JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF
Technisch-Naturwissenschaftliche
Fakultät

StateGraphViewer
Ein Eclipse-Plugin zur visuellen Darstellung von Zustandsmodellen

BACHELORARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

INFORMATIK

Eingereicht von:
Martin Ennemoser, 0855518

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
o.Univ.-Prof. Dr. Dr. hc. Hanspeter Mössenböck

Mitwirkung:
Dr. Herbert Prähofer

Linz, Februar 2012

Kurzfassung

Fujitsu Microelectronics hat in Zusammenarbeit mit dem Institut für Systemsoftware (SSW) eine spezielle Domain Specific Language (kurz DSL) zur Beschreibung von HMI-Systemen (Human Machine Inteface - Systeme) im Bereich der Automobil-Industrie entwickelt. Diese DSL ist eine textbasierte Programmiersprache und beschreibt das Verhalten von reaktiven HMI-Systemen mithilfe von Zustandsautomaten.

Diese Arbeit verfolgt das Ziel, ein System zu implementieren, welches eine graphische Darstellung aus diesen textuell beschriebenen Zustandsmodellen generiert. Eine wichtige Eigenschaft dieses Systems ist dabei, dass das graphische Layout automatisch erzeugt werde soll. Aus diesem Grund implementiert StateGraphViewer spezielle Layout-Algorithmen, um diesen automatischen Layout-Vorgang zu verwirklichen. StateGraphViewer wurde dabei als Eclipse-Plugin entwickelt und unterstützt das automatische Layouting von textbasierten Zustandsmodellen.

Zu Beginn dieser Arbeit werden die eingesetzten Technologien wie das Eclipse-Framework und das Graphen-Framework Zest näher erklärt. Anschließend wird der Layout-Algorithmus, welcher die Größe und Position der Knoten und Kanten eines Zustandsgraphen bestimmt, beschrieben.

Abstract

Fujitsu Microelectronics developed in cooperation with the Institute for Systems Software (SSW) a Domain Specific Language (DSL) for the definition of HMI systems (human machine interface systems) in the automotive domain. This DSL is a text-based programming language for the modeling of state transition models. This state transition models describe the reactive behavior of HMI systems.

In course of this thesis work a system–called StateGraphViewer –has been developed which generates a graphical representation out of these text-based finite automaton specifications. One important property of this system is that the graphical layout is generated automatically. Therefore StateGraphViewer implements special layout algorithms to realize this layout process. StateGraphViewer is implemented as an Eclipse plugin and supports the automatic layout process of text-based finite automata.

In the first part of this thesis an introduction into the used technologies, like the Eclipse framework of the graph framework Zest, are given. The next part focuses on the implementation of the layout algorithm that determines the width and height of nodes and edges of state transition models.

Inhalt

1 Einleitung und Zielsetzung

Diese Bachelorarbeit beschreibt die Realisierung einer graphischen Darstellung von Zustandsautomaten mit dem Eclipse-Plugin *Zest*. Am *Institut für Systemsoftware* wurde in Kooperation mit *Fujitsu Microelectronics* (http://emea.fujitsu.com/microelectronics) ein Prototyp eines Systems zur Modellierung von Zustandsmodellen entwickelt, welches das Verhalten von *HMI-Systemen* (Human Machine Interface) im Automobilbereich ermöglicht. Zustandsmodelle sind dabei ähnlich wie *Statecharts* hierarchisch organisiert. Im Gegensatz zu Statecharts-Modellierungssystemen werden die Modelle nicht graphisch, sondern in einer textbasierten Programmiersprache modelliert. Diese Sprache wurde *CGI-DSL* (Cluster Graphics Interface – Domain Specific Language) genannt.

In dieser Arbeit soll nun eine graphische Darstellung dieser Zustandsmodelle realisiert werden. Eine wichtige Eigenschaft des Systems ist dabei, dass die Darstellungen automatisch aus den textbasierten Modellen generiert werden. Das heißt, die Knoten und Kanten müssen automatisch positioniert, die Größe bestimmt und die Verbindungen entsprechend gelayoutet werden. Ein wesentlicher Teil der Arbeit ist daher die Einteilung und Implementierung der entsprechenden Layout-Algorithmen für die Darstellung der Zustandsmodelle. Die Implementierung des Systems erfolgt als Eclipse-Plugin unter Verwendung der Systeme *Draw2d*, *JFace* und *Zest*.

Die Arbeit gliedert sich in folgende Kapitel:

- Kapitel 2 gibt einen Überblick über die wichtigsten Komponten von *Statecharts*
- Kapitel 3 erklärt die wichtigsten Eigenschaften der CGI-DSL

- Kapitel 4 zeigt, wie die Eclipse-Platform aufgebaut ist. Desweiteren werden die Plugins Zest und Draw2d erklärt.
- Kapitel 5 stellt die eigentliche Arbeit vor. Es werden die Features von StateGraphViewer erklärt, sowie die implementierten Algorithmen näher erklärt.
- Kapitel 6 geht näher auf die Implementierung ein und zeigt die StateGraphViewer -Komponente in Form eines Paketdiagrammes.
- Kapitel 7 fasst diese Arbeit nochmals zusammen und präsentiert die persönlichen Erfahrungen, die während der Entwicklung gemacht wurden.

2 Statecharts

Statecharts sind eine Möglichkeit, um das dynamische Verhalten während des Lebenszyklus von Objekten beziehungsweise von endlichen Automaten zu beschreiben. Ein Statechart stellt einen Graph dar, dessen Knoten den Zuständen (auch States genannt) entsprechen, die von Objekten eingenommen werden können und dessen gerichtete Kanten die möglichen Zustandsübergänge (auch Transitionen genannt) angeben.

2.1 Zustände

Zustände beschreiben ein System zu einem gewissen Zeitpunkt. Im Diagramm werden Zustände durch Rechtecke mit abgerundeten Ecken dargestellt. Das Rechteck kann in zwei Abschnitte unterteilt sein: Der obere Abschnitt enthält den Namen des Zustandes, im unteren Bereich können Angaben über Aktivitäten und sogenannte innere Transitionen stehen. Eine Aktivität wird mit dem Präfix */do* angegeben.

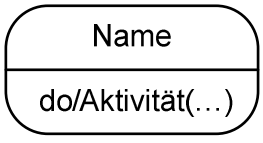

Abbildung 2.1: Darstellung eines Zustandes

Objekte verweilen solange in einem bestimmten Zustand, bis ein Ereignis ausgelöst, wird, welches eine Zustandsüberführung bewirkt.

Um den Start und das Ende eines Diagrammes zu visualisieren gibt es spezielle Notationen für Start- und Endzustände. Ein Startzustand wird als gefüllter Kreis dargestellt während ein Endzustand als gefüllter Kreis mit umgebendem Ring visualisiert wird.

Abbildung 2.2: Start- und Endzustandssymbole

2.2 Transitionen

Transitionen beschreiben eine Zustandsüberführung von einem Zustand in einen anderen. Sie werden durch einen beschrifteten Pfeil dargestellt, der einen Quellzustand (oder auch Source-State genannt) mit dessen Folgezustand (Target-State) verbindet.

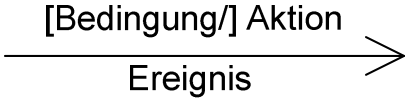

Abbildung 2.3: Darstellung einer Transition

Die Beschriftung einer Transition enthält folgende Angaben:

- Das Ereignis, welches ausgelöst wird, sobald die zugehörige Bedingung erfüllt ist (siehe nächster Punkt). Durch das ausgelöste Ereignis erfolgt die Transition in einen anderen Zustand. Fehlt die Angabe des Ereignisses, so gilt die Beendigung der Aktivität im Vorzustand als Auslöser der Transition.
- Eine optionale Bedingung, die die Transition überwacht. Wenn ein Ereignis eintritt, jedoch die Bedingung nicht erfüllt ist, erfolgt keine Zustandsüberführung und das Ereignis geht verloren. Um eine Zustandsüberführung zu bewirken muss ein Ereignis vorhanden sein und die Bedingung muss erfüllt sein. Da die

Bedingung eine Transition überwacht werden Bedingungen häufig auch als *Guards* [Zitzke 2011] bezeichnet.

- Aktionen, die ausgeführt werden sobald die Transition schaltet. Aktionen können sehr unterschiedliche Verhalt in Pseudocode oder in einer Programmiersprache formuliert sein.

2.3 ODER-Verfeinerung

Die ODER-Verfeinerung aus [Hitz & Kappel 1999] erlaubt es komplexe Zustände in Subzuständen aufzuspalten. Ein Zustand kann damit mehrere Subzustände besitzen, wobei immer nur ein Subzustand aktiv sein kann, wenn der komplexe Zustand aktiv ist. Ein Beispiel für einen komplexen Zustand liefert Abbildung 1.4. Der komplexe Zustand Z verschachtelt zwei weitere Zustände X und Y.

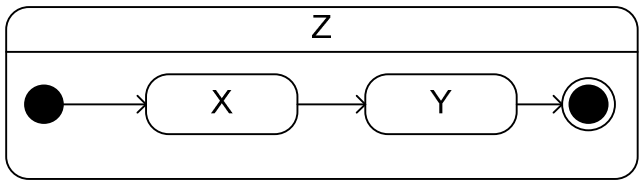

Abbildung 2.4: Zusammengesetzter Zustand

Wenn eine Transition in den Zustand Z führt, entspricht das einer Transition in den Startzustand der Verfeinerung. Die Transition von Y zum Endzustand führt zur Beendigung der Aktivität von Z und dem Schalten einer allfälligen „ereignislosen" Transition.

Eine besondere Bedeutung im Zusammenhang mit ODER-Verfeinerungen nehmen sogenannte History-Zustände ein. Ein History-Zustand ist ein Pseudozustand innerhalb einer Verfeinerung, welcher „Buch" über den zuletzt aktiven Subzustand innerhalb der Verfeinerung führt. Das heißt wenn eine Transition in den History-Zustand führt, dann wird wieder der Subzustand der Verfeinerung angenommen, in der sich auch der History-Zustand befindet. Abbildung 2.5 zeigt einen History-Zustand inner-

halb von Y. Ein Historyzustand wird als Kreis mit einem eingeschlossenen „H“ gekennzeichnet.

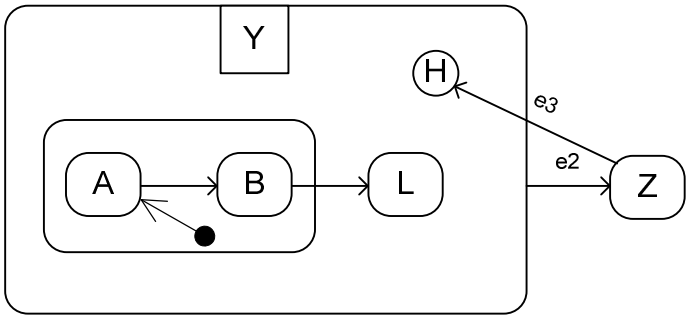

Abbildung 2.5: Oder-Verfeinerung mit History-Zustand

Das Ereignis e2 löst einen Übergang von jedem beliebigen Subzustand von Y nach Z aus. Die anschließende e3-Transition führ zur Rückkehr in einen der Subzustände von Y. Ob dabei der Subzustand K oder L angenommen wird hängt davon ab, welcher Zustand das Ereignis e2 ausgelöst hat.

