLayouter extends GraphvizLibraryLayouter {
    /**
     * @see GraphvizLibraryLayouter#doCleanup(int)
     */
    protected void doCleanup(final int graphPointer) {
        GraphvizAPI.dotCleanup(graphPointer);
    }

    /**
     * @see GraphvizLibraryLayouter#doLayout(int)
     */
    protected void doLayout(final int graphPointer) {
        GraphvizAPI.doDotLayout(graphPointer);
    }
}
```

C.2.16. kiel.layouter.graphviz.NeatoLibLayouter

```java
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/11 18:18:28 $
 */
package kiel.layouter.graphviz;

/**
 * <p>
 * Description: This layouter uses the GraphViz neato algorithm via the
 * graphviz library to layout a given layer.
 * </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Uni Kiel</p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss</a>
 * @version $Revision: 1.5 $
 */
public final class NeatoLibLayouter extends GraphvizLibraryLayouter {
    /**
     * @see GraphvizLibraryLayouter#doCleanup(int)
     */
    protected void doCleanup(final int graphPointer) {
        GraphvizAPI.neatoCleanup(graphPointer);
    }

    /**
     * @see GraphvizLibraryLayouter#doLayout(int)
     */
    protected void doLayout(final int graphPointer) {
        GraphvizAPI.doNeatoLayout(graphPointer);
    }
}
```

C.2.17. kiel.layouter.graphviz.TwopiLibLayouter

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/11 18:18:28 $
 */
package kiel.layouter.graphviz;

/**
 * <p>
 * Description: This layouter uses the GraphViz twopi algorithm via the
 * graphviz library to layout a given layer.
 * </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Uni Kiel</p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss</a>
 * @version $Revision: 1.5 $
 */
public final class TwopiLibLayouter extends GraphvizLibraryLayouter {
    /**
     * @see GraphvizLibraryLayouter#doCleanup(int)
     */
    protected void doCleanup(final int graphPointer) {
        GraphvizAPI.twopiCleanup(graphPointer);
    }

    /**
     * @see GraphvizLibraryLayouter#doLayout(int)
     */
    protected void doLayout(final int graphPointer) {
        GraphvizAPI.doTwopiLayout(graphPointer);
    }
}
```

C.2.18. kiel.layouter.hvlayouter.HVLayouter

```java
/*
 * $Author: tkl $
 *
 * $Date: 2005/04/11 18:18:28 $
 */
package kiel.layouter.hvlayouter;

import java.awt.Dimension;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import kiel.graphicalInformations.Point;
import kiel.graphicalInformations.Properties;
import kiel.graphicalInformations.View;
import kiel.layouter.LayerwiseLayouter;

/**
 * <p>
 * Description: This layouter defines levels to place states an transitions
 * onto.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.14 $
 */
public final class HVLayouter extends LayerwiseLayouter {
    /**
     * Compares all possible layers (depends on state order) with eachother
     * and takes the best layouted.
     *
     * @param nodes nodes in layer
     * @param view the view
     * @param horizontalLayout true, if layout should be done horizontal
     * @return the best layouted layer
     */
    private HVLayoutedLayer getBestLayoutedLayer(final Collection nodes,
            final View view, final boolean horizontalLayout) {
        PermutationIterator permIter = new PermutationIterator(nodes);
        HVLayoutedLayer bestLayer = null;
        ArrayList comparators = new ArrayList();

        comparators.add(new InitialStateComparator());
        comparators.add(new TransitionCrossingComparator());
        comparators.add(new NumberOfLevelsComparator());
        comparators.add(new WellBalancedLevelsComparator());

        while (permIter.hasMorePermutations()) {
            HVLayoutedLayer testedLayer;
            Iterator compIter;

            testedLayer = HVLayoutedLayer.getLayer(
                    (Collection) permIter.nextPermutation(),
horizontalLayout);

            if (bestLayer == null) {
                bestLayer = testedLayer;
            } else {
                compIter = comparators.iterator();
                while (compIter.hasNext()) {
                    int compResult = ((PropertyComparator) compIter.next())
                                          .compare(bestLayer, testedLayer);

                    if (compResult == -1) {
                        // better layouted layer found
                        bestLayer = testedLayer;
                        break;
                    } else if (compResult == 1) {
                        // worse layout
                        break;
                    }
                }
            }
        }

        return bestLayer;
    }

    /**
     * @see LayerwiseLayouter#layoutLayer(Collection, View, boolean)
     */
    protected Dimension layoutLayer(final Collection subnodes,
            final View view, final boolean horizontalLayout) {
        HVLayoutedLayer layer;

        layer = getBestLayoutedLayer(subnodes, view, horizontalLayout);
        layer.calculateRelativeLevelPositions(view);
        layer.placeAt(new Point(Properties.getLeftOffset(),
                                Properties.getUpperOffset()), view);
        layer.calculateDimension(view);

        return layer.getDimension();
    }
}
```

C.2.19. kiel.layouter.hvlayouter.HVLayoutedLayer

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

import java.awt.Dimension;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;

import kiel.dataStructure.Edge;
import kiel.dataStructure.Node;
import kiel.graphicalInformations.Point;
import kiel.graphicalInformations.View;


/**
 * <p>
 * Description: The HVLayouter seperates each statechart into different
 *     layers.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.9 $
 */
public abstract class HVLayoutedLayer {
    /**
     * Returns an instance of a layer depending on
     * <code>horizontalLayout</code>.
     *
     * @param nodes nodes in layer
     * @param horizontalLayout true, if layout should be done horizontal
     * @return a layer
     */
    protected static HVLayoutedLayer getLayer(final Collection nodes,
            final boolean horizontalLayout) {
        HVLayoutedLayer layer;

        if (horizontalLayout) {
            layer = new HorizontalLayoutedLayer(nodes);
        } else {
            layer = new VerticalLayoutedLayer(nodes);
        }

        return layer;
    }

    /** This is the dimension of this layer. */
    private Dimension dimension;

    /** This are the incoming and outgoing transitions of each state. */
    private ArrayList edges;

    /**
     * This is the lookup table to find the position on the chart of each
     * level.
     */
    private HashMap levelPositions = null;

    /** This is the lookup table to find the level for each level number.
     */
    private HashMap levels = null;

    /** This are the states in this layer. */
    private ArrayList nodes;

    /**
     * Creates a new instance of HVLayoutedLayer containing the given
     states.
     *
     * @param nodesInLayer
     *            Collection of type HVLayotedState
     */
    protected HVLayoutedLayer(final Collection nodesInLayer) {
        Iterator nodeIter;

        // copy all elements in state ArrayList
        nodes = new ArrayList();
        edges = new ArrayList();

        nodeIter = nodesInLayer.iterator();
        while (nodeIter.hasNext()) {
            Node node = (Node) nodeIter.next();
            HVLayoutedNode hvNode = new HVLayoutedNode(node, this);
            Iterator edgeIter;

            nodes.add(hvNode);
            ConversionLookupTable.getInstance().put(node, hvNode);
            edgeIter = node.getOutgoingTransitions().iterator();
            while (edgeIter.hasNext()) {
                Edge edge = (Edge) edgeIter.next();
                HVLayoutedEdge hvEdge = new HVLayoutedEdge(edge);

                edges.add(hvEdge);
                ConversionLookupTable.getInstance().put(edge, hvEdge);
            }
        }

        determineLevels();

        dimension = new Dimension();
    }
```

```
110
        /**
         * Places the transitions of one type (up/down) on levels.
         *
         * @param transitions
         *            transitions of one type (up/down)
         */
        private void assignLevels(final Collection transitions) {
            Iterator transitionIterator = transitions.iterator();
            HVLayoutedEdge transition;
120         Iterator levelIterator;
            TransitionLevel level;
            Integer key;
            Integer minKey = new Integer(0);
            Integer maxKey = new Integer(0);
            boolean transitionAdded;
            ArrayList keySet;

            // find closest level form state level for each transition
            while (transitionIterator.hasNext()) {
130             transition = (HVLayoutedEdge) transitionIterator.next();
                transitionAdded = false;
                keySet = new ArrayList(levels.keySet());
                Collections.sort(keySet);
                if (transition.isUpwardsTransition()) {
                    Collections.reverse(keySet);
                }
                levelIterator = keySet.iterator();

                while (levelIterator.hasNext()) {
140                 key = (Integer) levelIterator.next();
                    level = (TransitionLevel) levels.get(key);
                    // called for up or downwards transitions ?
                    // important for appending new levels
                    if (key.compareTo(minKey) < 0) {
                        minKey = key;
                    } else if (key.compareTo(maxKey) > 0) {
                        maxKey = key;
                    }
                    if ((key.intValue() > 0) && transition.
         isDownwardsTransition()
150                         && level.fits(transition)) {
                        level.addTransition(transition);
                        transitionAdded = true;
                        transition.setLevel(key);
                        break;
                    } else if ((key.intValue() < 0)
                            && transition.isUpwardsTransition()
                            && level.fits(transition)) {
                        level.addTransition(transition);
                        transitionAdded = true;
160                     transition.setLevel(key);
                        break;
                    }
                }
                if (!transitionAdded) {
                    // append new level
                    level = new TransitionLevel(nodes.size());
                    level.addTransition(transition);
                    if (transition.isDownwardsTransition()) {
170                     key = new Integer(maxKey.intValue() + 1);
                    } else {
                        key = new Integer(minKey.intValue() - 1);
                    }
                    transition.setLevel(key);
                    levels.put(key, level);
                }
            }
        }

180     /**
         * Calculates the minimum dimension of this layer.
         *
         * @param view
         *            current view
         */
        protected abstract void calculateDimension(final View view);

        /**
         * Calculates the relative level positions.
190      *
         * @param view
         *            current view
         */
        protected abstract void calculateRelativeLevelPositions(final View view
         );

        /**
         * Place all transitions on levels.
         */
        private void determineLevels() {
200         ArrayList downwardsTransitions = new ArrayList();
            ArrayList upwardsTransitions = new ArrayList();
            Iterator iter = nodes.iterator();
            HVLayoutedNode state;

            // clear level position
            levelPositions = null;

            // classify transitions
            while (iter.hasNext()) {
210             state = (HVLayoutedNode) iter.next();
                downwardsTransitions
                        .addAll(state.getOutgoingDownwardsTransitions());
                upwardsTransitions.addAll(state.getOutgoingUpwardsTransitions()
         );
            }

            // sort transitions by their length
            Collections.sort(downwardsTransitions);
            Collections.sort(upwardsTransitions);

220         levels = new HashMap();
            // assign levels for downwards transitions
            assignLevels(downwardsTransitions);

            // assign levels for upwards transitions
            assignLevels(upwardsTransitions);
```

```
                levels.put(new Integer(0), new StateLevel(nodes));
            }

230         /**
             * Returns all crossing transitions in couples.
             *
             * @return all crossing transitions in couples
             */
            protected final ArrayList getCrossingTransitions() {
                ArrayList crossings = new ArrayList();
                HVLayoutedEdge t1, t2;

                for (int i = 0; i < edges.size(); i++) {
240                 for (int j = i + 1; j < edges.size(); j++) {
                        t1 = (HVLayoutedEdge) edges.get(i);
                        t2 = (HVLayoutedEdge) edges.get(j);
                        if (t1.crosses(t2)) {
                            ArrayList v = new ArrayList();
                            v.add(t1);
                            v.add(t2);
                            crossings.add(v);
                        }
                    }
250             }

                return crossings;
            }

            /**
             * Returns the dimension of bounding box.
             *
             * @return Returns the dimension.
             */
260         protected final Dimension getDimension() {
                return dimension;
            }

            /**
             * Returns the downwards transitions of this layer.
             *
             * @return the downwards transitions of this layer
             */
            protected final ArrayList getDownwardsTransitionLevels() {
270             ArrayList downwardsLevels = new ArrayList();
                Iterator iter;
                Integer level;

                iter = levels.keySet().iterator();
                while (iter.hasNext()) {
                    level = (Integer) iter.next();
                    if (level.intValue() > 0) {
                        downwardsLevels.add(level);
                    }
280             }

                return downwardsLevels;
            }

            /**
             * Returns the level positions.
             *
             * @return Returns the levelPositions.
             */
290         protected final HashMap getLevelPositions() {
                return levelPositions;
            }

            /**
             * Returns the levels.
             *
             * @return Returns the levels.
             */
            protected final HashMap getLevels() {
300             return levels;
            }

            /**
             * Returns the nodes in this layer.
             *
             * @return returns the nodes.
             */
            protected final ArrayList getNodes() {
                return nodes;
310         }

            /**
             * Returns the number of downwards transitions.
             *
             * @return the number of downwards transitions
             */
            protected final int getNumberOfDownwardsTransitionLevels() {
                return getDownwardsTransitionLevels().size();
            }
320
            /**
             * Returns the number of transition crossings in this layer.
             *
             * @return the number of transition crossings in this layer
             */
            protected final int getNumberOfTransitionCrossings() {
                return getCrossingTransitions().size();
            }

330         /**
             * Returns the number of upwards transition.
             *
             * @return the number of upwards transitions
             */
            protected final int getNumberOfUpwardsTransitionLevels() {
                return getUpwardsTransitionLevels().size();
            }

            /**
340          * Returns the position of the given state in this layer. (From 1
             * (left/top) to n (right/bottom))
             *
             * @param state
             *            state which position should be found out
             * @return the state position
             */
```

```
        protected final int getStatePos(final HVLayoutedNode state) {
            return nodes.indexOf(state);
        }
350
        /**
         * Returns the upwards transitions of this layer.
         *
         * @return the upwards transitions of this layer
         */
        protected final ArrayList getUpwardsTransitionLevels() {
            ArrayList upwardsLevels = new ArrayList();
            Iterator iter;
            Integer level;
360
            iter = levels.keySet().iterator();
            while (iter.hasNext()) {
                level = (Integer) iter.next();
                if (level.intValue() < 0) {
                    upwardsLevels.add(level);
                }
            }

            return upwardsLevels;
370     }

        /**
         * Layoutes the given state. If the state has a sublayer, the sublayer
         * will be layouted first.
         *
         * @param state
         *            state to be layouted
         * @param view
         *            current view
380      */
        protected abstract void layoutState(final HVLayoutedNode state,
                final View view);

        /**
         * Places the states and the transitions on the chart.
         *
         * @param topLeft
         *            the top left point of the parants client area
         * @param view
390      *            current view
         */
        protected abstract void placeAt(final Point topLeft, final View view);

        /**
         * Set dimension of bounding box.
         *
         * @param d The dimension to set.
         */
        protected final void setDimension(final Dimension d) {
400         this.dimension = d;
        }

        /**
         * Set level positions.
         *
         * @param lp The levelPositions to set.
         */
        protected final void setLevelPositions(final HashMap lp) {
            this.levelPositions = lp;
410     }