In Abbildung 2.5 ist jedoch im Falle der Rückkehr zu K nicht eindeutig definiert, ob der Subzustand A oder B angenommen werden soll. Ein „normaler“ History-Zustand kann nämlich nur „Buchhaltung“ über die States führen, die sich auf gleicher Ebene mit dem History-Zustand befinden. Zu diesem Zweck wurden sogenannte „tiefe“ History-Zustände eingeführt. Ein tiefer History-Zustand dehnt das „Gedächtnis“ auf alle Subzustände von beliebigen Verfeinerungsebenen des komplexen Zustands.

3 CGI-DSL

Die CGI-DSL ist eine textbasierte *Domain Specific Language*, die am Institut für Systemsoftware entwickelt wurde. Mithilfe dieser DSL kann die reaktive Verhaltenslogik von MMI-Systemen (Machine Machine Interface – System) modelliert werden. Diese DSL ist dabei in einer 3-schichten Architektur aufgebaut, wobei jede Schicht eine Aufgabe besitzt und von den anderen Schichten klar abgetrennt ist.

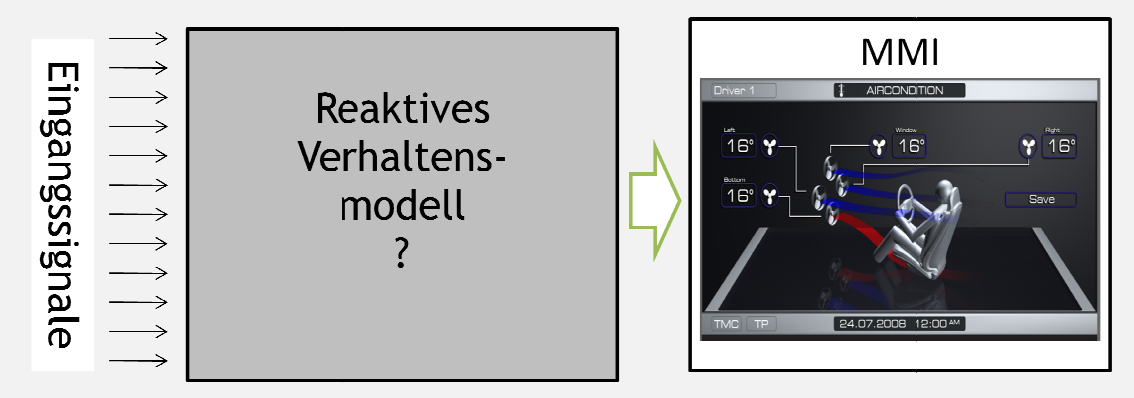

Abbildung 3.1: 3 Schichten Architektur

Abbildung 3.1 zeigt diese 3 Schichten anhand eines Beispiels: Ganz links sind die Inputsignale eines PKWs gelistet, welche von Sensoren erzeugt werden. Diese Signale werden in reaktive Verhaltensmodelle (*State-Models*) für die Ablauflogik umgewandelt. UI-Modelle (User Interface –Modelle) besitzen eine Anzahl an Szenen, welche die Schnittstelle zwischen Benutzer und Maschine darstellt.

Im Folgenden soll nun näher auf jede einzelne Schicht eingegangen werden. Im Bezug zu StateGraphViewer ist vor allem die 2. Schicht wichtig, da StateGraphViewer State-Models grafisch repräsentiert.

3.1 Schicht 1: Inputsignale

Inputsignale werden mit dem Schlüsselwort `signal` definiert und besitzen folgende Eigenschaften:

- einen Namen
- einen Typ (Bool, Int, Real)
- einen optionalen Range, welcher in eckigen Klammern neben dem Datentyp angegeben werden kann
- einen initalen Startwert

Abbildung 3.2 zeigt ein Beispiel für die Definition von Signalen. Des Weiteren können Signale mit dem Schlüsselwort `structure` zusammengefasst werden.

Abbildung 3.2: Signale

3.2 Schicht 2: Verhaltensmodelle

Verhaltensmodelle abstrahieren von Signalen und zerlegen diese in viele einzelne Automaten (State-Models), welche disjunkte Zustände halten. State-Models können Substates über sogenannte OrStates verschachteln. Diese entsprechen der ODER-Verfeinerung aus Kapitel 2. Ein Beispiel für ein State-Model mit dem Namen *Direction* ist in Abbildung 3.3 zu sehen. Dieses State-Model hat einen übergeordneten OrState *Forward*, welcher wiederum sechs Substates besitzt.

```
115
116  statemodel Direction
117      states {
118          orstate Forward {
119              state Gear1 cond [Key_pos.Ignition && gear == 1]
120              state Gear2 cond [Key_pos.Ignition && gear == 2]
121              state Gear3 cond [Key_pos.Ignition && gear == 3]
122              state Gear4 cond [Key_pos.Ignition && gear == 4]
123              state Gear5 cond [Key_pos.Ignition && gear == 5]
124              state Gear6 cond [Key_pos.Ignition && gear == 6]
125          }
126          state Idle cond [Key_pos.Ignition && gear == 0]
127          state Backward cond [Key_pos.Ignition && gear == -1]
128      }
129
```

Abbildung 3.3: Ein einfaches State-Model

States können sogenannte *Conditions* besitzen. Das sind Boolsche Bedingungen, welche den aktiven State bestimmen. In Abbildung 3.3 ist zum Beispiel der Zustand *Gear3* aktiv, sobald die Condition `(Key_pos.Ignition && gear == 3)` wahr ist.

Neben den Zuständen können State-Models auch *Transitionen* besitzen, welche Zustandsübergänge von einem State in einen anderen beschreiben und in der CGI-DSL mit dem Schlüsselwort `transitions` definiert werden. Zum Beispiel ist in Abbildung 3.4 eine Transition vom State `Normal` in den State `Reserve` definiert worden. Dieser Zustandsübergang wird dann aktiv, sobald die in Klammer angegeben Bedingung wahr ist.

```
167  statemodel Fuel_level
168      states {
169          state Normal
170          state Reserve
171          state Empty
172      }
173      transitions {
174          transition Normal -> Reserve cond [fuel_level < 9.0 && fuel_level > 1.0]
175          transition Reserve -> Normal cond [fuel_level > 11.0]
176          transition Reserve -> Empty cond [fuel_level <= 1.0]
177          transition Empty -> Reserve cond [fuel_level <= 11.0 && fuel_level > 2.0]
178      }
```

Abbildung 3.4: State-Model mit Transition

3.3 Schicht 3: UI-Models

Über User Interface – Models werden Signale über sogenannte Widgets dargestellt. Widgets werden dabei in Szenen definiert, welche mit dem Schlüsselwort `scene` erzeugt werden. Mit dem Schlüsselwort `bind` können Signale an Properties gebunden werden. Mit dem Schlüsselwort `event` kann ein Trigger ausgelöst werden, sobald die boolsche Bedingung des Triggers wahr ist. Ein einfaches UI-Model zeigt Listing 3.1.

```
uiModel
    scenes {
          scene Main_view {
                widget Speedometer
                    properties {
                      prop val: Num bind (speed)
                    }
                    events {
                        event alarm trigger [speed > 220]

                    }
          }
    }
```

Listing 3.1: Ein einfaches UI-Model

Die Scene `main_view` beinhaltet ein Widget `Speedometer`, welches eine Property `val` definiert. Das Ereignis `alarm` wird ausgelöst, sobald `speed` einen höheren Wert als 220 besitzt.

3.4 State-Models in XML

Zur einfachen und effizienten Persistierung von State-Models werden diese als XML-Dokumente gespeichert. Da State-Models aus verschachtelten States (OrStates) bestehen, ist XML ein ideales Datenformat. Ein Beispiel für ein State-Model, welches in XML dargestellt ist liefert Listing 3.1.

```
<Statemodel name="Coolant_temp">
  <State name="Cold"/>
  <State name="Warm"/>
  <State name="Hot"/>
  <State name="Critical"/>
    <Transition>
      <source>
        <State name="Critical"/>
      </source>
      <target>
        <State name="Cold"/>
      </target>
    </Transition>
    <Transition>
      <source>
        <State name="Hot"/>
      </source>
      <target>
        <State name="Cold"/>
      </target>
    </Transition>
    <Transition>
      <source>
        <State name="Cold"/>
      </source>
      <target>
        <State name="Warm"/>
      </target>
      <condition value="coolant_temp &gt;= 62.5 &amp;&amp; cool-
ant_temp &lt; 87.5"/>
      </Transition>
            .
            .
            .
            .
      <Transition>
        <source>
          <State name="Critical"/>
        </source>
        <target>
          <State name="Hot"/>
        </target>
        <condition value="coolant_temp &gt;= 82.5 &amp;&amp; cool-
ant_temp &lt; 92.5"/>
      </Transition>
</Statemodel>
```

Listing 3.2: Beispiel eines State-Models

Aus Platzgründen wurde nur ein Teil dieses State-Models abgebildet. Man erkennt schon an diesem kleinen Beispiel, wie groß und unübersichtlich ein State-Model in XML werden kann.

Die Teile eines State-Models werden im Folgenden näher erläutert, indem das State-Model aus Listing 3.1 analysiert wird.

- `<Statemodel name="Coolant_temp">` ist der Wurzelknoten und bezeichnet den Namen des State-Models.
- `<State name="Cold"/>` definiert einen State. Mit dem Attribut Name kann dem State ein Name zugewiesen werden. States können keine Child-States besitzen, dafür sind OrStates zuständig.
- `<Transition>` beschreibt einen Übergang von einem State in einen anderen State. <Transition> braucht dafür einen Source-State (angegeben mit dem Source-Tag) und einen Target-State (angegeben mit dem Target-Tag). Transitions können zwischen States und OrStates existieren.
- `<condition>` gibt an, welche Bedingungen eintreffen müssen, damit vom Source-State in den Target-State gewechselt werden kann. Das Attribut `value` legt dafür die Bedingung fest.
- `<OrState>` diese Art von State kann wiederum (muss aber nicht) States oder OrStates beinhalten. Durch die Verschachtelung ergibt sich die Baumstruktur des Graphen.

Abbildung 5.1 in Kapitel 5 zeigt, wie das State-Model aus Listing 3.1 in StateGraph-Viewer aussieht. Man kann deutlich erkennen, dass die grafische Darstellung sehr viel angenehmer anzusehen ist, als der lange und unübersichtliche Code aus Listing 3.1.

4 Verwendete Technologien

StateGraphViewer baut auf verschiedenen Java-Technologien, wie Zest oder Draw2d auf. Um die Arbeitsweise von StateGraphViewer besser verstehen zu können ist ein Grundwissen über diese Technologien nötig. Deswegen soll dieses Kapitel einen Überblick über die eingesetzten Technologien vermitteln.

4.1 Eclipse-Plattform

Eclipse wird oftmals als Integrated Development Environment (kurz IDE) für die Programmiersprache Java bezeichnet. Dies ist jedoch nicht ganz richtig, da es im Grunde „nur“ die Basis für Erweiterungen (Plugins) darstellt. Eclipse-Plugins werden im Kapitel 4.2 näher beschrieben.
Abbildung 4.1 gibt einen Überblick über die Eclipse Plattform.