        /**
         * Returns a string with the current state order.
         *
         * @return a string with the current state order
         */
        public final String toString() {
            return "States = " + nodes;
        }
420 }
```

C.2.20. kiel.layouter.hvlayouter.HorizontalLayoutedLayer

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/29 17:56:53 $
 */
package kiel.layouter.hvlayouter;

import java.awt.Dimension;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;

import kiel.dataStructure.Edge;
import kiel.dataStructure.Node;
import kiel.dataStructure.Transition;
import kiel.dataStructure.TransitionLabel;
import kiel.graphicalInformations.LabelLayoutInformation;
import kiel.graphicalInformations.LinetoPath;
import kiel.graphicalInformations.MovetoPath;
import kiel.graphicalInformations.NodeLayoutInformation;
import kiel.graphicalInformations.Point;
import kiel.graphicalInformations.Properties;
import kiel.graphicalInformations.View;
import kiel.layouter.LayouterProperties;


/**
 * <p>
 * Description: Specifies a horizontal layouted layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.13 $
 */
public final class HorizontalLayoutedLayer extends HVLayoutedLayer {
    /** This is the multiplier for height calculation. */
    private static final int STEPWIDTH_MULTIPLIER = 3;

    /**
     * Creates a new instance of HorizontalLayoutedLayer with the given
     * states.
     *
     * @param statesInLayer
     *            the states in this layer
     */
    protected HorizontalLayoutedLayer(final Collection statesInLayer) {
        super(statesInLayer);
    }

    /**
     * Calculates the dimension of this layer.
     *
     * @param view
     *            current view
     */
    protected void calculateDimension(final View view) {
        int width = 0;
        int height = 0;
        Iterator selfloopIterator;
        int maxSelfloopWidth = 0;
        HVLayoutedNode state;
        Iterator iter;
        ArrayList levelNumbers;
        Level level;

        levelNumbers = new ArrayList(getLevels().keySet());
        Collections.sort(levelNumbers);
        level = (Level) getLevels().get(levelNumbers.get(0));

        height = level.getMaxHeight(view)
                + ((Integer) getLevelPositions().get(
                        levelNumbers.get(0))).intValue();
        if (((Integer) levelNumbers.get(0)).intValue() < 0) {
            height += LayouterProperties.getLabelBorder();
        } else {
            height += LayouterProperties.getStateBorder();
        }

        iter = getNodes().iterator();
        while (iter.hasNext()) {
            state = (HVLayoutedNode) iter.next();
            NodeLayoutInformation nodeLayout = view
                    .getLayoutInformation(state.getState());

            width += nodeLayout.getWidth();

            selfloopIterator = state.getSelfloops().iterator();
            while (selfloopIterator.hasNext()) {
                Edge edge = ((HVLayoutedEdge) selfloopIterator.next())
                        .getTransition();
                int labelWidth;

                if (edge instanceof Transition) {
                    TransitionLabel label = ((Transition) edge).getLabel();
                    LabelLayoutInformation labelLayout = view
                            .getLayoutInformation(label);

                    labelWidth = labelLayout.getWidth();
                } else {
                    labelWidth = 0;
                }

                maxSelfloopWidth = Math.max(maxSelfloopWidth, labelWidth
                        + LayouterProperties.getLabelBorder()
```

```
                    + LayouterProperties.getStateBorder());
            }
            width += maxSelfloopWidth;

            if (iter.hasNext()) {
                width += LayouterProperties
                        .getHorizontalStateStepWidth();
            }
        }

        width += Properties.getLeftOffset() + Properties.getRightOffset();
        height += Properties.getUpperOffset() + Properties.getLowerOffset()
  ;

        setDimension(new Dimension(width, height));
    }

    /**
     * @see HVLayoutedLayer#calculateRelativeLevelPositions(View)
     */
    protected void calculateRelativeLevelPositions(final View view) {
        ArrayList levelNumbers = new ArrayList(getLevels().keySet());
        Level level;
        int currentPosition = 0;

        Collections.sort(levelNumbers);
        setLevelPositions(new HashMap());
        for (int i = levelNumbers.size() - 1; i >= 0; i--) {
            level = (Level) getLevels().get(levelNumbers.get(i));

            if (((Integer) levelNumbers.get(i)).intValue() > 0) {
                currentPosition += level.getMaxHeight(view)
                        + LayouterProperties.getLabelBorder();
            } else if (((Integer) levelNumbers.get(i)).intValue() == 0) {
                currentPosition += LayouterProperties
                        .getStateBorder();
            } else if (((Integer) levelNumbers.get(i)).intValue() < 0) {
                currentPosition += LayouterProperties
                        .getLabelBorder();
            }

            getLevelPositions().put(levelNumbers.get(i),
                    new Integer(currentPosition));

            if (((Integer) levelNumbers.get(i)).intValue() > 0) {
                currentPosition += LayouterProperties
                        .getLabelBorder();
            } else if (((Integer) levelNumbers.get(i)).intValue() == 0) {
                currentPosition += level.getMaxHeight(view)
                        + LayouterProperties.getStateBorder();
            } else if (((Integer) levelNumbers.get(i)).intValue() < 0) {
                currentPosition += level.getMaxHeight(view)
                        + LayouterProperties.getLabelBorder();
            }
        }
    }

    /**
     * Layouts a single state. If the state has child states these states
     are
     * layouted first.
     *
     * @param state
     *            the state to be layouted
     * @param view
     *            curretn view
     */
    protected void layoutState(final HVLayoutedNode state,
            final View view) {
        Node nativeState = state.getState();
        NodeLayoutInformation nodeLayout = view
                .getLayoutInformation(nativeState);

        // make sure state has minimum dimension
        nodeLayout.setHeight(Math.max(nodeLayout.getHeight(),
                state.getNumberOfSelfloops()
                * STEPWIDTH_MULTIPLIER
                * LayouterProperties
                        .getVerticalTransitionStepWidth()));
        nodeLayout.setWidth(Math.max(nodeLayout.getWidth(),
                Math.max(state.getNumberOfDownwardsTransitions(),
                        state.getNumberOfUpwardsTransitions())
                * LayouterProperties
                    .getHorizontalTransitionStepWidth()));
    }

    /**
     * Place states, transitions and labels at the chart.
     *
     * @param topLeft
     *            offset from the top left corner of the root chart
     * @param view
     *            current view
     */
    protected void placeAt(final Point topLeft, final View view) {
        Iterator stateIterator;
        HVLayoutedNode state;
        Iterator transitionIterator;
        HVLayoutedLayer sublayer;
        int top;
        int left;
        NodeLayoutInformation nodeLayout;

        top = topLeft.getY()
                + ((Integer) getLevelPositions().get(new Integer(0)))
                        .intValue();
        left = topLeft.getX();
        stateIterator = getNodes().iterator();

        while (stateIterator.hasNext()) {
            // place state
            int maxSelfloopWidth = 0;

            state = (HVLayoutedNode) stateIterator.next();
            layoutState(state, view);
            transitionIterator = state.getSelfloops().iterator();

            while (transitionIterator.hasNext()) {
                int selfloopWidth;
                Edge edge = ((HVLayoutedEdge) transitionIterator.next())
```

```
230                 .getTransition();

            if (edge instanceof Transition) {
                TransitionLabel label = ((Transition) edge).getLabel();
                LabelLayoutInformation labelLayout = view
                        .getLayoutInformation(label);

                labelLayout.setDimension(label.toString());
                selfloopWidth = labelLayout.getWidth()
                        + LayouterProperties.getLabelBorder();
240         } else {
                selfloopWidth = 0;
            }

            selfloopWidth += LayouterProperties.getStateBorder();

            maxSelfloopWidth = Math.max(maxSelfloopWidth, selfloopWidth
);
        }

        nodeLayout = view.getLayoutInformation(state.getState());
250     if (LayouterProperties.getStateAlignment()
                .compareTo("top/left") == 0) {
            nodeLayout.setUpperLeftPoint(new Point(left, top));
        } else if (LayouterProperties.getStateAlignment()
                .compareTo("bottom/right") == 0) {
            nodeLayout
                    .setUpperLeftPoint(new Point(left, top
                            + (((Level) getLevels().get(new Integer(0))
)
                                    .getMaxHeight(view) - nodeLayout
                                    .getHeight())));
260     } else {
            nodeLayout.setUpperLeftPoint(new Point(left, top
                    + (((Level) getLevels().get(new Integer(0)))
                            .getMaxHeight(view) - nodeLayout.getHeight
())
                    / 2));
        }

        left += maxSelfloopWidth;

        //place transitions
270     placeSelfloops(state, view);
        placeOutgoingDownwardsTransitions(state, topLeft, view);
        placeIncomingDownwardsTransitions(state, topLeft, view);
        placeOutgoingUpwardsTransitions(state, topLeft, view);
        placeIncomingUpwardsTransitions(state, topLeft, view);

        left += nodeLayout.getWidth()
             + LayouterProperties.getHorizontalStateStepWidth();
    }
}
280
/**
 * Places incoming downwards transitions.
 *
 * @param state
 *            the transitions of this state will be placed
 * @param topLeft
 *            offset from the top left corner of the root chart
 * @param view
 *            current view
290  */
private void placeIncomingDownwardsTransitions(
        final HVLayoutedNode state, final Point topLeft, final View
 view) {
    Iterator transitionIterator;
    ArrayList transitions;
    HVLayoutedEdge transition;
    int portPosition;
    NodeLayoutInformation nodeLayout = view.getLayoutInformation(state
            .getState());

300 portPosition = nodeLayout.getUpperLeftPoint().getX()
            + LayouterProperties
                    .getHorizontalTransitionStepWidth() / 2;
    transitions = state.getIncomingDownwardsTransitions();
    Collections.sort(transitions);
    transitionIterator = transitions.iterator();

    while (transitionIterator.hasNext()) {
        transition = (HVLayoutedEdge) transitionIterator.next();
        transition.setSinkPort(new Point(portPosition, nodeLayout
310             .getUpperLeftPoint().getY()));
        if (transition.getSourcePort() != null) {
            placeTransition(transition, topLeft, view);

            if (transition.getTransition() instanceof Transition) {
                LabelLayoutInformation labelLayout = view
                        .getLayoutInformation(((Transition) transition
                                .getTransition()).getLabel());

                // place label
320             labelLayout.setUpperLeftPoint(new Point(
                        transition.getSourcePort().getX()
                                + (transition.getSinkPort().getX()
                                        - transition.getSourcePort()
                                                .getX() - labelLayout
                                        .getWidth()) / 2,
                        ((Integer) getLevelPositions().get(
                                transition.getLevel())).intValue()
                                + topLeft.getY()
                                - LayouterProperties
330                                     .getLabelBorder()
                                - labelLayout.getHeight()));

                // place priority
                labelLayout = view.getLayoutInformation(
                        ((Transition) transition.getTransition())
                        .getPriority());
                labelLayout.setUpperLeftPoint(transition.getSourcePort
 ());
            }
        }
340     portPosition += LayouterProperties
                .getHorizontalTransitionStepWidth();
    }
}
```

```java
/**
 * Places incoming upwards transitions.
 *
 * @param state
 *            the transitions of this state will be placed
 * @param topLeft
 *            offset from the top left corner of the root chart
 * @param view
 *            current view
 */
private void placeIncomingUpwardsTransitions(final HVLayoutedNode state
 ,
        final Point topLeft, final View view) {
    Iterator transitionIterator;
    ArrayList transitions;
    HVLayoutedEdge transition;
    int portPosition;
    NodeLayoutInformation nodeLayout = view.getLayoutInformation(state
            .getState());

    portPosition = nodeLayout.getUpperLeftPoint().getX()
            + nodeLayout.getWidth()
            - LayouterProperties
                    .getHorizontalTransitionStepWidth() / 2;
    transitions = state.getIncomingUpwardsTransitions();
    Collections.sort(transitions);
    transitionIterator = transitions.iterator();

    while (transitionIterator.hasNext()) {
        transition = (HVLayoutedEdge) transitionIterator.next();
        transition.setSinkPort(new Point(portPosition, nodeLayout
                .getUpperLeftPoint().getY()
                + nodeLayout.getHeight()));
        if (transition.getSourcePort() != null) {
            placeTransition(transition, topLeft, view);

            // place label
            if (transition.getTransition() instanceof Transition) {
                LabelLayoutInformation labelLayout = view
                        .getLayoutInformation(((Transition) transition
                                .getTransition()).getLabel());

                labelLayout.setUpperLeftPoint(new Point(
                        transition.getSourcePort().getX()
                                + (transition.getSinkPort().getX()
                                        - transition.getSourcePort()
                                                .getX() - labelLayout
                                        .getWidth()) / 2,
                        ((Integer) getLevelPositions().get(
                                transition.getLevel())).intValue()
                                + topLeft.getY()
                                + LayouterProperties
                                        .getLabelBorder()));

                // place priority
                labelLayout = view.getLayoutInformation(
                        ((Transition) transition.getTransition())
                        .getPriority());
                labelLayout.setUpperLeftPoint(transition.getSourcePort
 ());
            }
        }

        portPosition -= LayouterProperties
                .getHorizontalTransitionStepWidth();
    }
}

/**
 * Places outgoing downwards transitions.
 *
 * @param state
 *            the transitions of this state will be placed
 * @param topLeft
 *            offset from the top left corner of the root chart
 * @param view
 *            current view
 */
private void placeOutgoingDownwardsTransitions(
        final HVLayoutedNode state, final Point topLeft, final View
 view) {
    Iterator transitionIterator;
    ArrayList transitions;
    HVLayoutedEdge transition;
    int portPosition;
    NodeLayoutInformation nodeLayout = view.getLayoutInformation(state
            .getState());

    portPosition = nodeLayout.getUpperLeftPoint().getX()
            + nodeLayout.getWidth()
            - LayouterProperties
                    .getHorizontalTransitionStepWidth() / 2;
    transitions = state.getOutgoingDownwardsTransitions();
    Collections.sort(transitions);
    transitionIterator = transitions.iterator();

    while (transitionIterator.hasNext()) {
        // place outgoing downwards transitions
        transition = (HVLayoutedEdge) transitionIterator.next();
        transition.setSourcePort(new Point(portPosition, nodeLayout
                .getUpperLeftPoint().getY()));
        if (transition.getSinkPort() != null) {
            placeTransition(transition, topLeft, view);
            if (transition.getTransition() instanceof Transition) {
                LabelLayoutInformation labelLayout = view
                        .getLayoutInformation(((Transition) transition
                                .getTransition()).getLabel());
                labelLayout.setUpperLeftPoint(new Point(
                        transition.getSourcePort().getX()
                                + (transition.getSinkPort().getX()
                                        - transition.getSourcePort()
                                                .getX() - labelLayout
                                        .getWidth()) / 2,
                        ((Integer) getLevelPositions().get(
                                transition.getLevel())).intValue()
                                + topLeft.getY()
                                - LayouterProperties
                                        .getLabelBorder()
                                - labelLayout.getHeight()));
```

```
                    // place priority
                    labelLayout = view.getLayoutInformation(
                            ((Transition) transition.getTransition())
                            .getPriority());
                    labelLayout.setUpperLeftPoint(transition.getSourcePort
 ());
                }
            }
            portPosition -= LayouterProperties
470                 .getHorizontalTransitionStepWidth();
        }
    }

    /**
     * Places outgoing upwards transitions.
     *
     * @param state
     *            the transitions of this state will be placed
     * @param topLeft
480  *            offset from the top left corner of the root chart
     * @param view
     *            current view
     */
    private void placeOutgoingUpwardsTransitions(final HVLayoutedNode state
     ,
            final Point topLeft, final View view) {
        Iterator transitionIterator;
        ArrayList transitions;
        HVLayoutedEdge transition;
        int portPosition;
490     NodeLayoutInformation nodeLayout = view.getLayoutInformation(state
                .getState());

        portPosition = nodeLayout.getUpperLeftPoint().getX()
                + LayouterProperties
                        .getHorizontalTransitionStepWidth() / 2;
        transitions = state.getOutgoingUpwardsTransitions();
        Collections.sort(transitions);
        transitionIterator = transitions.iterator();

500     while (transitionIterator.hasNext()) {
            transition = (HVLayoutedEdge) transitionIterator.next();
            transition.setSourcePort(new Point(portPosition, nodeLayout
                    .getUpperLeftPoint().getY()
                    + nodeLayout.getHeight()));
            if (transition.getSinkPort() != null) {
                placeTransition(transition, topLeft, view);

                if (transition.getTransition() instanceof Transition) {
                    LabelLayoutInformation labelLayout = view
510                         .getLayoutInformation(((Transition) transition
                                    .getTransition()).getLabel());

                    labelLayout.setUpperLeftPoint(new Point(
                            transition.getSourcePort().getX()
                                    + (transition.getSinkPort().getX()
                                            - transition.getSourcePort()
                                                    .getX() - labelLayout
                                            .getWidth()) / 2,
                            ((Integer) getLevelPositions().get(
520                                 transition.getLevel())).intValue()
                                    + topLeft.getY()
                                    + LayouterProperties
                                            .getLabelBorder()));

                    // place priority
                    labelLayout = view.getLayoutInformation(
                            ((Transition) transition.getTransition())
                            .getPriority());
                    labelLayout.setUpperLeftPoint(transition.getSourcePort
 ());
530             }
            }

            portPosition += LayouterProperties
                    .getHorizontalTransitionStepWidth();
        }
    }

    /**
     * Places selfloops and labels.
540  *
     * @param state
     *            the transitions of this state will be placed
     * @param view
     *            current view
     */
    private void placeSelfloops(final HVLayoutedNode state, final View view
     ) {
        Iterator transitionIterator;
        int portPosition;
        NodeLayoutInformation nodeLayout = view.getLayoutInformation(state
550             .getState());
        LabelLayoutInformation labelLayout;
        Point sourcePort;

        portPosition = nodeLayout.getUpperLeftPoint().getY()
                + LayouterProperties
                        .getVerticalTransitionStepWidth() / 2;
        transitionIterator = state.getSelfloops().iterator();

        while (transitionIterator.hasNext()) {
560         // place selfloop
            HVLayoutedEdge transition =
                (HVLayoutedEdge) transitionIterator.next();
            ArrayList path = new ArrayList();

            sourcePort = new Point(nodeLayout.getUpperLeftPoint().getX()
                    + nodeLayout.getWidth(), portPosition);
            path.add(new MovetoPath(sourcePort));
            path.add(new LinetoPath(new Point(nodeLayout.getUpperLeftPoint
 ()
                    .getX()  + nodeLayout.getWidth()
570                 + LayouterProperties.getStateBorder(),
                    portPosition)));
            portPosition += 2 * LayouterProperties
                    .getVerticalTransitionStepWidth();
            path.add(new LinetoPath(new Point(nodeLayout.getUpperLeftPoint
 ()
                    .getX() + nodeLayout.getWidth()
```

```
                    + LayouterProperties.getStateBorder(),
                    portPosition)));
            path.add(new LinetoPath(new Point(nodeLayout.getUpperLeftPoint
    ()
                    .getX() + nodeLayout.getWidth(), portPosition)));
580         view.getLayoutInformation(transition.getTransition()).setPath(
                    path);

            // place label
            if (transition.getTransition() instanceof Transition) {
                labelLayout = view
                        .getLayoutInformation(((Transition) transition
                                .getTransition()).getLabel());

                labelLayout.setUpperLeftPoint(new Point(nodeLayout
590                     .getUpperLeftPoint().getX() + nodeLayout.getWidth()
                        + LayouterProperties.getStateBorder()
                        + LayouterProperties.getLabelBorder(),
                        portPosition - LayouterProperties
                                .getVerticalTransitionStepWidth()
                        - labelLayout.getHeight() / 2));

                portPosition += LayouterProperties
                                    .getVerticalTransitionStepWidth();

600             // place priority
                labelLayout = view.getLayoutInformation(
                        ((Transition) transition.getTransition())
                        .getPriority());
              labelLayout.setUpperLeftPoint(sourcePort);
            }
        }
    }

    /**
610  * Place a single transition.
     *
     * @param transition
     *            the transition
     * @param offset
     *            the offset from the top left corner
     * @param view
     *            curretn view
     */
    protected void placeTransition(final HVLayoutedEdge transition,
620         final Point offset, final View view) {
        ArrayList path = new ArrayList();

        path.add(new MovetoPath(transition.getSourcePort()));
        path.add(new LinetoPath(new Point(transition.getSourcePort().getX()
    ,
                ((Integer) getLevelPositions().get(transition.getLevel()))
                        .intValue()
                        + offset.getY())));
        path.add(new LinetoPath(new Point(transition.getSinkPort().getX(),
                ((Integer) getLevelPositions().get(transition.getLevel()))
630                     .intValue()
                        + offset.getY())));
        path.add(new LinetoPath(transition.getSinkPort()));

        view.getLayoutInformation(transition.getTransition()).setPath(path)
    ;
    }
}
```

C.2.21. kiel.layouter.hvlayouter.VerticalLayoutedLayer

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/29 17:56:53 $
 */
package kiel.layouter.hvlayouter;

import java.awt.Dimension;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;

import kiel.dataStructure.Edge;
import kiel.dataStructure.Node;
import kiel.dataStructure.Transition;
import kiel.graphicalInformations.EdgeLayoutInformation;
import kiel.graphicalInformations.LabelLayoutInformation;
import kiel.graphicalInformations.LinetoPath;
import kiel.graphicalInformations.MovetoPath;
import kiel.graphicalInformations.NodeLayoutInformation;
import kiel.graphicalInformations.Point;
import kiel.graphicalInformations.Properties;
import kiel.graphicalInformations.View;
import kiel.layouter.LayouterProperties;


/**
 * <p>
 * Description: Specifies a vertical layouted layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.12 $
 */
public final class VerticalLayoutedLayer extends HVLayoutedLayer {
    /** This the multiplier for width calculation. */
    private static final int STEPWIDTH_MULTIPLIER = 3;

    /**
     * Creates a new instance of VerticalLayoutedLayer.
     *
     * @param statesInLayer
     *            the states in this layer
     */
    protected VerticalLayoutedLayer(final Collection statesInLayer) {
        super(statesInLayer);
    }

    /**
     * Calculates the dimension of this layer.
     *
     * @param view
     *            current view
     */
    protected void calculateDimension(final View view) {
        int width;
        int height;
        ArrayList levelNumbers = new ArrayList(getLevels().keySet());
        Level level;
        Iterator iter;
        HVLayoutedNode state;
        Iterator selfloopIterator;
        int maxSelfloopHeight = 0;

        Collections.sort(levelNumbers);
        level = (Level) getLevels().get(
                levelNumbers.get(levelNumbers.size() - 1));

        width = level.getMaxWidth(view)
                + ((Integer) getLevelPositions().get(
                        levelNumbers.get(levelNumbers.size() - 1)))
                        .intValue();
        if (((Integer) levelNumbers.get(levelNumbers.size() - 1))
                .intValue() > 0) {
            width += LayouterProperties.getLabelBorder();
        } else {
            width += LayouterProperties.getStateBorder();
        }

        iter = getNodes().iterator();
        height = 0;
        while (iter.hasNext()) {
            state = (HVLayoutedNode) iter.next();
            NodeLayoutInformation nodeLayout = view
                    .getLayoutInformation(state.getState());

            height += nodeLayout.getHeight();

            selfloopIterator = state.getSelfloops().iterator();
            while (selfloopIterator.hasNext()) {
                int labelHeight;
                Edge edge = ((HVLayoutedEdge) selfloopIterator.next())
                        .getTransition();

                if (edge instanceof Transition) {
                    LabelLayoutInformation labelLayout = view
                            .getLayoutInformation(((Transition) edge)
                                    .getLabel());

                    labelHeight = labelLayout.getHeight()
                            + LayouterProperties.getLabelBorder();
                } else {
                    labelHeight = 0;
                }
```

```
                labelHeight += LayouterProperties.getStateBorder();

                maxSelfloopHeight = Math.max(maxSelfloopHeight, labelHeight
 );
            }

            height += maxSelfloopHeight;
120
            if (iter.hasNext()) {
                height += LayouterProperties
                        .getVerticalStateStepWidth();
            }
        }

        width += Properties.getLeftOffset() + Properties.getRightOffset();
        height += Properties.getUpperOffset() + Properties.getLowerOffset()
 ;

130     setDimension(new Dimension(width, height));
    }

    /**
     * @see HVLayoutedLayer#calculateRelativeLevelPositions(View)
     */
    protected void calculateRelativeLevelPositions(final View view) {
        ArrayList levelNumbers = new ArrayList(getLevels().keySet());
        Level level;
        int currentPosition = 0;
140
        Collections.sort(levelNumbers);
        setLevelPositions(new HashMap());
        for (int i = 0; i < levelNumbers.size(); i++) {
            level = (Level) getLevels().get(levelNumbers.get(i));

            if (((Integer) levelNumbers.get(i)).intValue() > 0) {
                currentPosition += LayouterProperties
                        .getLabelBorder();
            } else if (((Integer) levelNumbers.get(i)).intValue() == 0) {
150             currentPosition += LayouterProperties
                        .getStateBorder();
            } else if (((Integer) levelNumbers.get(i)).intValue() < 0) {
                currentPosition += level.getMaxWidth(view)
                        + LayouterProperties.getLabelBorder();
            }

            getLevelPositions().put(levelNumbers.get(i),
                    new Integer(currentPosition));

160         if (((Integer) levelNumbers.get(i)).intValue() > 0) {
                currentPosition += level.getMaxWidth(view)
                        + LayouterProperties.getLabelBorder();
            } else if (((Integer) levelNumbers.get(i)).intValue() == 0) {
                currentPosition += level.getMaxWidth(view)
                        + LayouterProperties.getStateBorder();
            } else if (((Integer) levelNumbers.get(i)).intValue() < 0) {
                currentPosition += LayouterProperties
                        .getLabelBorder();
            }
170     }
    }

    /**
     * Layouts the specified state and its possible existing child states.
     *
     * @param state
     *            the state to be layouted
     * @param view
     *            current view
180  */
    protected void layoutState(final HVLayoutedNode state,
            final View view) {
        Node nativeState = state.getState();
        NodeLayoutInformation nodeLayout = view.getLayoutInformation(state
                .getState());

        // make sure state has minimum dimension
        nodeLayout.setWidth(Math.max(nodeLayout.getWidth(),
                state.getNumberOfSelfloops()
190             * STEPWIDTH_MULTIPLIER
                * LayouterProperties
                        .getHorizontalTransitionStepWidth()));
        nodeLayout.setHeight(Math.max(nodeLayout.getHeight(),
                Math.max(state.getNumberOfDownwardsTransitions(),
                        state.getNumberOfUpwardsTransitions())
                * LayouterProperties
                    .getVerticalTransitionStepWidth()));
    }

200 /**
     * Places the states, transitions and labels in thsi layer beginning at
     * topLeft.
     *
     * @param topLeft
     *            the top left offset of this layer
     * @param view
     *            current view
     */
    protected void placeAt(final Point topLeft, final View view) {
210     Iterator stateIterator = getNodes().iterator();
        Iterator transitionIterator;
        int top;
        int left;
        HVLayoutedLayer sublayer;

        top = topLeft.getY();
        left = topLeft.getX()
                + ((Integer) getLevelPositions().get(new Integer(0)))
                        .intValue();
220
        while (stateIterator.hasNext()) {
            int maxSelfloopHeight = 0;
            HVLayoutedNode state;
            NodeLayoutInformation nodeLayout;

            // place state
            state = (HVLayoutedNode) stateIterator.next();
            layoutState(state, view);

230         transitionIterator = state.getSelfloops().iterator();
```

```
            while (transitionIterator.hasNext()) {
                int labelHeight;
                HVLayoutedEdge transition = (HVLayoutedEdge)
 transitionIterator
                        .next();

                if (transition.getTransition() instanceof Transition) {
                    LabelLayoutInformation labelLayout = view
                            .getLayoutInformation(((Transition) transition
                                    .getTransition()).getLabel());

                    labelLayout.setDimension(((Transition) transition
                            .getTransition()).getLabel().toString());
                    labelHeight = labelLayout.getHeight();
                } else {
                    labelHeight = 0;
                }

                maxSelfloopHeight = Math.max(maxSelfloopHeight, labelHeight
                        + LayouterProperties.getLabelBorder()
                        + LayouterProperties.getStateBorder());
            }
            top += maxSelfloopHeight;

            nodeLayout = view.getLayoutInformation(state.getState());
            if (LayouterProperties.getStateAlignment()
                    .compareTo("top/left") == 0) {
                nodeLayout.setUpperLeftPoint(new Point(left, top));
            } else if (LayouterProperties.getStateAlignment()
                    .compareTo("bottom/right") == 0) {
                nodeLayout.setUpperLeftPoint(new Point(left
                        + (((Level) getLevels().get(new Integer(0)))
                                .getMaxWidth(view) - nodeLayout.getWidth())
 ,
                        top));
            } else {
                nodeLayout.setUpperLeftPoint(new Point(left
                        + (((Level) getLevels().get(new Integer(0)))
                                .getMaxWidth(view) - nodeLayout.getWidth())
                        / 2, top));
            }

            // place transitions
            placeSelfloops(state, view);
            placeOutgoingDownwardsTransitions(state, topLeft, view);
            placeIncomingDownwardsTransitions(state, topLeft, view);
            placeOutgoingUpwardsTransitions(state, topLeft, view);
            placeIncomingUpwardsTransitions(state, topLeft, view);

            top += nodeLayout.getHeight()
                + LayouterProperties.getHorizontalStateStepWidth();
        }
    }

    /**
     * Places incoming downwards transitions.
     *
     * @param state
     *            the transitions of this state will be placed
     * @param topLeft
     *            offset from the top left corner of the root chart
     * @param view
     *            current view
     */
    private void placeIncomingDownwardsTransitions(
            final HVLayoutedNode state, final Point topLeft, final View
 view) {
        Iterator transitionIterator;
        ArrayList transitions;
        HVLayoutedEdge transition;
        int portPosition;
        NodeLayoutInformation nodeLayout = view.getLayoutInformation(state
                .getState());

        portPosition = nodeLayout.getUpperLeftPoint().getY()
                + LayouterProperties
                        .getVerticalTransitionStepWidth() / 2;
        transitions = state.getIncomingDownwardsTransitions();
        Collections.sort(transitions);
        transitionIterator = transitions.iterator();
        while (transitionIterator.hasNext()) {
            transition = (HVLayoutedEdge) transitionIterator.next();
            transition.setSinkPort(new Point(nodeLayout.getUpperLeftPoint()
                    .getX()
                    + nodeLayout.getWidth(), portPosition));
            if (transition.getSourcePort() != null) {
                placeTransition(transition, topLeft, view);

                if (transition.getTransition() instanceof Transition) {
                    LabelLayoutInformation labelLayout = view
                            .getLayoutInformation(((Transition) transition
                                    .getTransition()).getLabel());

                    labelLayout.setUpperLeftPoint(new Point(
                            ((Integer) getLevelPositions().get(
                                    transition.getLevel())).intValue()
                                    + topLeft.getX()
                                    + LayouterProperties
                                            .getLabelBorder(), transition
                                    .getSourcePort().getY()
                                    + (transition.getSinkPort().getY()
                                            - transition.getSourcePort()
                                                    .getY() - labelLayout
                                            .getHeight()) / 2));

                    // place priority
                    labelLayout = view.getLayoutInformation(
                            ((Transition) transition.getTransition())
                            .getPriority());
                    labelLayout.setUpperLeftPoint(transition.getSourcePort
 ());
                }
            }
            portPosition += LayouterProperties
                    .getVerticalTransitionStepWidth();
        }
    }

    /**
     * Places incoming upwards transitions.
```

```
     *
     * @param state
     *            the transitions of this state will be placed
     * @param topLeft
     *            offset from the top left corner of the root chart
     * @param view
     *            current view
     */
    private void placeIncomingUpwardsTransitions(final HVLayoutedNode state
     ,
            final Point topLeft, final View view) {
        Iterator transitionIterator;
        ArrayList transitions;
        HVLayoutedEdge transition;
        int portPosition;
        NodeLayoutInformation nodeLayout = view.getLayoutInformation(state
                .getState());

        portPosition = nodeLayout.getUpperLeftPoint().getY()
                + nodeLayout.getHeight()
                - LayouterProperties
                        .getVerticalTransitionStepWidth() / 2;
        transitions = state.getIncomingUpwardsTransitions();
        Collections.sort(transitions);
        transitionIterator = transitions.iterator();
        while (transitionIterator.hasNext()) {
            transition = (HVLayoutedEdge) transitionIterator.next();
            transition.setSinkPort(new Point(nodeLayout.getUpperLeftPoint()
                    .getX(), portPosition));

            if (transition.getSourcePort() != null) {
                placeTransition(transition, topLeft, view);

                if (transition.getTransition() instanceof Transition) {
                    LabelLayoutInformation labelLayout = view
                            .getLayoutInformation(((Transition) transition
                                    .getTransition()).getLabel());

                    labelLayout.setUpperLeftPoint(new Point(
                            ((Integer) getLevelPositions().get(
                                    transition.getLevel())).intValue()
                                    + topLeft.getX()
                                    - LayouterProperties
                                            .getLabelBorder()
                                    - labelLayout.getWidth(), transition
                                    .getSourcePort().getY()
                                    + (transition.getSinkPort().getY()
                                            - transition.getSourcePort()
                                                    .getY() - labelLayout
                                            .getHeight()) / 2));