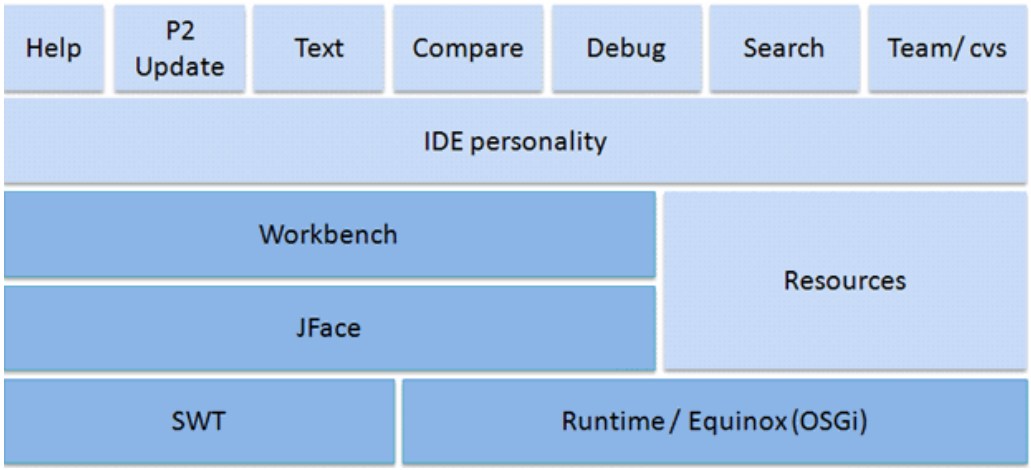

Abbildung 4.1: Eclipse Architektur [Vogel 2011]

An Abbildung 4.1 erkennt man, dass Eclipse modular aufgebaut ist. Dabei bilden die Komponenten die Kernteile und werden im Folgenden kurz beschrieben.

Runtime / Equinox

In der modernen Softwareentwicklung wird versucht, Komponenten so modular wie möglich zu gestalten. Deswegen hat die *OSGi Alliance* einen Standard gegründet, der genau dies ermöglichen soll. OSGi ist ein Java-Framework und stellt eine Laufzeitumgebung zur modularen Entwicklung von Anwendungen in Form von *Bundles* oder *Plugins* bereit. So können Module zur Laufzeit installiert, gestartet, gestoppt, aktualisiert und deinstalliert werden. Eclipse implementiert den OSGi Standard unter dem Namen *Equinox* [Arndt 2007].

SWT

Das Standard Widget Toolkit (kurz SWT) [SWT 2011] ist ein User-Interface-Toolkit für Java. Es stellt grundlegende grafische Komponenten (sogenannte *Widgets*) wie zum Beispiel Buttons oder Textfelder bereit. Im Gegensatz zu Swing sind die Komponenten von SWT vom jeweiligen Betriebssystem abhängig. Das heißt, dass SWT die nativen GUI Elemente des Betriebssystems nutzt. Aus diesem Grund werden die Widgets von SWT oft als „schwergewichtig“ bezeichnet.

JFace

Wie oben beschrieben stellt SWT „nur“ grundlegende User-Interface-Komponenten bereit. JFace [ECL 2011] bereinigt dieses Manko, indem es SWT mit weitergehenden Komponenten wie Wizards und Dialoge erweitert. Desweitern erleichtert es die User-

Interface-Programmierung, indem es Klassen für offen vorkommende Probleme bereitstellt:

- *Viewers* und *Provider*-Klassen stellen einen MVC-Ansatz für komplexere Controls bereit
- Actionframework zur allgemeinen Behandlung von Benutzeraktionen
- Image- und Font-Registries verwaltet Schrift- und Bildresourcen
- Dialoge and Wizards ermöglichen komplexe Interationen mit dem Benutzer

Wie schon erwähnt implementiert JFace das MVC-Modell in Form von Viewer- und Provider-Klassen [Ebert 2011]. Das Diagramm in Abbildung 4.2 soll einen kurzen Überblick geben, wie das MVC-Modell in JFace aussieht.

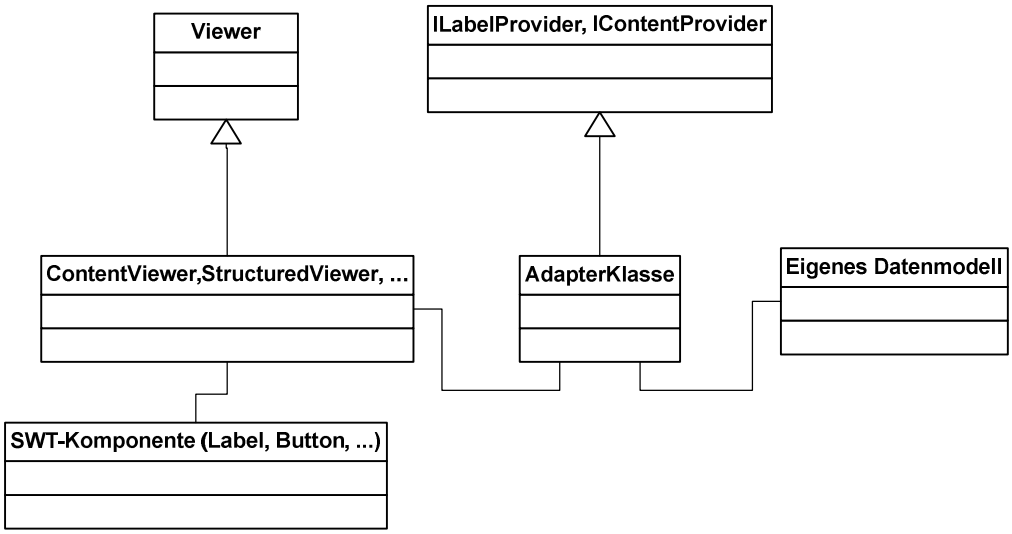

Abbildung 4.2: MVC-Prinzip in JFace

Es muss eine eigene Klasse implementiert werden, die die entsprechenden Provider-Interfaces implementiert [Kocher 2005]. In Abbildung 4.2 implementiert `AdapterKlasse` das `ILabelProvider`- und das `IContentProvider`-Interface. `IContentProvider` wird benötigt, um das eigene Datenmodell in ein Datenformat zu konvertieren, mit dem der `ContentViewer` umgehen kann. Mit `ILabelProvider` können Texte oder Images gespeichert werden, die dann von einem Label angezeigt werden. Die Viewer-Klassen sind Adapter für SWT-Widgets. Es gibt für jedes Widget eine Viewer-Klasse, mit dessen Hilfe die Daten der Provider dargestellt werden. Der

Vorteil an diesem Konzept ist, dass das bestehende Datenmodell verwendet werden kann, ohne es für die SWT-Widgets ändern zu müssen.

Wie und welche Viewer und Provider StateGraphViewer verwendet wird in Kapitel 5 genauer beschrieben.

Workbench

Ein Workbench ist der grafische Benutzerteil von Eclipse, mit dessen Hilfe der User mit Eclipse interagieren kann. Ein Workbench besteht aus grafischen Komponenten wie Views, Perspektiven und Texteditoren. Das Aussehen und Verhalten der Workbench kann vom Benutzer stark verändert werden. Zum Beispiel können die Toolbars verschoben, neue Perspektiven hinzugefügt, oder verschiedene Editoren installiert werden.

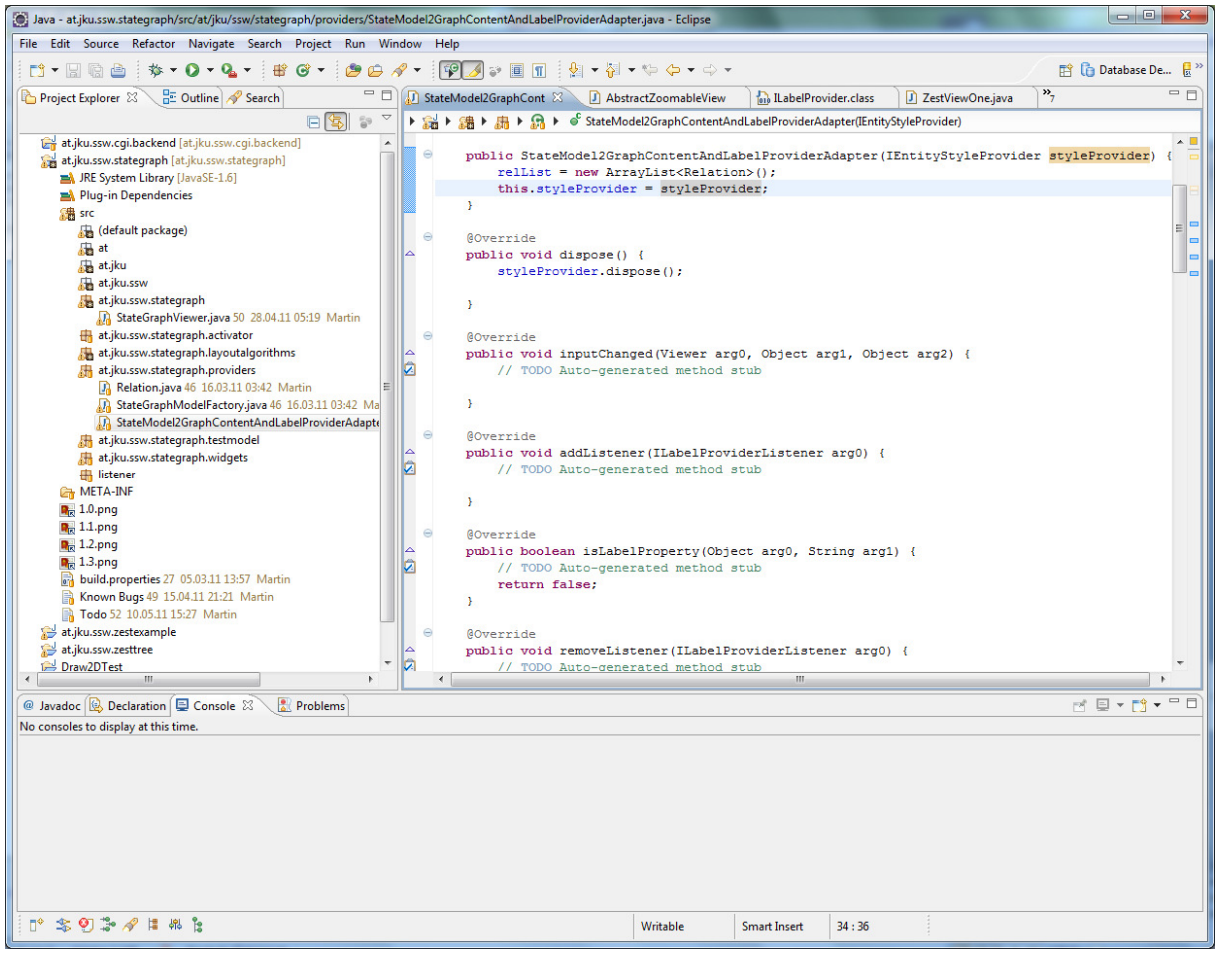

Abbildung 4.3: Der Eclipse Workbench

4.2 Eclipse-Plugins

Mithilfe von Plugins kann die Funktionalität von Eclipse erheblich erweitert werden. Zum Beispiel gibt es Plugins, um Programme in anderen Programmiersprachen wie zum Beispiel C++ oder Ruby in Eclipse zu entwickeln.