                    // place priority
                    labelLayout = view.getLayoutInformation(
                            ((Transition) transition.getTransition())
                            .getPriority());
                    labelLayout.setUpperLeftPoint(transition.getSourcePort
     ());
                }
            }
                portPosition -= LayouterProperties
                        .getVerticalTransitionStepWidth();
            }
        }

    /**
     * Places outgoing downwards transitions.
     *
     * @param state
     *            the transitions of this state will be placed
     * @param topLeft
     *            offset from the top left corner of the root chart
     * @param view
     *            current view
     */
    private void placeOutgoingDownwardsTransitions(
            final HVLayoutedNode state, final Point topLeft, final View
     view) {
        Iterator transitionIterator;
        ArrayList transitions;
        HVLayoutedEdge transition;
        int portPosition;
        NodeLayoutInformation nodeLayout = view.getLayoutInformation(state
                .getState());

        portPosition = nodeLayout.getUpperLeftPoint().getY()
                + nodeLayout.getHeight()
                - LayouterProperties
                        .getVerticalTransitionStepWidth() / 2;
        transitions = state.getOutgoingDownwardsTransitions();
        Collections.sort(transitions);
        transitionIterator = transitions.iterator();
        while (transitionIterator.hasNext()) {
            transition = (HVLayoutedEdge) transitionIterator.next();
            transition.setSourcePort(new Point(nodeLayout.getUpperLeftPoint
     ()
                    .getX()
                    + nodeLayout.getWidth(), portPosition));
            if (transition.getSinkPort() != null) {
                placeTransition(transition, topLeft, view);

                if (transition.getTransition() instanceof Transition) {
                    LabelLayoutInformation labelLayout = view
                            .getLayoutInformation(((Transition) transition
                                    .getTransition()).getLabel());

                    labelLayout.setUpperLeftPoint(new Point(
                            ((Integer) getLevelPositions().get(
                                    transition.getLevel())).intValue()
                                    + topLeft.getX()
                                    + LayouterProperties
                                            .getLabelBorder(), transition
                                    .getSourcePort().getY()
                                    + (transition.getSinkPort().getY()
                                            - transition.getSourcePort()
                                                    .getY() - labelLayout
                                            .getHeight()) / 2));

                    // place priority
                    labelLayout = view.getLayoutInformation(
```

```
                        ((Transition) transition.getTransition())
                        .getPriority());
                labelLayout.setUpperLeftPoint(transition.getSourcePort
 ());
            }
        }

        portPosition -= LayouterProperties
                .getVerticalTransitionStepWidth();
    }
}

/**
 * Places outgoing upwards transitions.
 *
 * @param state
 *            the transitions of this state will be placed
 * @param topLeft
 *            offset from the top left corner of the root chart
 * @param view
 *            current view
 */
private void placeOutgoingUpwardsTransitions(final HVLayoutedNode state
 ,
        final Point topLeft, final View view) {
    Iterator transitionIterator;
    ArrayList transitions;
    HVLayoutedEdge transition;
    int portPosition;
    NodeLayoutInformation nodeLayout = view.getLayoutInformation(state
            .getState());

    portPosition = nodeLayout.getUpperLeftPoint().getY()
            + LayouterProperties
                    .getVerticalTransitionStepWidth() / 2;
    transitions = state.getOutgoingUpwardsTransitions();
    Collections.sort(transitions);
    transitionIterator = transitions.iterator();
    while (transitionIterator.hasNext()) {
        transition = (HVLayoutedEdge) transitionIterator.next();
        transition.setSourcePort(new Point(nodeLayout.getUpperLeftPoint
 ()
                .getX(), portPosition));
        if (transition.getSinkPort() != null) {
            placeTransition(transition, topLeft, view);

            if (transition.getTransition() instanceof Transition) {
                LabelLayoutInformation labelLayout = view
                        .getLayoutInformation(((Transition) transition
                                .getTransition()).getLabel());

                labelLayout.setUpperLeftPoint(new Point(
                        ((Integer) getLevelPositions().get(
                                transition.getLevel())).intValue()
                                + topLeft.getX()
                                - LayouterProperties
                                        .getLabelBorder()
                                - labelLayout.getWidth(), transition
                                .getSourcePort().getY()
                                + (transition.getSinkPort().getY()
                                - transition.getSourcePort()
                                        .getY() - labelLayout
                                .getHeight()) / 2));

                // place priority
                labelLayout = view.getLayoutInformation(
                        ((Transition) transition.getTransition())
                        .getPriority());
                labelLayout.setUpperLeftPoint(transition.getSourcePort
 ());
}
            }

        portPosition += LayouterProperties
                .getVerticalTransitionStepWidth();
    }
}

/**
 * Places selfloops and labels.
 *
 * @param state
 *            the transitions of this state will be placed
 * @param view
 *            current view
 */
private void placeSelfloops(final HVLayoutedNode state, final View view
 ) {
    Iterator transitionIterator;
    int portPosition;
    NodeLayoutInformation nodeLayout = view.getLayoutInformation(state
            .getState());
    Point sourcePort;

    portPosition = nodeLayout.getUpperLeftPoint().getX()
            + LayouterProperties
                    .getHorizontalTransitionStepWidth() / 2;
    transitionIterator = state.getSelfloops().iterator();
    while (transitionIterator.hasNext()) {
        HVLayoutedEdge transition =
            (HVLayoutedEdge) transitionIterator.next();
        ArrayList path = new ArrayList();

        // place selfloop
        sourcePort = new Point(portPosition, nodeLayout
                .getUpperLeftPoint().getY());
        path.add(new MovetoPath(sourcePort));
        path.add(new LinetoPath(new Point(portPosition, nodeLayout
                .getUpperLeftPoint().getY()
                - LayouterProperties.getStateBorder())));
        portPosition += 2 * LayouterProperties
                .getHorizontalTransitionStepWidth();
        path.add(new LinetoPath(new Point(portPosition, nodeLayout
                .getUpperLeftPoint().getY()
                - LayouterProperties.getStateBorder())));
        path.add(new LinetoPath(new Point(portPosition, nodeLayout
                .getUpperLeftPoint().getY())));

        view.getLayoutInformation(transition.getTransition()).setPath(
 path);
```

```
            // place label
            if (transition.getTransition() instanceof Transition) {
580             LabelLayoutInformation labelLayout = view.
    getLayoutInformation(
                        ((Transition) transition.getTransition()).getLabel
    ());

                labelLayout.setUpperLeftPoint(new Point(portPosition
                        - LayouterProperties
                                .getHorizontalTransitionStepWidth()
                        - labelLayout.getWidth() / 2,
                        nodeLayout.getUpperLeftPoint().getY()
                        - LayouterProperties.getStateBorder()
                        - LayouterProperties.getLabelBorder()
590                     - labelLayout.getHeight()));

                // place priority
                labelLayout = view.getLayoutInformation(
                        ((Transition) transition.getTransition())
                        .getPriority());
                labelLayout.setUpperLeftPoint(sourcePort);
            }

            portPosition += LayouterProperties
600                 .getHorizontalTransitionStepWidth();
        }
    }

    /**
     * Places a transition in the layer.
     *
     * @param transition
     *            the transition to be placed
     * @param view
610  *            current view
     * @param offset
     *            the offset from the absolute top left corner
     */
    protected void placeTransition(final HVLayoutedEdge transition,
            final Point offset, final View view) {
        ArrayList path = new ArrayList();
        EdgeLayoutInformation edgeLayout = view
                .getLayoutInformation(transition.getTransition());

620     path.add(new MovetoPath(transition.getSourcePort()));
        path.add(new LinetoPath(new Point(((Integer) getLevelPositions().
    get(
                transition.getLevel())).intValue()
                + offset.getX(), transition.getSourcePort().getY())));
        path.add(new LinetoPath(new Point(((Integer) getLevelPositions().
    get(
                transition.getLevel())).intValue()
                + offset.getX(), transition.getSinkPort().getY())));
        path.add(new LinetoPath(transition.getSinkPort()));

        edgeLayout.setPath(path);
630 }
}
```

C.2.22. kiel.layouter.hvlayouter.ConversionLookupTable

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

import java.util.Hashtable;

/**
 * <p>
 * Description: This table is used as a lookup table to find for each KIEL
 * element HVLayoted one.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.7 $
 */
public final class ConversionLookupTable {
    /**
     * This is the instance of ConversionLookupTable.
     */
    private static ConversionLookupTable instance = new
     ConversionLookupTable();

    /**
     * Returns the instance.
     *
     * @return the instance
     */
    protected static ConversionLookupTable getInstance() {
        return instance;
    }

    /**
     * This is the primitive table.
     */
    private Hashtable table;

    /** Creates a new instance of ConversionLookupTable. */
    private ConversionLookupTable() {
        table = new Hashtable();
    }

    /**
     * Tests if some key maps into the specified value in this hashtable.
     * This operation is more expensive than the containsKey  method.
     *
     * @param key the primitive object
     * @return true if and only if some key maps to the value argument in
     this
     *         hashtable as determined by the equals method; false
     otherwise.
     */
    protected boolean contains(final Object key) {
        return table.containsKey(key);
    }

    /**
     * Returns the value to which the specified key is mapped in this
     hashtable.
     *
     * @param key the primitive object
     * @return the value to which the key is mapped in this hashtable;
     *         null if the key is not mapped to any value in this hashtable
     .
     */
    protected Object get(final Object key) {
        return table.get(key);
    }

    /**
     * Maps the specified key to the specified value in this hashtable.
     * Neither the key nor the value can be null.
     *
     * @param key the primitve object
     * @param value the hvlayouted object
     * @return the previous value of the specified key in this hashtable,
     *         or null if it did not have one.
     **/
    protected Object put(final Object key, final Object value) {
        return table.put(key, value);
    }
}
```

C.2.23. kiel.layouter.hvlayouter.HVLayoutedEdge

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

import kiel.dataStructure.Edge;
import kiel.graphicalInformations.Point;

/**
 * <p>
 * Description: This class makes some extension to the primitive transition
 *     .
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.8 $
 */
public final class HVLayoutedEdge implements Comparable {
    /** This is the primitive transition. */
    private Edge edge;

    /** This is the level number. */
    private Integer level;

    /** This is the sink state. */
    private HVLayoutedNode sink;

    /**
     * This is the point where this transition is attached to to sink state
     * .
     */
    private Point sinkPort;

    /** This is the source state. */
    private HVLayoutedNode source;

    /**
     * This is the point where this transition is attached to to source
     * state.
     */
    private Point sourcePort;

    /**
     * Creates a new instance of HVLayoutedTransition.
     *
     * @param anEdge the primitive transition
     */
    protected HVLayoutedEdge(final Edge anEdge) {
        edge = anEdge;
    }

    /**
     * Compares this transition and obj by their length.
     *
     * @param obj the transition this one should be compared with
     * @return the difference of their length
     */
    public int compareTo(final Object obj) {
        return getLength() - ((HVLayoutedEdge) obj).getLength();
    }

    /**
     * Returns true if this transition corsses transition.
     *
     * @param transition an other transtion
     * @return true, if both transition cosses at least once
     */
    protected boolean crosses(final HVLayoutedEdge transition) {
        boolean sameDirection =
            ((isDownwardsTransition() && transition.isDownwardsTransition()
 )
            || (isUpwardsTransition() && transition.isUpwardsTransition()))
 ;
        int mySourcePosition = getSource().getPositionInLayer();
        int mySinkPosition = getSink().getPositionInLayer();
        int itsSourcePosition = transition.getSource().getPositionInLayer()
 ;
        int itsSinkPosition = transition.getSink().getPositionInLayer();
        boolean transitionSurroundsSource =
            ((itsSourcePosition < mySourcePosition)
                && (mySourcePosition < itsSinkPosition))
            || ((itsSourcePosition > mySourcePosition)
                && (mySourcePosition > itsSinkPosition));
        boolean transitionSurroundsSink =
            ((itsSourcePosition < mySinkPosition)
                && (mySinkPosition < itsSinkPosition))
            || ((itsSourcePosition > mySinkPosition)
                && (mySinkPosition > itsSinkPosition));
        boolean sameSource = (getSource() == transition.getSource());
        boolean sameSink = (getSink() == transition.getSink());

        return sameDirection && !sameSource && !sameSink
            && (transitionSurroundsSource ^ transitionSurroundsSink);
    }

    /**
     * Returns the length of this transition, calculated by the state
     position
     * difference of source and sink state.
     *
     * @return the length
     */
    protected int getLength() {
        return Math.abs(getSource().getPositionInLayer()
```

```
                            - getSink().getPositionInLayer());
        }

        /**
110      * Returns the level number.
         *
         * @return the level number
         */
        protected Integer getLevel() {
            return level;
        }

        /**
         * Returns the sink state.
120      *
         * @return the sink state
         */
        protected HVLayoutedNode getSink() {
            if (sink == null) {
                sink = (HVLayoutedNode) ConversionLookupTable.getInstance().
                                         get(edge.getTarget());
                sink.getPositionInLayer();
            }

130         return sink;
        }

        /**
         * Returns the point at which the transition is attached to the sink
         state.
         *
         * @return the sink port
         */
        protected Point getSinkPort() {
            return sinkPort;
140     }

        /**
         * Returns the source state.
         *
         * @return the source state
         */
        protected HVLayoutedNode getSource() {
            if (source == null) {
                source = (HVLayoutedNode) ConversionLookupTable.getInstance().
150                                        get(edge.getSource());
                source.getPositionInLayer();
            }

            return source;
        }

        /**
         * Returns the point at which the transition is attached to the source
         * state.
160      *
         * @return the source port
         */
        protected Point getSourcePort() {
            return sourcePort;
        }

        /**
         * Returns the primitive transition.
         *
170      * @return the primitive transition
         */
        protected Edge getTransition() {
            return edge;
        }

        /**
         * Returns true, if this transition is a downwards transition.
         *
         * @return true, if downwards transition
180      */
        protected boolean isDownwardsTransition() {
            int sourcePosition = getSource().getPositionInLayer();
            int sinkPosition = getSink().getPositionInLayer();

            return (sourcePosition < sinkPosition);
        }

        /**
         * Returns true, if this transition is an upwards transition.
190      *
         * @return true, if upwards transition
         */
        protected boolean isUpwardsTransition() {
            int sourcePosition = getSource().getPositionInLayer();
            int sinkPosition = getSink().getPositionInLayer();

            return (sourcePosition > sinkPosition);
        }

200     /**
         * Sets the level number to aLevel.
         *
         * @param aLevel the level number
         */
        protected void setLevel(final Integer aLevel) {
            level = new Integer(aLevel.intValue());
        }

        /**
210      * Sets the point at which the transition is attached to the sink state
         .
         *
         * @param port the sink port
         */
        protected void setSinkPort(final Point port) {
            sinkPort = port;
        }

        /**
         * Sets the point at which the transition is attached to the source
         state.
220      *
         * @param port the source port
         */
```

```
    protected void setSourcePort(final Point port) {
        sourcePort = port;
    }

    /**
     * Returns "[source] -> [sink]".
     *
230  * @return "[source] -> [sink]"
     */
    public String toString() {
        return getSource() + "->" + getSink();
    }
}
```

C.2.24. kiel.layouter.hvlayouter.HVLayoutedNode

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

import java.util.ArrayList;
import java.util.Iterator;

import kiel.dataStructure.Node;


/**
 * <p>
 * Description: This class brings some extensions to the primitive state.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.10 $
 */
public final class HVLayoutedNode {
    /** This are the incomming downwards transitions of this state. */
    private ArrayList incomingDownwardsTransitions;

    /** This are the incoming upwards transitions of this state. */
    private ArrayList incomingUpwardsTransitions;

    /** This is the layer containing this state. */
    private HVLayoutedLayer layer;

    /** This is the primitve state. */
    private Node node;

    /** This are the outgoing downwards transitions of this state. */
    private ArrayList outgoingDownwardsTransitions;

    /** This are the outgoing upwards transitions of this state. */
    private ArrayList outgoingUpwardsTransitions;

    /** This are the selfloops of this state. */
    private ArrayList selfloops;

    /**
     * Creates a new instance of HVLayoutetdState.
     *
     * @param aNode
     *            the primitive state
     * @param aLayer
     *            the layer aNode is onto
     */
    protected HVLayoutedNode(final Node aNode, final HVLayoutedLayer aLayer
     ) {
        node = aNode;
        layer = aLayer;
    }

    /**
     * Classifies the transitions.
     */
    private void classifyTransitions() {
        Iterator iter;
        ConversionLookupTable lookup = ConversionLookupTable.getInstance();

        selfloops = new ArrayList();
        outgoingUpwardsTransitions = new ArrayList();
        incomingUpwardsTransitions = new ArrayList();
        outgoingDownwardsTransitions = new ArrayList();
        incomingDownwardsTransitions = new ArrayList();

        iter = node.getOutgoingTransitions().iterator();
        while (iter.hasNext()) {
            HVLayoutedEdge edge = (HVLayoutedEdge) lookup.get(iter.next());

            if (edge.isUpwardsTransition()) {
                outgoingUpwardsTransitions.add(edge);
            } else if (edge.isDownwardsTransition()) {
                outgoingDownwardsTransitions.add(edge);
            } else {
                selfloops.add(edge);
            }
        }

        iter = node.getIncomingTransitions().iterator();
        while (iter.hasNext()) {
            HVLayoutedEdge edge = (HVLayoutedEdge) lookup.get(iter.next());

            if (edge.isUpwardsTransition()) {
                incomingUpwardsTransitions.add(edge);
            } else if (edge.isDownwardsTransition()) {
                incomingDownwardsTransitions.add(edge);
            }
        }
    }

    /**
     * Returns the number of incoming downwards transitions.
     *
     * @return the number of incoming downwards transitions
     */
    protected ArrayList getIncomingDownwardsTransitions() {
        if (incomingDownwardsTransitions == null) {
            classifyTransitions();
        }

        return incomingDownwardsTransitions;
```

```
        }

        /**
         * Returns the number of incoming upwards transitions.
         *
         * @return the number of incoming upwards transitions
         */
        protected ArrayList getIncomingUpwardsTransitions() {
120         if (incomingUpwardsTransitions == null) {
                classifyTransitions();
            }

            return incomingUpwardsTransitions;
        }

        /**
         * Returns the number of downwards transitions.
         *
130      * @return the number of downwards transitions
         */
        protected int getNumberOfDownwardsTransitions() {
            return getIncomingDownwardsTransitions().size()
                    + getOutgoingDownwardsTransitions().size();
        }

        /**
         * Returns the number of selfloops.
         *
140      * @return the number of selfloops
         */
        protected int getNumberOfSelfloops() {
            return getSelfloops().size();
        }

        /**
         * Returns the number of upwards transitions.
         *
         * @return the number of upwards transitions
150      */
        protected int getNumberOfUpwardsTransitions() {
            return getIncomingUpwardsTransitions().size()
                    + getOutgoingUpwardsTransitions().size();
        }

        /**
         * Returns the number of outgoing downwards transitions.
         *
         * @return the number of outgoing downwards transitions
160      */
        protected ArrayList getOutgoingDownwardsTransitions() {
            if (outgoingDownwardsTransitions == null) {
                classifyTransitions();
            }

            return outgoingDownwardsTransitions;
        }

        /**
170      * Returns the number of outgoing upwards transitions.
         *
         * @return the number of outgoing upwards transitions
         */
        protected ArrayList getOutgoingUpwardsTransitions() {
            if (outgoingUpwardsTransitions == null) {
                classifyTransitions();
            }

            return outgoingUpwardsTransitions;
180     }

        /**
         * Returns the position of this state in this layer.
         *
         * @return the position
         */
        protected int getPositionInLayer() {
            return layer.getStatePos(this);
        }
190
        /**
         * Returns the selfloops.
         *
         * @return the selfloops
         */
        protected ArrayList getSelfloops() {
            if (selfloops == null) {
                classifyTransitions();
            }
200
            return selfloops;
        }

        /**
         * Returns the primitive state.
         *
         * @return the primitive state
         */
        protected Node getState() {
210         return node;
        }

        /**
         * Returns the transitons connected with this state.
         *
         * @return the transitions
         */
        protected ArrayList getTransitions() {
            ArrayList t = new ArrayList();
220
            t.addAll(getSelfloops());
            t.addAll(getIncomingDownwardsTransitions());
            t.addAll(getOutgoingDownwardsTransitions());
            t.addAll(getIncomingUpwardsTransitions());
            t.addAll(getOutgoingUpwardsTransitions());

            return t;
        }

230     /**
         * Returns the state name.
```

```
     *
     * @return the name of this state
     */
    public String toString() {
```

```
            return node.getName();
        }
    }
```

C.2.25. `kiel.layouter.hvlayouter.Level`

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

import java.util.Vector;

import kiel.graphicalInformations.View;


/**
 * <p>
 * Description: abstract level. Each layer has exactly one state level and
 * zero to many transition levels.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.9 $
 */
public abstract class Level {
    /** This is the content of this level. */
    private Vector content = new Vector();

    /**
     * @return Returns the content.
     */
    protected final Vector getContent() {
        return content;
    }

    /**
     * Returns the maximum height of this level.
     *
     * @param view
     *            curent view
     * @return the maximum height of this level
     */
    protected abstract int getMaxHeight(final View view);

    /**
     * Returns the maximum width of this level.
     *
     * @param view
     *            curent view
     * @return the maximum width of this level
     */
    protected abstract int getMaxWidth(final View view);

    /**
     * Sets the content.
     *
     * @param c
     *            The content to set.
     */
    protected final void setContent(final Vector c) {
        this.content = c;
    }
}
```

C.2.26. kiel.layouter.hvlayouter.StateLevel

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

import java.util.Collection;
import java.util.Iterator;

import kiel.graphicalInformations.NodeLayoutInformation;
import kiel.graphicalInformations.View;


/**
 * <p>
 * Description: A state level contains all states of one layer.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.8 $
 */
public final class StateLevel extends Level {
    /**
     * Creates a new instance of StateLevel.
     *
     * @param states
     *            the states in this level
     */
    protected StateLevel(final Collection states) {
        getContent().addAll(states);
    }

    /**
     * Returns the maximum height of this level.
     *
     * @param view
     *            current view
     * @return the maximum height of this level
     */
    protected int getMaxHeight(final View view) {
        int maxHeight = 0;
        Iterator iter = getContent().iterator();

        while (iter.hasNext()) {
            NodeLayoutInformation nodeLayout = view
                    .getLayoutInformation(((HVLayoutedNode) iter.next())
                            .getState());

            maxHeight = Math.max(maxHeight, nodeLayout.getHeight());
        }

        return maxHeight;
    }

    /**
     * Returns the maximum width of this level.
     *
     * @param view
     *            current view
     * @return the maximum width of this level
     */
    protected int getMaxWidth(final View view) {
        int maxWidth = 0;
        Iterator iter = getContent().iterator();

        while (iter.hasNext()) {
            NodeLayoutInformation nodeLayout = view
                    .getLayoutInformation(((HVLayoutedNode) iter.next())
                            .getState());

            maxWidth = Math.max(maxWidth, nodeLayout.getWidth());
        }

        return maxWidth;
    }
}
```