Die Erweiterung von Eclipse geschieht dabei über sogenannte *Extension-Points*. Eclipse, aber auch weitere Plugins können Extension-Points definieren, mit deren Hilfe Erweiterungen in Form von Plugins vorgenommen werden können.

4.3 Graphical Editing Framework

Das Bearbeiten von Datenmodellen geschieht wegen des erhöhten Komforts durch grafische Editoren. Für Eclipse wurde das *Graphical Editing Framework* (kurz GEF) [GEF1 2011] entwickelt, das es ermöglicht grafische Editoren in Eclipse zu bauen. Mittels GEF können folgende Aufgaben bewerkstelligt werden:

- das Erstellen von *Views* zur Bearbeitung von Datenmodellen
- das Einbinden von Datenmodellen
- Maus- und Tastaturunterstützung um die Views zu verändern
- MVC-Unterstützung, um das Datenmodell zu aktualisieren, wenn die View verändert wurde

GEF setzt sich aus drei Komponenten zusammen: Draw2d, GEF und Zest [GEF2 2011]. Draw2d und Zest wurden in dieser Arbeit verwendet und werden deswegen im Folgenden näher erklärt.

4.3.1 Draw2d

Draw2d [DRAW2D 2011] ist ähnlich wie Swing ein leichtgewichtiges Grafik-Toolkit. Leichtgewichtig bedeutet, dass die Komponenten von Draw2d komplett in Java geschrieben wurden und von den Betriebssystem-Ressourcen unabhängig sind. Der große Unterschied zwischen Draw2d und Swing ist jedoch, dass Draw2d rein für die Darstellung von grafischen Komponenten entwickelt wurde. Die MVC-Architektur von Swing gibt es in Draw2d nicht, deswegen können auch keine eigenen Models (wie zum Beispiel `TreeModel` für `JTree` in Swing) entwickelt werden. Draw2d wurde rein für die Darstellung entwickelt und nicht für die Haltung oder Manipulation von Daten.

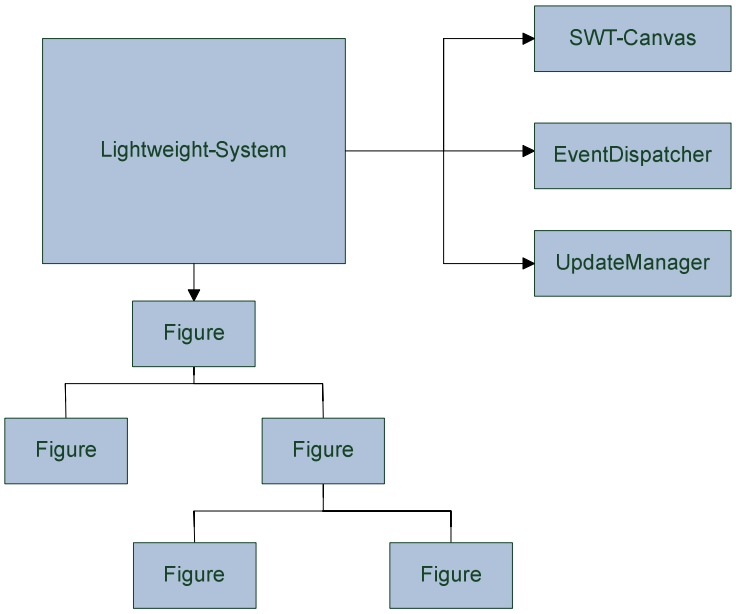

Abbildung 4.4: Die Struktur von Draw2d

Abbildung 4.4 gibt einen schematischen Überblick über das Draw2d-Toolkit. Mittels der Klasse `Figure` oder dem Interface `IFigure` lassen sich grafische Elemente erstellen. Wie Abbildung 4.4 zeigt, lassen sich diese Elemente ineinander schachteln, um komplexere Figures zu bilden. Das Leightweight-System verbindet eine Figure-Zusammensetzung mit einem SWT-Canvas, auf welches die Figures gezeichnet werden. An das Leightweight-System können außerdem SWT-Event-Listeners angehängt werden, diese werden abgesehen von *Paint-Events* an den *EventDispatcher* weitergeleitet. Paint-Events werden an den *UpdateManager* geschickt. Dieser ist bei Empfang von Paint-Events für das Neuzeichnen und für das Layout von Figures verantwortlich.

Desweiteren bietet Draw2d die Möglichkeit Figures miteinander zu verbinden. Mithilfe von Connections, die spezielle Figures sind, werden an den zwei Connection-Endpunkten sogenannte Anchors installiert. Anschließend werden Quell- und Ziel Figures mit einer Linie verbunden. Mithilfe von Connections können Figures zu Graphen oder Bäume verbunden werden. Kapitel 5.2.3 geht näher auf Connections in Bezug auf diese Arbeit ein.

4.3.2 Zest

Zest [ZEST 2011] ist ein Graphen-Visualisierungs-Tool, das speziell für Eclipse entwickelt wurde. Zest zeichnet Graphen mittels Draw2d und verwendet das Viewer-Konzept von JFace. Das bedeutet, dass schon vorhandene Provider- und Listener-Klassen in Zest verwendet werden können. Trotz der mächtigen Anwendungsmöglichkeiten ist Zest einfach zu verwenden. Listing 4.1 zeigt, wie schnell ein einfacher Graph mittels Zest erstellt werden kann.

```
public static void main(String[] args) {
            // Create the shell
            Display d = new Display();
            Shell shell = new Shell(d);
            shell.setText("GraphSnippet1");
            shell.setLayout(new FillLayout());
            shell.setSize(400, 400);

            Graph g = new Graph(shell, SWT.NONE);
            GraphNode n = new GraphNode(g, SWT.NONE, "State One");
            GraphNode n2 = new GraphNode(g, SWT.NONE, "State Two");
            GraphNode n3 = new GraphNode(g, SWT.NONE, "State Three");
            new GraphConnection(g, SWT.NONE, n, n2);
            new GraphConnection(g, SWT.NONE, n2, n3);
            new GraphConnection(g, SWT.NONE, n3, n);
            g.setLayoutAlgorithm(new
SpringLayoutAlgorithm(LayoutStyles.NO_LAYOUT_NODE_RESIZING), true);

            shell.open();
            while (!shell.isDisposed()) {
                  while (!d.readAndDispatch()) {
                        d.sleep();
                  }
            }
      }
```

Listing 4.1: Code zum Erzeugen eines einfachen Graphen

Die SWT-Klassen `Display` und `Shell` erzeugen ein leeres SWT-Fenster, in das die Zest-Graphen gezeichnet werden. Die Klasse `Graph` speichert die erzeugten Connections und Nodes. Dem Konstruktor von Graph wird das erzeugte SWT-Fenster übergeben. Da die Klasse `Graph` von der Draw2d Klasse `FigureCanvas` ableitet, ist sie für das Zeichnen aller Nodes und Connections (die Draw2d-Figures sind) verantwortlich. Mit der Klasse `GraphNode` können Knoten erzeugt werden und mit der Klasse `GraphConnection` können die Knoten mit einer Linie verbunden werden. Zest bietet verschiedene Layout-Algorithmen, um die Nodes und Connections auszurichten. Zum Beispiel versucht der Springlayout-Algorithmus, der in Listing 4.1 eingesetzt wird, die Nodes so anzuordnen, dass die Connection-Linien möglichst gleich lang sind. Abbildung 4.5 zeigt das Ergebnis.

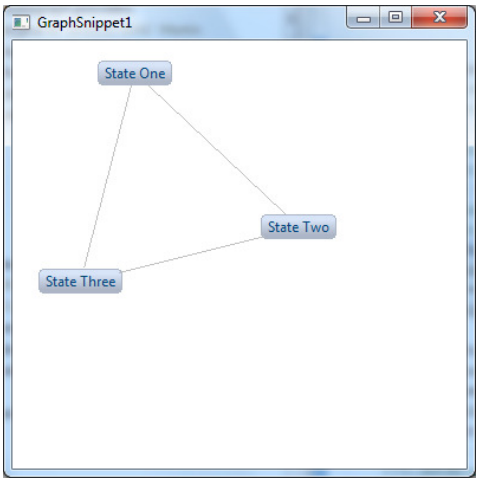

Abbildung 4.5: Ein einfacher Zest Graph

Folgende Layout-Algorithmen werden in Zest standardmäßig unterstützt:

- `TreeLayoutAlgorithm`: Stellt Graphen in Form eines Baums dar
- `HorizontalTreeLayoutAlgorithm`: Ähnlich dem TreeLayoutAlgorithm, nur dass die Knoten horizontal angeordnet werden
- `RadialLayoutAlgorithm`: Ein Knoten ist im Zentrum, alle anderen warden um diesen Knoten angeordnet
- `GridLayoutAlgorithm` : Ordnet Knoten nach einem Gittermuster an
- `SpringLayoutAlgorithm`: Alle Knoten sollen ungefähr die gleiche Länge haben und Kanten sollen sich minimal überlappen
- `HorizontalShift`: Überlappende Knoten warden nach rechts geschoben

- `CompositeLayoutAlgorithm`: Kombiniert Layout-Algorithmen miteinander. Zum Beispiel kann `HorizontalShift` angewendet werden, um Knoten nach rechts zu verschieben, nachdem ein anderer Layout-Algorithmus überlappende Knoten erzeugt hat.

5 StateGraphViewer

Nachdem im vorigen Kapitel eine Einführung in die verschiedenen verwendeten Technologien gegeben wurde, soll nun die eigentliche Arbeit vorgestellt werden.

5.1 State-Model-Visualisierung

StateGraphViewer wurde als Plugin für Eclipse entwickelt und kann als View zum Eclipse-Workbench hinzugefügt werden. StateGraphViewer verwendet und erweitert das in Kapitel 4.3.2 vorgestellte Zest-Framework, um die in Kapitel 3.1 erklärten State-Models grafisch als sogenannte State-Graphen darzustellen. StateGraphViewer stellt einen State-Graph als verschachtelte Rechtecke dar, wobei ein Rechteck einen State repräsentiert. Die Pfeile, die von einem Rechteck zu einem anderen reichen sollen die Connections repräsentieren. Das Besondere an StateGraphViewer ist, dass die Positionierung der Knoten und das Layout der Kanten automatisch erfolgt. Abbildung 5.1 zeigt einen einfachen Graphen, der das State-Model „Coolent_temp“ aus Listing 3.1 darstellt.