C.2.27. kiel.layouter.hvlayouter.TransitionLevel

```java
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

import java.util.BitSet;
import java.util.Iterator;

import kiel.dataStructure.Edge;
import kiel.dataStructure.Transition;
import kiel.graphicalInformations.LabelLayoutInformation;
import kiel.graphicalInformations.View;


/**
 * <p>
 * Description: Each position in a transition level can be assigned only
 *   once.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.9 $
 */
public final class TransitionLevel extends Level {
    /**
     * This is the load of this level, each bit symbolizes the space
     between
     * two states.
     */
    private BitSet load;

    /**
     *  This is the size of the load (number of states in the belonging
     layer).
     */
    private int size;

    /**
     * Creates a new instance of TransitionLevel.
     *
     * @param numberOfStates
     *            the number of states in the belonging layer
     */
    protected TransitionLevel(final int numberOfStates) {
        size = numberOfStates - 1;
        load = new BitSet(size);
    }

    /**
     * Adds transition into this level, if it fits.
     *
     * @param transition
     *            transition to be added
     */
    protected void addTransition(final HVLayoutedEdge transition) {
        if (fits(transition)) {
            load.or(getTransitionBitSet(transition));
            getContent().add(transition);
        }
    }

    /**
     * Returns true, if transition fits into this level.
     *
     * @param transition
     *            transition to be tested
     * @return true, if transition fits into this level
     */
    protected boolean fits(final HVLayoutedEdge transition) {
        return !load.intersects(getTransitionBitSet(transition));
    }

    /**
     * Returns the maximum height of this level.
     *
     * @param view
     *            current view
     * @return the maximum height of this level
     */
    protected int getMaxHeight(final View view) {
        int maxHeight = 0;
        Iterator iter = getContent().iterator();

        while (iter.hasNext()) {
            Edge edge = ((HVLayoutedEdge) iter.next()).getTransition();
            int labelHeight;

            if (edge instanceof Transition) {
                LabelLayoutInformation labelLayout;

                labelLayout = view.getLayoutInformation(((Transition) edge)
                        .getLabel());
                labelLayout.setDimension(((Transition) edge)
                        .getLabel().toString());
                labelHeight = labelLayout.getHeight();
            } else {
                labelHeight = 0;
            }

            maxHeight = Math.max(maxHeight, labelHeight);
        }

        return maxHeight;
    }
```

```
110
        /**
         * Returns the maximum width of this level.
         *
         * @param view
         *            current view
         * @return the maximum width of this level
         */
        protected int getMaxWidth(final View view) {
            int maxWidth = 0;
120         Iterator iter = getContent().iterator();

            while (iter.hasNext()) {
                Edge edge = ((HVLayoutedEdge) iter.next()).getTransition();
                int labelWidth;

                if (edge instanceof Transition) {
                    LabelLayoutInformation labelLayout;

                    labelLayout = view.getLayoutInformation(((Transition) edge)
130                         .getLabel());
                    labelLayout.setDimension(((Transition) edge)
                            .getLabel().toString());
                    labelWidth = labelLayout.getWidth();
                } else {
                    labelWidth = 0;
                }

                maxWidth = Math.max(maxWidth, labelWidth);
            }
140
            return maxWidth;
        }

        /**
         * Returns the bit set of transition.
         *
         * @param transition
         *            the transition the bit set should be return of
         * @return the bit set of transition
150      */
        protected BitSet getTransitionBitSet(final HVLayoutedEdge transition) {
            BitSet transitionBitSet = new BitSet(size);
            transitionBitSet.set(Math.min(transition.getSource()
                    .getPositionInLayer(), transition.getSink()
                    .getPositionInLayer()), Math.max(transition.getSource()
                    .getPositionInLayer(), transition.getSink()
                    .getPositionInLayer()));

            return transitionBitSet;
160     }

        /**
         * Returns a string containing the load of this transition level.
         *
         * @return a string containing the load of this transition level
         */
        public String toString() {
            return load.toString();
        }
170 }
```

C.2.28. kiel.layouter.hvlayouter.PropertyComparator

```java
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

/**
 * <p>
 * Description: This is the basic property comparator.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.8 $
 */
public abstract class PropertyComparator {
    /** Creates a new instance of PropertyComparator. */
    protected PropertyComparator() {
    }

    /**
     * Compares two layers.
     *
     * @param layerOne
     *            a layer
     * @param layerTwo
     *            another layer
     * @return -1 if layerTwo is better layouted 0 if both layers a equal
     +1
     *         if layerOne is better layouted
     */
    protected abstract int compare(final HVLayoutedLayer layerOne,
            final HVLayoutedLayer layerTwo);
}
```

C.2.29. kiel.layouter.hvlayouter.InitialStateComparator

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

import java.util.Iterator;

import kiel.dataStructure.InitialState;

/**
 * <p>
 * Description: Compares the position of the initial states.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.7 $
 */
public final class InitialStateComparator extends PropertyComparator {
    /** Creates a new instance of InitialStateComparator. */
    protected InitialStateComparator() {
    }

    /**
     * Compares two layers by their position of each initial state.
     *
     * @param layerOne a layer
     * @param layerTwo another layer
     * @return -1 if layerTwo is better layouted
     *          0 if both layers a equal
     *         +1 if layerOne is better layouted
     */
    protected int compare(final HVLayoutedLayer layerOne,
                          final HVLayoutedLayer layerTwo) {
        int pos1 = 0;
        int pos2 = 0;
        int result;
        Iterator iter;
        HVLayoutedNode state;

        iter = layerOne.getNodes().iterator();
        while (iter.hasNext()) {
            state = (HVLayoutedNode) iter.next();
            if (state.getState() instanceof InitialState) {
                pos1 = state.getPositionInLayer();
                break;
            }
        }

        iter = layerTwo.getNodes().iterator();
        while (iter.hasNext()) {
            state = (HVLayoutedNode) iter.next();
            if (state.getState() instanceof InitialState) {
                pos2 = state.getPositionInLayer();
                break;
            }
        }

        if (pos1 > pos2) {
            result = -1;
        } else if (pos1 == pos2) {
            result = 0;
        } else {
            result = 1;
        }

        return result;
    }
}
```

C.2.30. kiel.layouter.hvlayouter.NumberOfLevelsComparator

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

/**
 * <p>
 * Description: Compares two levels by their numbers of levels.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.8 $
 */
public final class NumberOfLevelsComparator extends PropertyComparator {
    /** Creates a new instance of NumberOfLevelsComparator. */
    protected NumberOfLevelsComparator() {
    }

    /**
     * Compares two layers by their number of levels.
     *
     * @param layerOne
     *            a layer
     * @param layerTwo
     *            another layer
     * @return -1 if layerTwo is better layouted 0 if both layers a equal
     +1
     *         if layerOne is better layouted
     */
    protected int compare(final HVLayoutedLayer layerOne,
            final HVLayoutedLayer layerTwo) {
        int nol1;
        int nol2;
        int result;

        nol1 = layerOne.getNumberOfDownwardsTransitionLevels()
                + layerOne.getNumberOfUpwardsTransitionLevels();
        nol2 = layerTwo.getNumberOfDownwardsTransitionLevels()
                + layerTwo.getNumberOfUpwardsTransitionLevels();

        if (nol1 > nol2) {
            result = -1;
        } else if (nol1 == nol2) {
            result = 0;
        } else {
            result = 1;
        }

        return result;
    }
}
```

C.2.31. kiel.layouter.hvlayouter.TransitionCrossingComparator

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

/**
 * <p>
 * Description: Compares two layers by the number of their transition
 * crossings.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.8 $
 */
public final class TransitionCrossingComparator extends PropertyComparator
     {
    /** Creates a new instance of TransitionCrossingComparator. */
    protected TransitionCrossingComparator() {
    }

    /**
     * Compares two layers by the number of their transition crossings.
     *
     * @param layerOne
     *            a layer
     * @param layerTwo
     *            another layer
     * @return -1 if layerTwo is better layouted 0 if both layers a equal
     +1
     *         if layerOne is better layouted
     */
    protected int compare(final HVLayoutedLayer layerOne,
            final HVLayoutedLayer layerTwo) {
        int crossings1;
        int crossings2;
        int result;

        crossings1 = layerOne.getNumberOfTransitionCrossings();
        crossings2 = layerTwo.getNumberOfTransitionCrossings();

        if (crossings1 > crossings2) {
            result = -1;
        } else if (crossings1 == crossings2) {
            result = 0;
        } else {
            result = 1;
        }

        return result;
    }
}
```

C.2.32. kiel.layouter.hvlayouter.WellBalancedLevelsComparator

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

/**
 * <p>
 * Description: Compares two layers by looking at the balance of their
     levels.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.8 $
 */
public final class WellBalancedLevelsComparator extends PropertyComparator
     {
    /** Creates a new instance of WellBalancedLevelsComparator. */
    protected WellBalancedLevelsComparator() {
    }

    /**
     * Compares two layers by looking at the balance of their levels.
     *
     * @param layerOne
     *            a layer
     * @param layerTwo
     *            another layer
     * @return -1 if layerTwo is better layouted 0 if both layers a equal
     +1
     *         if layerOne is better layouted
     */
    protected int compare(final HVLayoutedLayer layerOne,
            final HVLayoutedLayer layerTwo) {
        int levelDiff1;
        int levelDiff2;
        int result;

        levelDiff1 = Math.abs(layerOne.getNumberOfDownwardsTransitionLevels
     ()
                - layerOne.getNumberOfUpwardsTransitionLevels());
        levelDiff2 = Math.abs(layerTwo.getNumberOfDownwardsTransitionLevels
     ()
                - layerTwo.getNumberOfUpwardsTransitionLevels());

        if (levelDiff1 > levelDiff2) {
            result = -1;
        } else if (levelDiff1 == levelDiff2) {
            result = 0;
        } else {
            result = 1;
        }

        return result;
    }
}
```

C.2.33. kiel.layouter.hvlayouter.PermutationIterator

```
/*
 * $Author: tkl $
 *
 * $Date: 2005/03/15 22:40:36 $
 */
package kiel.layouter.hvlayouter;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

/**
 * <p>
 * Description: This an iterator to get all permutations of a given
 *    collection.
 * </p>
 * <p>
 * Copyright: Copyright (c) 2004
 * </p>
 * <p>
 * Company: Uni Kiel
 * </p>
 *
 * @author <a href="mailto:tkl@informatik.uni-kiel.de">Tobias Kloss </a>
 * @version $Revision: 1.7 $
 */
public final class PermutationIterator {
    /** This contains all objects. */
    private ArrayList collection;

    /** This is the currently observed object. */
    private Object currentObject;

    /** This iterates over all objects. */
    private Iterator iterator;

    /**
     * This iterates over all permutations of the objects
     * except the current object.
     */
    private PermutationIterator subiterator;

    /**
     * This is the contructor.
     *
     * @param c a collection
     */
    protected PermutationIterator(final Collection c) {
        collection = new ArrayList(c);
        iterator = collection.iterator();
        subiterator = null;
    }

    /**
     * Returns true if the iteration has more elements. (In other words,
     * returns true if next would return an element rather than throwing an
     * exception.)
     *
     * @return true if the iterator has more elements.
     */
    protected boolean hasMorePermutations() {
        return iterator.hasNext()
            || ((subiterator != null) && subiterator.hasMorePermutations())
     ;
    }

    /**
     * Returns the next element in the iteration.
     *
     * @return the next element in the iteration.
     */
    protected Collection nextPermutation() {
        ArrayList result = new ArrayList();

        if ((subiterator == null) || (!subiterator.hasMorePermutations()))
     {
            currentObject = iterator.next();
            if (collection.size() > 1) {
                ArrayList al = new ArrayList(collection);
                al.remove(currentObject);
                subiterator = new PermutationIterator(al);
            }
        }

        result.add(currentObject);
        if (subiterator != null) {
            result.addAll(subiterator.nextPermutation());
        }

        return result;
    }
}
```

C.3. Der *C++-Code*

C.3.1. `kiel_layouter_graphviz_GraphvizAPI.h`

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class kiel_layouter_graphviz_GraphvizAPI */

#ifndef _Included_kiel_layouter_graphviz_GraphvizAPI
#define _Included_kiel_layouter_graphviz_GraphvizAPI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    createGraph
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_createGraph
    (JNIEnv *, jclass, jstring);

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    createNode
 * Signature: (ILjava/lang/String;)I
 */
JNIEXPORT jint JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_createNode
    (JNIEnv *, jclass, jint, jstring);

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    createEdge
 * Signature: (III)I
 */
JNIEXPORT jint JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_createEdge
    (JNIEnv *, jclass, jint, jint, jint);

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    doDotLayout
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_doDotLayout
    (JNIEnv *, jclass, jint);

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    doNeatoLayout
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_doNeatoLayout
    (JNIEnv *, jclass, jint);

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    doTwopiLayout
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_doTwopiLayout
    (JNIEnv *, jclass, jint);

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    doCircoLayout
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_doCircoLayout
    (JNIEnv *, jclass, jint);

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    closeGraph
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_closeGraph
    (JNIEnv *, jclass, jint);

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    dotCleanup
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_dotCleanup
    (JNIEnv *, jclass, jint);

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    neatoCleanup
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_neatoCleanup
    (JNIEnv *, jclass, jint);

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    twopiCleanup
 * Signature: (I)V
 */
```

```
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_twopiCleanup
        (JNIEnv *, jclass, jint);

    /*
110  * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    circoCleanup
     * Signature: (I)V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_circoCleanup
        (JNIEnv *, jclass, jint);

    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
120  * Method:    setGraphAttribute
     * Signature: (ILjava/lang/String;Ljava/lang/String;)V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_setGraphAttribute
        (JNIEnv *, jclass, jint, jstring, jstring);

    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    setGlobalNodeAttribute
130  * Signature: (ILjava/lang/String;Ljava/lang/String;)V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_setGlobalNodeAttribute
        (JNIEnv *, jclass, jint, jstring, jstring);

    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    setLocalNodeAttribute
     * Signature: (IILjava/lang/String;Ljava/lang/String;)V
140  */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_setLocalNodeAttribute
        (JNIEnv *, jclass, jint, jint, jstring, jstring);

    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    setGlobalEdgeAttribute
     * Signature: (ILjava/lang/String;Ljava/lang/String;)V
     */
150 JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_setGlobalEdgeAttribute
        (JNIEnv *, jclass, jint, jstring, jstring);

    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    setLocalEdgeAttribute
     * Signature: (IILjava/lang/String;Ljava/lang/String;)V
     */
    JNIEXPORT void JNICALL
160   Java_kiel_layouter_graphviz_GraphvizAPI_setLocalEdgeAttribute
        (JNIEnv *, jclass, jint, jint, jstring, jstring);

    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    getAttribute
     * Signature: (ILjava/lang/String;)Ljava/lang/String;
     */
    JNIEXPORT jstring JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_getAttribute
170     (JNIEnv *, jclass, jint, jstring);

    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    attachAttributes
     * Signature: (I)V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_attachAttributes
        (JNIEnv *, jclass, jint);
180
    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    writeDOT
     * Signature: (ILjava/lang/String;)V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_writeDOT
        (JNIEnv *, jclass, jint, jstring);

190 /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    getVersionString
     * Signature: ()Ljava/lang/String;
     */
    JNIEXPORT jstring JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_getVersionString
        (JNIEnv *, jclass);

    /*
200  * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    initialize
     * Signature: ()V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_initialize
        (JNIEnv *, jclass);

    #ifdef __cplusplus
    }
210 #endif
    #endif
```

C.3.2. kiel_layouter_graphviz_GraphvizAPI.c

```c
#include <jni.h>
#include <dotneato.h>
#include <string.h>
#include "kiel_layouter_graphviz_GraphvizAPI.h"

extern char* Info[];

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    createGraph
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_createGraph
    (JNIEnv *env, jclass obj, jstring name)
{
    Agraph_t *g;
    const char *c_constname = (*env)->GetStringUTFChars(env, name, 0);
    char *c_name = malloc((strlen(c_constname) + 1) * sizeof(char));

    strcpy(c_name, c_constname);
    g = agopen(c_name, AGDIGRAPH);
    dot_init_graph(g);

    return (jint) g;
}

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    createNode
 * Signature: (ILjava/lang/String;)I
 */
JNIEXPORT jint JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_createNode
    (JNIEnv *env, jclass obj, jint graph, jstring name)
{
    Agnode_t *node;
    const char *c_constname = (*env)->GetStringUTFChars(env, name, 0);
    char *c_name = malloc((strlen(c_constname) + 1) * sizeof(char));

    strcpy(c_name, c_constname);
    node = agnode((Agraph_t*) graph, c_name);

    return (jint) node;
}

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    createEdge
 * Signature: (III)I
 */
JNIEXPORT jint JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_createEdge
    (JNIEnv *env, jclass obj, jint graph, jint source, jint target)
{
    Agedge_t *edge;

    edge = agedge((Agraph_t*) graph, (Agnode_t*) source,
                  (Agnode_t*) target);

    return (jint) edge;
}

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    doDotLayout
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_doDotLayout
    (JNIEnv *env, jclass obj, jint graph)
{
    dot_layout((Agraph_t*) graph);
}

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    doNeatoLayout
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_doNeatoLayout
    (JNIEnv *env, jclass obj, jint graph)
{
    neato_layout((Agraph_t*) graph);
}

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    doTwopiLayout
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_doTwopiLayout
    (JNIEnv *env, jclass obj, jint graph)
{
    twopi_layout((Agraph_t*) graph);
}

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    doCircoLayout
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_doCircoLayout
    (JNIEnv *env, jclass obj, jint graph)
{
    circo_layout((Agraph_t*) graph);
}

/*
```

```
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    closeGraph
     * Signature: (I)V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_closeGraph
        (JNIEnv *env , jclass obj, jint graph)
120 {
        agclose((Agraph_t*) graph);
    }

    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    dotCleanup
     * Signature: (I)V
     */
    JNIEXPORT void JNICALL
130   Java_kiel_layouter_graphviz_GraphvizAPI_dotCleanup
        (JNIEnv *env , jclass obj, jint graph)
    {
        dot_cleanup((Agraph_t*) graph);
    }

    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    neatoCleanup
     * Signature: (I)V
140  */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_neatoCleanup
        (JNIEnv *env , jclass obj, jint graph)
    {
        neato_cleanup((Agraph_t*) graph);
    }

    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
150  * Method:    twopiCleanup
     * Signature: (I)V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_twopiCleanup
        (JNIEnv *env , jclass obj, jint graph)
    {
        twopi_cleanup((Agraph_t*) graph);
    }

160 /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    circoCleanup
     * Signature: (I)V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_circoCleanup
        (JNIEnv *env , jclass obj, jint graph)
    {
        circo_cleanup((Agraph_t*) graph);
170 }

    /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    setGraphAttribute
     * Signature: (ILjava/lang/String;Ljava/lang/String;)V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_setGraphAttribute
        (JNIEnv *env, jclass obj, jint graph, jstring attr, jstring value)
180 {
        const char *c_constattr = (*env)->GetStringUTFChars(env, attr, 0);
        char *c_attr = malloc((strlen(c_constattr) + 1) * sizeof(char));
        const char *c_constvalue = (*env)->GetStringUTFChars(env, value, 0);
        char *c_value = malloc((strlen(c_constvalue) + 1) * sizeof(char));

        strcpy(c_attr, c_constattr);
        strcpy(c_value, c_constvalue);
        agraphattr((Agraph_t*) graph, c_attr, c_value);
    }
190 /*
     * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    setGlobalNodeAttribute
     * Signature: (ILjava/lang/String;Ljava/lang/String;)V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_setGlobalNodeAttribute
        (JNIEnv *env, jclass obj, jint graph, jstring attr, jstring value)
    {
        const char *c_constattr = (*env)->GetStringUTFChars(env, attr, 0);
200     char *c_attr = malloc((strlen(c_constattr) + 1) * sizeof(char));
        const char *c_constvalue = (*env)->GetStringUTFChars(env, value, 0);
        char *c_value = malloc((strlen(c_constvalue) + 1) * sizeof(char));

        strcpy(c_attr, c_constattr);
        strcpy(c_value, c_constvalue);
        agnodeattr((Agraph_t*) graph, c_attr, c_value);
    }

    /*
210  * Class:     kiel_layouter_graphviz_GraphvizAPI
     * Method:    setLocalNodeAttribute
     * Signature: (IILjava/lang/String;Ljava/lang/String;)V
     */
    JNIEXPORT void JNICALL
      Java_kiel_layouter_graphviz_GraphvizAPI_setLocalNodeAttribute
        (JNIEnv *env, jclass obj, jint graph,
         jint node, jstring attr, jstring value)
    {
        const char *c_constattr = (*env)->GetStringUTFChars(env, attr, 0);
220     char *c_attr = malloc((strlen(c_constattr) + 1) * sizeof(char));
        const char *c_constvalue = (*env)->GetStringUTFChars(env, value, 0);
        char *c_value = malloc((strlen(c_constvalue) + 1) * sizeof(char));

        strcpy(c_attr, c_constattr);
        strcpy(c_value, c_constvalue);
        if (agfindattr((void*) graph, c_attr) == 0) {
            agnodeattr((Agraph_t*) graph, c_attr, "");
        }
        agset((void*) node, c_attr, c_value);
230 }

    /*
```

```
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    setGlobalEdgeAttribute
 * Signature: (ILjava/lang/String;Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
   Java_kiel_layouter_graphviz_GraphvizAPI_setGlobalEdgeAttribute
    (JNIEnv *env, jclass obj, jint graph,
240  jstring attr, jstring value)
{
    const char *c_constattr = (*env)->GetStringUTFChars(env, attr, 0);
    char *c_attr = malloc((strlen(c_constattr) + 1) * sizeof(char));
    const char *c_constvalue = (*env)->GetStringUTFChars(env, value, 0);
    char *c_value = malloc((strlen(c_constvalue) + 1) * sizeof(char));

    strcpy(c_attr, c_constattr);
    strcpy(c_value, c_constvalue);
    agedgeattr((Agraph_t*) graph, c_attr, c_value);
250 }

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    setLocalEdgeAttribute
 * Signature: (IILjava/lang/String;Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
   Java_kiel_layouter_graphviz_GraphvizAPI_setLocalEdgeAttribute
    (JNIEnv *env, jclass obj, jint graph,
260  jint edge, jstring attr, jstring value)
{
    const char *c_constattr = (*env)->GetStringUTFChars(env, attr, 0);
    char *c_attr = malloc((strlen(c_constattr) + 1) * sizeof(char));
    const char *c_constvalue = (*env)->GetStringUTFChars(env, value, 0);
    char *c_value = malloc((strlen(c_constvalue) + 1) * sizeof(char));

    strcpy(c_attr, c_constattr);
    strcpy(c_value, c_constvalue);
    if (agfindattr((void*) graph, c_attr) == 0) {
270     agedgeattr((Agraph_t*) graph, c_attr, "");
    }
    agset((void*) edge, c_attr, c_value);
}

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    getAttribute
 * Signature: (ILjava/lang/String;)Ljava/lang/String;
 */
280 JNIEXPORT jstring JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_getAttribute
    (JNIEnv *env, jclass obj, jint elem, jstring attr)
{
    const char *c_constattr = (*env)->GetStringUTFChars(env, attr, 0);
    char *c_attr = malloc((strlen(c_constattr) + 1) * sizeof(char));
    char *c_value;

    strcpy(c_attr, c_constattr);
    c_value = agget((void*) elem, c_attr);
290
    return (*env)->NewStringUTF(env, c_value);
}

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    attachAttributes
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
300  Java_kiel_layouter_graphviz_GraphvizAPI_attachAttributes
    (JNIEnv *env, jclass obj, jint graph)
{
    attach_attrs((Agraph_t*) graph);
}

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    writeDOT
 * Signature: (ILjava/lang/String;)V
310 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_writeDOT
    (JNIEnv *env, jclass obj, jint graph, jstring filename)
{
    const char *c_constfilename = (*env)->GetStringUTFChars(env, filename,
     0);
    char *c_filename = malloc((strlen(c_constfilename) + 1) * sizeof(char))
     ;
    FILE *file;

    strcpy(c_filename, c_constfilename);
320 file = fopen(c_filename, "w");

    agwrite((Agraph_t*) graph, file);
    fclose(file);
}

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
 * Method:    getVersionString
 * Signature: ()Ljava/lang/String;
330 */
JNIEXPORT jstring JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_getVersionString
    (JNIEnv *env, jclass obj)
{
    return (*env)->NewStringUTF(env, Info[1]);
}

/*
 * Class:     kiel_layouter_graphviz_GraphvizAPI
340 * Method:    initialize
 * Signature: ()V
 */
JNIEXPORT void JNICALL
  Java_kiel_layouter_graphviz_GraphvizAPI_initialize
    (JNIEnv *env, jclass obj) {
    aginit();
}
```