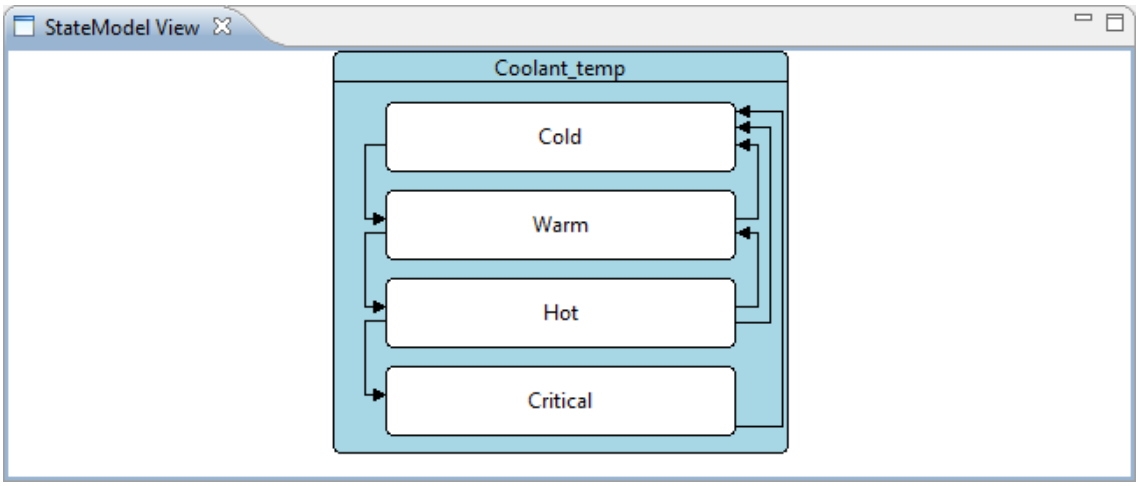

Abbildung 5.1: Ein einfacher State-Graph

5.1.1 Darstellungsdetails

Der grundsätzliche Aufbau eines State-Models im XML-Format wurde bereits in Kapitel 3.4 erklärt. Nun soll genauer gezeigt werden, wie StateGraphViewer State-Models grafisch in Form von State-Graphen darstellt. Dazu werden die Komponenten eines State-Graphen im Folgenden genauer analysiert.

Root-State

Der Root-State (Wurzelknoten) ist immer der größte Knoten und beherbergt alle anderen States, die seine Kinder sind. Im XML-Format wird der Root-State mit dem State-Model-Tag definiert. Wie alle anderen States hat der Root-State einen Namen. Dieser befindet sich wie bei OrState ganz oben. Im Root-State befinden sich weitere States, die entweder „normale“ States oder aber OrStates sein können.

OrState

OrStates können wiederum beliebig viele States oder OrStates beinhalten, jedoch kann ein OrState auch leer sein. Der Name der OrStates steht wie beim RootState oberhalb der Trennlinie in einem Label. Die Farbe der OrStates werden in StateGraphViewer heller, je tiefer sie verschachtelt werden. Abbildung 5.2 soll das verdeutlichen: Der

Root-Node *Key-Pos* ist am dunkelsten. *OrState1111* liegt zum Vergleich tiefer in der Hierarchie und hat deswegen eine hellere Farbe. Durch dieses Farbschema sollen sich die States besser voneinander abheben. Die Übersichtlichkeit soll dadurch erhöht werden.

State

Die tiefsten Knoten in der Baumhierarchie eines State-Graphen bilden die States. States können anders als OrStates keine weiteren States beinhalten. States werden in StateGraphViewer immer als weiße Rechtecke repräsentiert. Das Label für den Namen wird mittig im Rechteck dargestellt.

Transitionen

Wie Kapitel 2.2 schon gezeigt hat, ist es bei der Darstellung eines Zustandsautomaten üblich, Zustandsüberführungen durch Pfeile darzustellen. Auch StateGraphViewer visualisiert Transition, die in einem State-Model definiert wurden, durch Pfeile. Wie in Kapitel 3.4 schon erwähnt, besitzt jede Transition einen Source- und einen Target-State. Die Target-States werden in StateGraphViewer durch eine Pfeilspitze markiert. Transitionen, bei denen sich der Target-State oberhalb des Source-States befindet, werden auf der rechten Seite des Graphen dargestellt. Genau spiegelverkehrt wird mit Transitionen verfahren, bei denen sich der Target-State unterhalb des Source-States befindet. Diese Transitionen werden auf der linken Seite des Graphen gezeichnet. Abbildung 5.2 zeigt zum Beispiel eine Transition zwischen *State2* und *Starting*. Die Pfeilspitze, die auf *Starting* zeigt, deutet darauf hin, dass *Starting* der Target-State sein muss. Folglich kann daraus geschlossen werden, dass *State2* der Source-State ist. *Starting* befindet sich oberhalb von *State2*. Das bedeutet, dass der Transitionspfeil auf der rechten Seite gezeichnet wird.

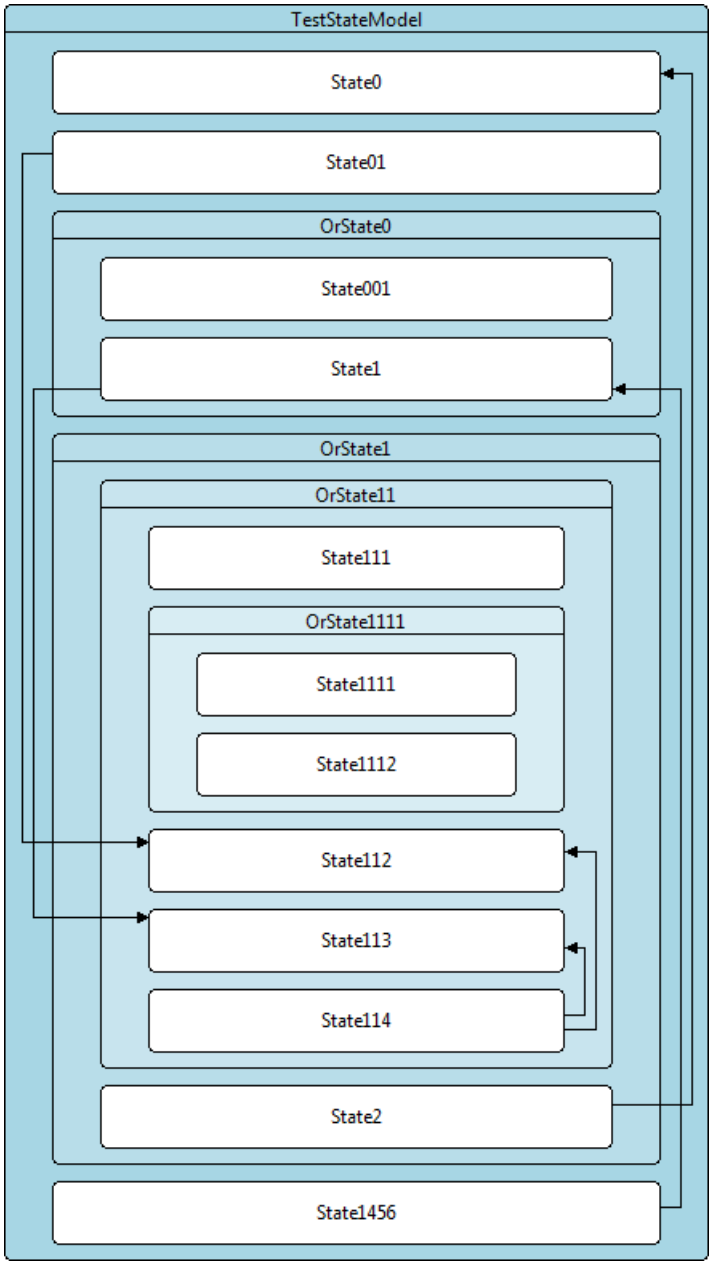

Abbildung 5.2: Visualisierung eines State-Models mit StateGraphViewer

5.1.2 Selektion

StateGraphViewer erlaubt die Selektion von States und von Connections mit der Maus. Andere Plugins können dadurch auf Selektionen des Benutzers horchen und diese Selektionen dann weiterverarbeiten. Wenn ein State oder ein OrState selektiert wird, dann wird der Rahmen des betreffenden States dicker dargestellt. Als Beispiel dient

der State *Steering_free_ignition_off*. Das linke Bild zeigt den State, wenn er nicht selektiert ist und das Rechte zeigt den State, wenn er mit der Maus selektiert wurde.

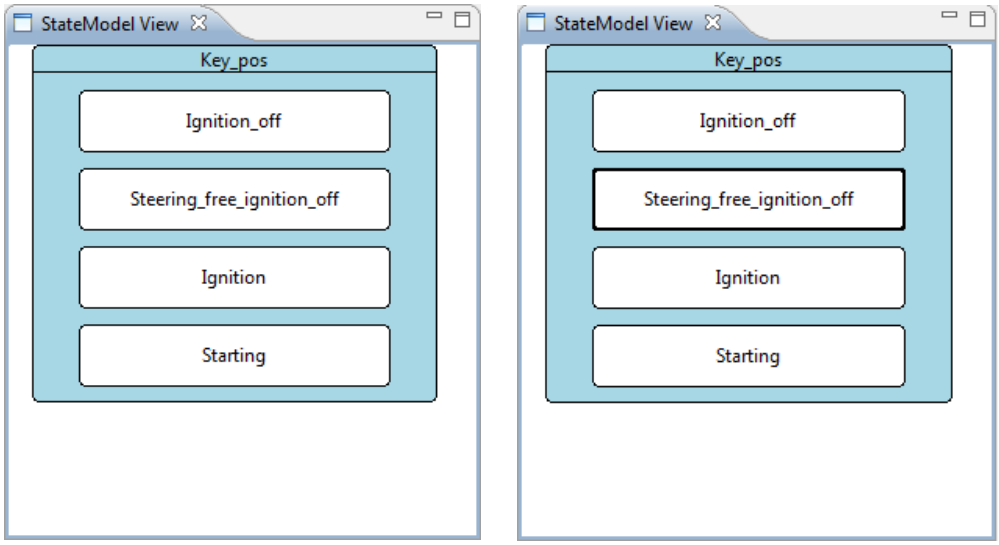

Abbildung 5.3: Visualisierung Links der nicht selektierte State und rechts der Selektierte

Nicht nur States, sondern auch Transitionspfeile können selektiert werden. Wenn ein Transitionspfeil mit der Maus selektiert wird, dann ändert der Pfeil seine Farbe von Schwarz zu Rot. Abbildung 5.4 zeigt als Beispiel den gleichen Graphen wie Abbildung 5.3, jedoch diesmal mit Transitionspfeilen. Wie man erkennen kann, wurde der Selektionspfeil von *Hot* zu *Cold* selektiert, da dieser rot gefärbt ist.

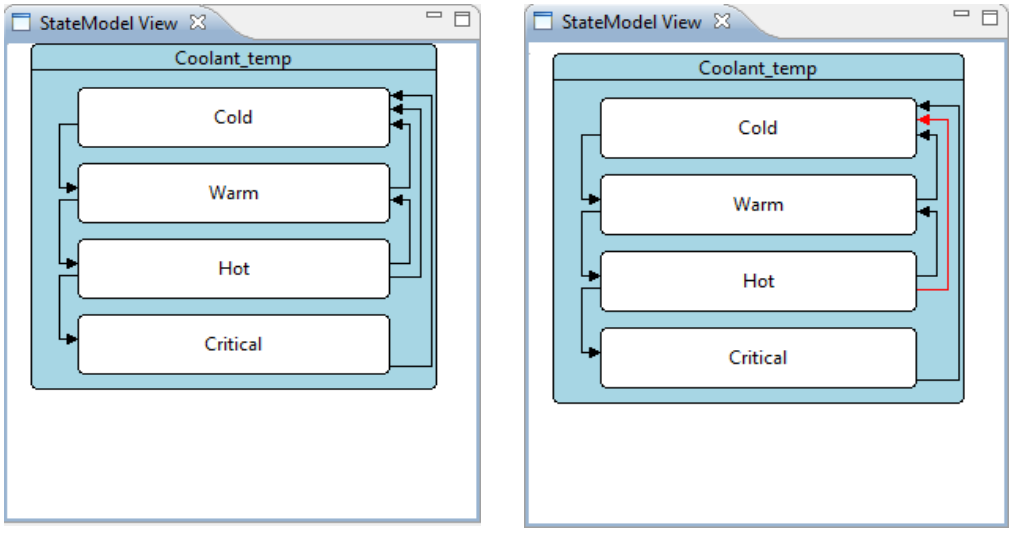

Abbildung 5.4: Links der nicht selektierte Transitionspfeil und rechts der Selektierte

5.1.3 Tooltips

Transitionen besitzen neben Source- und Target-States auch *Conditions*. Conditions sind im Grunde nichts anderes als Bedingungen, die entweder wahr oder falsch sind. Conditions werden in State-Models mit dem Condition-Tag definiert. Zum Beispiel besitzt in Listing 5.1 die Transition von *Cold* nach *Warm* folgende Condition:

```
<condition value="coolant_temp &gt;= 62.5 &amp;&amp; coolant_temp &lt; 87.5"/>
```

Listing 5.1: Condition der Transition von Cold nach Warm

Die Transitionspfeile in StateGraphViewer sind zu schmal, um alle Conditions auf einmal anzuzeigen. Desweiteren würde ein State-Graph sehr unübersichtlich werden, wenn viele Conditions im zugehörigen State-Model vorkommen. Deswegen werden Conditions in StateGraphViewer mittels Tooltips angezeigt. Wenn sich der Mauszeiger über einen Transitionspfeil befindet, erscheint ein kleines gelbes Rechteck, das die Condition der Transition anzeigt. Abbildung 5.5 soll dies verdeutlichen: Die Condition aus Listing 5.1 wird als Tooltip angezeigt, sobald sich der Mauszeiger über die dazugehörige Transition befindet.

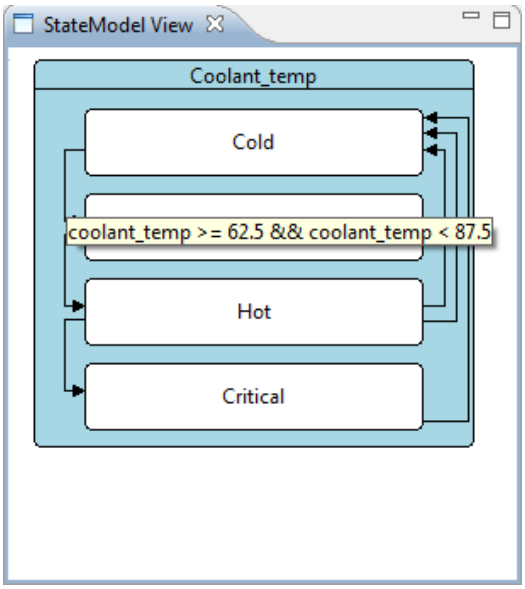

Abbildung 5.5: Tooltip der Transition aus Listing 4.1

Wenn der Mauszeiger den Transitionspfeil verlässt verschwindet der Tooltip wieder. Dadurch ist ein einfaches Navigieren über Transitionen möglich und der Graph bleibt übersichtlich.

XML-spezifische Sonderzeichen wie zum Beispiel `>=` werden, wie man in Abbildung 5.5 erkennt, zu normalen Text konvertiert.

Tooltips werden nicht nur für Transitionen angezeigt. Auch States und OrStates besitzen Tooltips, die den Namen der States anzeigen.

5.2 Layouting

In *Kapitel 3.4* wurde gezeigt, aus welchen Elementen State-Graphen bestehen. Nun soll gezeigt werden, wie die einzelnen States eines Graphen skaliert und positioniert werden. Die Darstellung von State-Models in StateGraphViewer erfordert spezielle Algorithmen, um die States und Transitionen zu layouten. Gerade bei sehr komplexen State-Models sind robuste Algorithmen wichtig, damit das Layout der State-Graphen korrekt dargestellt wird. StateModelViewer implementiert Algorithmen, die für die Größe und Position der States, sowie Algorithmen, die für die Berechnung der Länge und Ausrichtung der Transitionspfeile zuständig sind. Im Folgenden sollen diese Algorithmen genauer besprochen werden.

5.2.1 Baumdarstellung

StateGraphViewer verwaltet die States der State-Models intern in einer Baumstruktur. Wie schon erklärt bestehen State-Models aus einfachen States und OrStates, welche weitere einfache States oder OrStates enthalten können. Aus diesem Grund sind State-Models Bäume und lassen sich deswegen ideal in einer Baumstruktur verwalten.

Anmerkung: Wenn im Folgenden von States gesprochen wird, dann sind damit einfache States sowie OrStates gemeint. Wenn eine dieser zwei Typen speziell angesprochen wird, dann wird dies explizit erwähnt.

Abbildung 5.6 soll einen einfachen Baum zeigen, der das State-Model des State-Graphen aus Abbildung 5.2 darstellt. Der Baum besteht aus 4 Ebenen. Die Wurzel des Baumes in der obersten Ebene ist der Root-State. Die inneren Knoten repräsentieren OrStates und die Blätter des Baumes stellen einfache States dar.

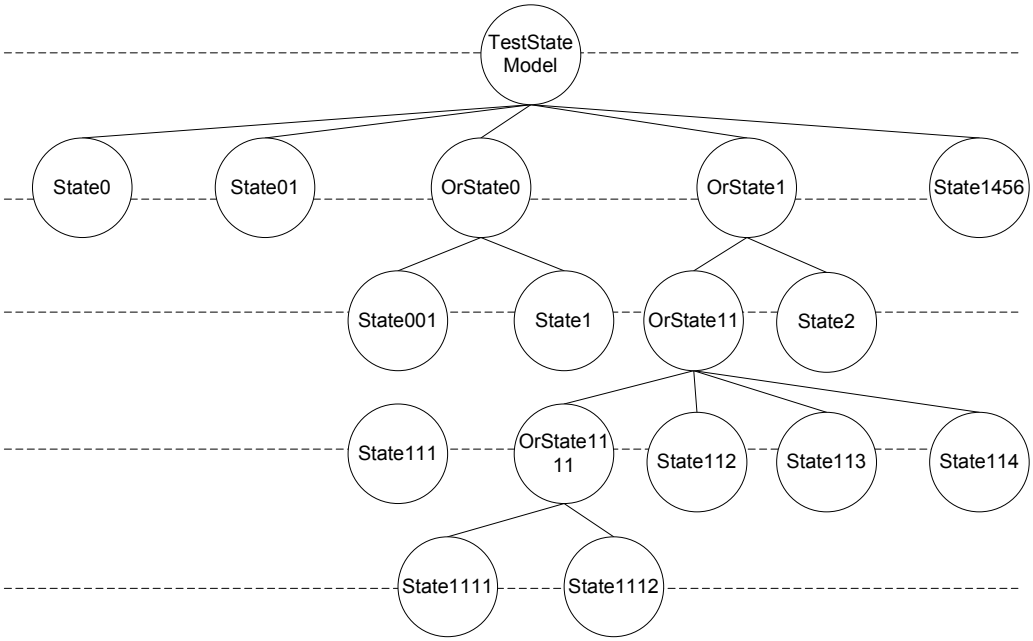

Abbildung 5.6: Baumdarstellung eines State-Models

Im State-Graphen in Abbildung 5.2 beherbergt der Wurzelknoten *TestStateModel* alle anderen States und ist deswegen auch visuell gesehen der größte Knoten. *TestStateModel* hat 5 Kinder: *OrState0, State01, OrState0, OrState1 und State1456*. OrStates können weitere States kapseln. *OrState0* kapselt zum Beispiel die States *State001* und *State1*. Im Baum sind diese zwei States Kinder vom Knoten OrState0 und werden im State-Graphen aus Abbildung 5.2 als zwei weiße Rechtecke dargestellt, die sich innerhalb von *OrState0* befinden. Für alle anderen Knoten des Baumes aus Abbildung 5.6 wird nach demselben Prinzip verfahren.

5.2.2 Algorithmen für States

Die Aufgabe der folgenden Algorithmen ist es, die Position und Größe jedes einzelnen States in der Eclipse-View zu bestimmen. Der Algorithmus zur Darstellung der States arbeitet in mehreren Schritten, wobei jeder Schritt rekursiv über die Baumstruktur traversiert:

1 Bestimme die Höhe und die Y Position jedes Knoten

2 Bestimme die Breite jedes Knoten

3 Bestimme die X-Position jedes Knoten

Bevor alle 3 Schritte näher erläutert werden, sollen das Grafiksystem von SWT sowie einige wichtige Parameter von StateGraphViewer genauer erklärt werden.

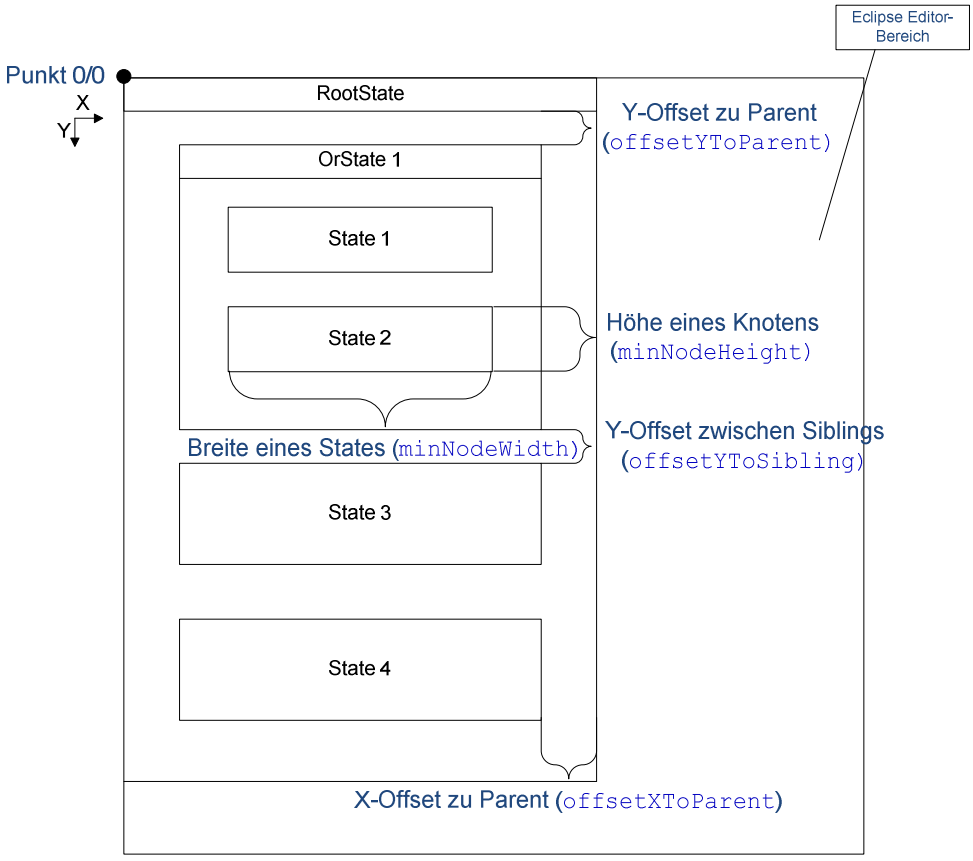

Abbildung 5.7: Darstellungseigenschaften eines State-Graphen

Wie bei den meisten Grafiksystemen ist es auch bei SWT so, dass die vertikale Y-Achse nach unten und die horizontale X-Achse nach rechts zeigen. Der Nullpunkt des Koordinatensystems befindet sich in der linken, oberen Ecke des Graphen und ist als

schwarzer Punkt eingezeichnet. Die Graphen werden von dort aus gezeichnet und erst ganz zum Schluss in die Mitte des Editor-Bereiches verschoben.

Für das Layouten der Zustände wurden mehrere Offsets definiert, welche in Abbildung 5.7 zu sehen sind. Die eingezeichneten Offsets sind konstante Werte, die einen fixen Abstand zwischen zwei States herstellen. Insgesamt gibt es 5 solcher Konstanten:

- `offsetXToParent`: Stellt einen vertikalen Abstand zu einen Eltern-State her
- `offsetYToParent`: Stellt einen horizontalen Abstand zu einem Parent-State her.
- `offsetYToSibling`: Stellt einen horizontalen Abstand zwischen 2 States her, die direkt untereinander stehen und den selben Parent haben
- `minNodeWidth`: Gibt die minimalste Breite an, die ein State haben kann.
- `minNodeHeight`: Gibt die minimalste Höhe an, die ein State haben kann

Um die Algorithmen bildlich darzustellen, wird der State-Graph aus Abbildung 5.7 schrittweise aufgebaut. Die Transitionspfeile dieses State-Graphen werden im nächsten Kapitel hinzugefügt.

Bestimmung der Höhe und der Y-Position

Die Berechnung der Höhe und der Y-Position der States werden in einem Algorithmus durchgeführt, da die Y-Position eines States von der Höhe der darüber liegenden States abhängt. Das bedeutet, dass die Y-Position eines States nicht bestimmt werden kann, wenn die Y-Positionen und die Höhen der darüber liegenden States nicht bekannt sind. Deswegen ist es sinnvoll und auch notwendig, diese zwei Schritte zu Einem zusammenzufassen.

Folgender Pseudocode soll zeigen, wie der Algorithmus arbeitet.

```
Annahmen:
- Die Baumstruktur ist wohlgeformt
- Die Baumstruktur besteht aus mindestens einem Element
- pointer ist vor erstem Aufruf 0

Input: Wurzelknoten der Baumstruktur
```

```
1:  function YPositionAndHeight(node, *pointer)
2:    stateHeight := 0
3:    for i := 0 to node.getNumberOfChildren() do
4:      gapHeight := 0
5:      child := node.getChildFromPosition(i)
6:      if (i == 0) then
7:        gapHeight := child.Y_Offset_To_Parent
8:      else
9:        gapHeight := child.offsetYToSibling
10:       pointer += gapheight
11:       stateHeight += gapHeight
12:       child.setYPositionOfNode(pointer)
13:       stateHeight += YPositionAndHeight(child, &pointer)
14:     end if
15:   end for
16:   if (node.getNumberOfChildren() == 0) then
17:     node.setStateHeightOfNode(minNodeHeight)
18:     stateHeight = minNodeHeight
19:     pointer += minNodeHeight
20:   else
22:     n := node.getChildFromPosition(0)
23:     vy := stateHeight + n.Y_Offset_To_Parent
24:     node.setStateHeightOfNode(vy);
25:     stateHeight = vy
26:     pointer += n.Y_Offset_To_Parent
27:   end if
28:   return stateHeight
29: end function
```

Listing 5.2: Pseudocode für den Höhenbestimmungs-Algorithmus

Die Arbeitsweise des Algorithmus, soll bildlich anhand der Abbildungen 5.8 bis 5.13 erklärt werden. Jede der folgenden Abbildungen hat ein rechtes und ein linkes Kästchen. Die linken Kästchen bauen den State-Graphen aus Abbildung 5.7 nach und nach auf, im rechten Kästchen befindet sich die zugehörige Baumstruktur des State-Graphen.

Der gerade besuchte Knoten und der dazugehörige State sind rot umrahmt. Die gestrichelten Linien sollen andeuten, dass die Breite oder Höhe eines States noch unbekannt ist und erst zu einem späteren Zeitpunkt berechnet wird. Um dieses Beispiel möglichst einfach zu halten, werden die Labels der States nicht dargestellt.

Abbildung 5.8 zeigt den Initialzustand des Algorithmus. Der große, schwarze Pfeil soll die Variable `pointer` aus Listing 5.2 darstellen. Der Pfeil wandert von oben nach unten, wobei er am Anfang an der Y-Position 0 steht. Die X-Positionen der States sind für den Zeiger komplett unerheblich, da dieser Algorithmus nur für die Y-Position und

Höhe der States zuständig ist. Die Y-Position des Root-States muss außerdem nicht berechnet werden, weil dieser immer bei Y = 0 positioniert wird.

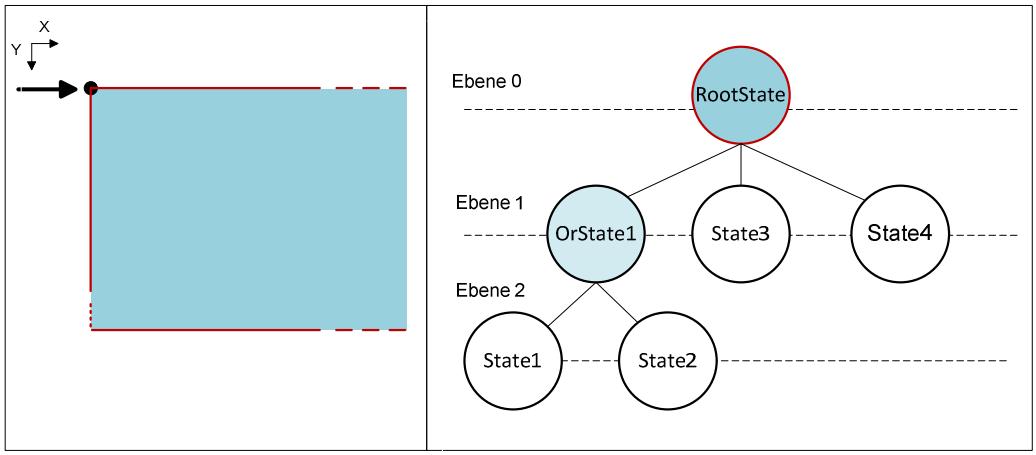

Abbildung 5.8: Layouting des Root-States

Nachdem das erste Kind von *RootState* ausgewählt wurde (Zeile 5 in Listing 5.2), wird die Y-Position für diesen State berechnet. Wie weiter oben schon erklärt gibt es verschiedene Offsets, um einen Abstand zwischen zwei States herzustellen. Da dies der erste (der am weitesten links stehende) State auf Ebene 1 ist, muss das Offset zum Parent genommen werden (Zeile 7). Das Offset wird zu `pointer` addiert (Zeile 10) und der Pfeil wandert um diesen Wert nach unten (Abbildung 5.9).

Die Höhe jedes States wird in der Variable `stateHeight` geführt. Die Höhe eines OrStates setzt sich aus der Gesamthöhe aller Kinder dieses States und den dazwischen liegenden Offsets zusammen. Das erste Offset wurde schon berechnet und kann zu `stateHeight` addiert werden (Zeile 11). Die Höhe von einfachen States ist ein fixer Wert (`minNodeHeight`) und muss somit nicht berechnet werden.

Nun kann die Y-Position des ersten States Or*State1*, welcher hier ein OrState ist, fixiert werden (Zeile 12). Die Funktion `setYPositionOfNode(node, pointer)` setzt den State `node` an die durch `pointer` angegebene Position. Die gestrichelte Höhe von *OrState1* soll andeuten, dass die Höhe von diesem State noch nicht bekannt ist. Diese wird sich im Folgenden gleich wie bei *RootState* durch seine Kinder ergeben.

Zeile 14 führt den rekursiven Aufruf durch. Der Funktion werden als Parameter das aktuelle Kind (*OrState1*) und die aktuelle Position des Pointers übergeben. Der Baum wird im Depth-First Verfahren durchlaufen. Das bedeutet, dass der Knoten, der am weitesten links in einer Ebene steht als erstes gezeichnet wird und sich somit im State-

Graphen über seinen Geschwistern befindet. Zum Beispiel erkennt man in Abbildung 5.9, dass der OrState *OrState1* vor *State3* und *State4* gezeichnet wird, weil sich der Knoten im Baum weiter links wie *State3* und *State4* befindet.

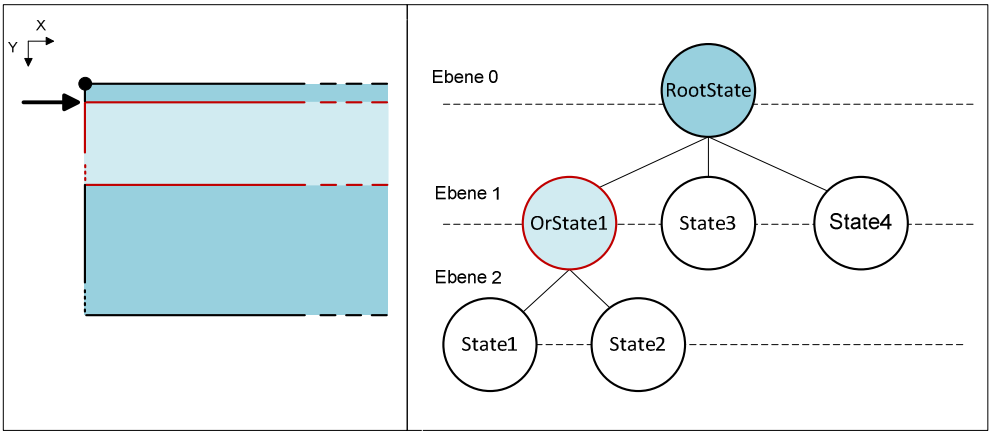

Abbildung 5.9: Layouting des Or-States

Nach dem rekursiven Aufruf wird *OrState1* zum neuen Root. Nun müssen *State1* und *State2* positioniert werden. Wegen des rekursiven Aufrufes ist das Vorgehen des Algorithmus gleich wie zuvor:

- der Pointer wird um den Offset hinunter geschoben
- zur Variable `stateHeight` wird dieser Offset dazu addiert
- das erste Kind von Or*State1* wird an die Stelle positioniert, wo der Pointer hinzeigt
- der nächste rekursive Aufruf folgt, *State1* wird zum neuen Root

Abbildung 5.10 zeigt, wie der Graph nach den Schritten 1-3 aussieht.

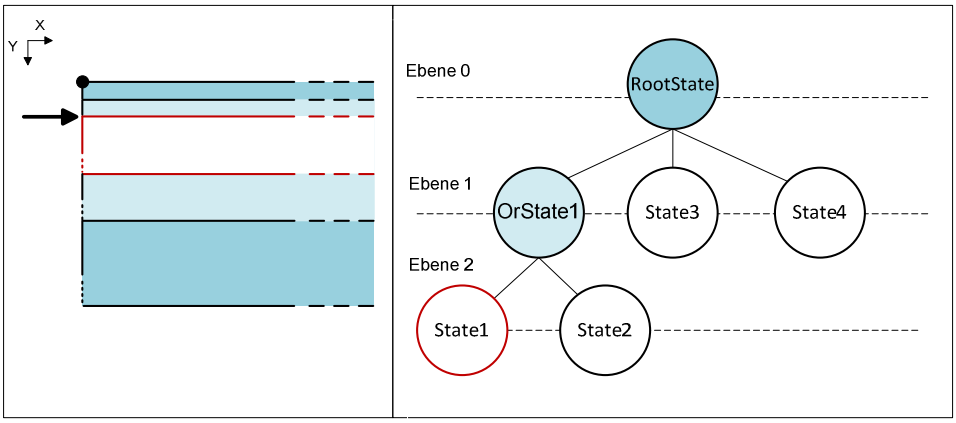

Abbildung 5.10: Positionierung des ersten Einfachen States

Der nächste rekursive Aufruf wurde gemacht und *State1* ist der neue Root. *State1* ist ein einfacher State und hat somit keine Kinder. Deswegen wird die for-Schleife nicht ausgeführt und der Algorithmus macht bei Zeile 16 weiter. Der Code ab Zeile 16 bis zum Schluss hat im Grunde nur die Aufgabe, die Höhe eines States zu berechnen. Dabei wird wieder unterschieden, ob der aktuelle Root ein einfacher State ist (Zeile 16 prüft das) oder ob es sich um einen OrState handelt. Im Falle eines einfachen States, wird die zuvor schon erwähnte Konstante `minNodeHeight` dem einfachen State zugewiesen (Zeile 17).

State1 ist ein einfacher State und bekommt deswegen die Höhe `minNodeHeight`. Der Pointer wird ebenfalls um diesen Wert erhöht und zeigt somit auf den unteren Rand von *State1* (Abb. 5.11). Danach wird der rekursive Aufruf beendet und die Höhe von *State1* (in diesem Fall `minNodeHeight)` zurückgegeben.

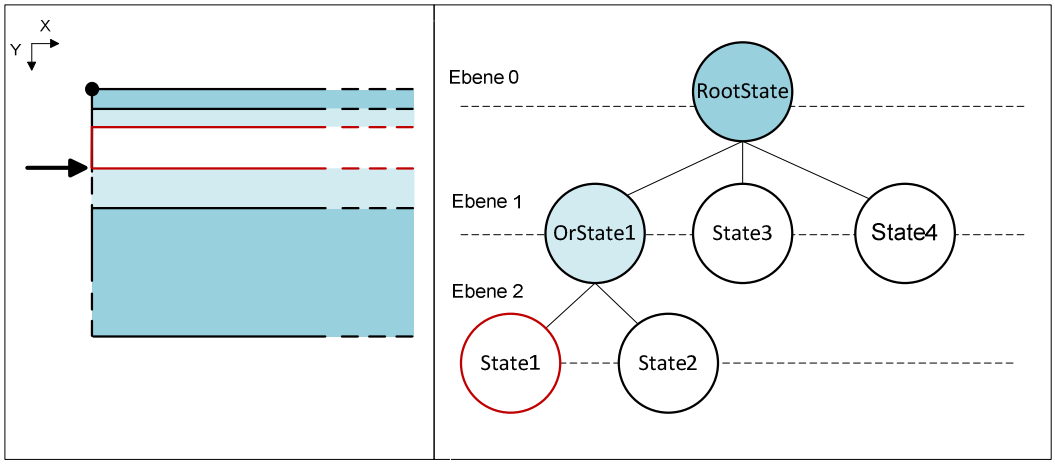

Abbildung 5.11: Setzen des Pointers

Als nächstes wird *State2* gelayoutet. Das Prinzip ist wieder das Selbe wie bei *State1* und wird deswegen nicht weiter erläutert. Der einzige Unterschied zu *State1* ist nur, dass nun ein anderes Offset genommen wird, nämlich `offsetYToSibling` (Zeile 9). Die Höhe von *State2* wird nach dem rekursiven Aufruf zurückgegeben und zur Höhe von *OrState1* addiert. Nun fehlt für die komplette Höhe von *OrState1* nur noch der Offset zwischen *State2* und dem OrState selbst. Dieser wird in Zeile 22 zur aktuellen Höhe des OrStates addiert. In Zeile 25 wird das gleiche Offset noch zum Pointer addiert, dieser zeigt nun auf den unteren Rand des OrStates (Abbildung 5.12).

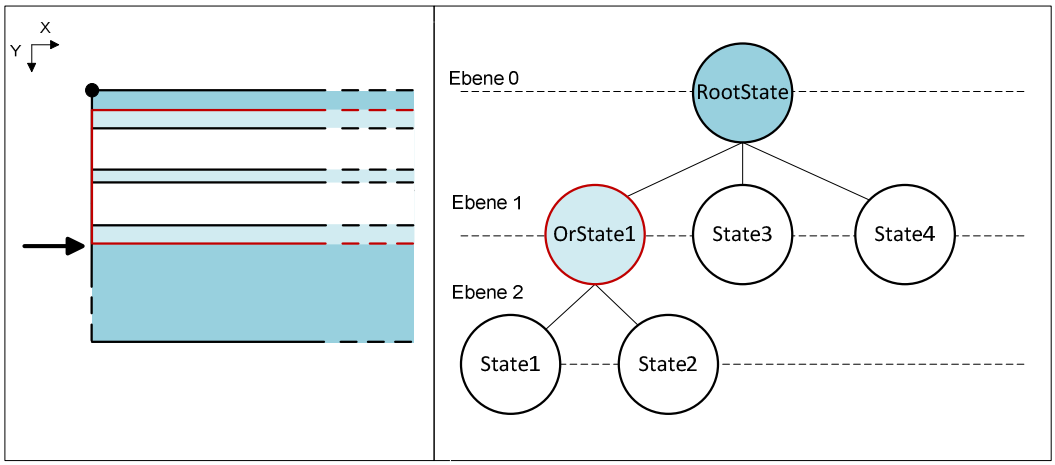

Abbildung 5.12: Ende der Positionierung des Or-States

Nun müssen noch die Knoten *State3* und *State4* gelayoutet werden. Der Vorgang ist der Gleiche wie bei den zwei States *State1* und *State1* und werden deswegen nicht weiter erläutert. Das Ergebnis des Layout-Vorganges zeigt Abbildung 5.13.

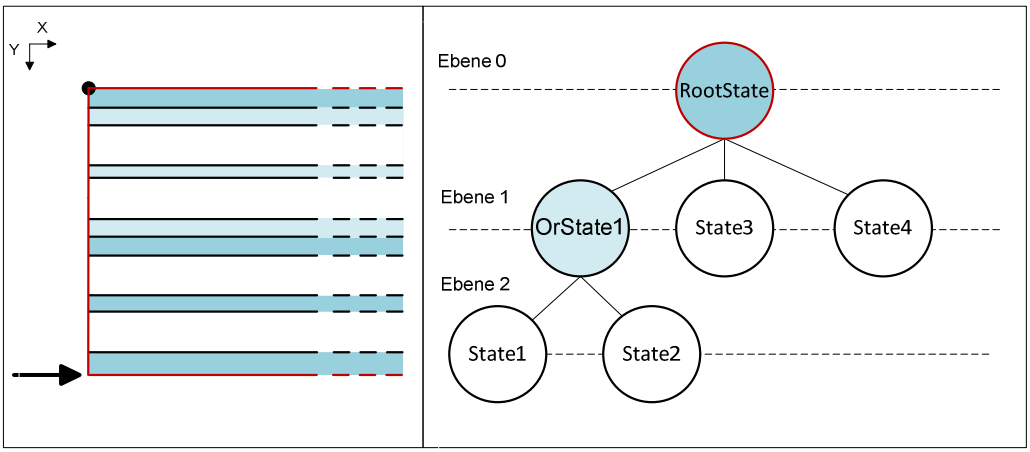

Abbildung 5.13: Höhe des State-Graphen fertig gelayoutet

Die Höhen und Y-Positionen aller Knoten sind nun bekannt. Wie man jedoch durch die gestrichelten, horizontalen Linien erkennen kann, sind die Breiten der States noch nicht bekannt. Der nächste Layout-Schritt bewerkstelligt genau diese Aufgabe.

Bestimmung der Breite

Der erste Layout-Schritt hat jeden State eine Y-Position und eine Höhe zugewiesen. Dieser Schritt soll nun die Breite der States berechnen, damit im dritten und letzten Layout-Schritt die X-Position festgelegt werden kann.

Listing 5.3 zeigt zuerst den Pseudocode für die Breitenbestimmung. Dieser wird dann anschließend anhand von Abbildungen erklärt, wobei das Beispiel aus den Abbildungen 5.8 – 5.13 weitergeführt wird.

```
Annahmen:
- Die Baumstruktur ist wohlgeformt
- Die Baumstruktur besteht aus mindestens einem Element
- level ist vor erstem Aufruf 0

Input:
- Wurzelknoten der Baumstruktur
- Die Ebene, auf der sich ein Blatt des Baumes befindet
```

```
1:  function setWidth(node, level)
2:    biggestNodeWidth := 0
3:    for i := 0 to node.getNumberOfChildren() do
4:      child := node.getChildFromPosition(i)
5:      width := setWidth(child, level++)
6:      if(width > biggestNodeWidth) then
7:        biggestNodeWidth := width
8:      end if
9:    end for

10:   if(biggestNodeWidth == 0) then
11:     biggestNodeWidth := minNodeWidth
12:   end if

13:   for i := 0 to node.getNumberOfChildren() do
14:     child := getChildFromPosition(node, i)
15:     realWidth = biggestNodeWidth - 2*offsetXToParent
16:     adjustNonLeaveNode(child, realWidth)
17:   end for

18:   node.setStateWidthOfNode(biggestNodeWith)
19:   return biggestNodeWith + 2*offsetXToParent
12:  end function

20:  function adjustNonLeaveNode(node, biggestNodeWidth)
21:    for i := 0 to getNumberOfChildren(node) do
22:      child := node.getChildFromPosition(i)
23:      with := biggestNodeWidth - 2*offsetXToParent
24:      adjustNonLeaveNode(child, width)
25:    end for
26:    node.setStateWidthOfNode(biggestNodeWidth)
27:  end function
```

Listing 5.3: Pseudocode zur Bestimmung der Breite der States

Der Algorithmus teilt sich in zwei Funktionen auf: `setWidth()` und `adjustNonLeaveNode()`. Die Funktion `setWidth()` hat die Aufgabe für jeden State eine Breite zu berechnen und diese den States zuzuweisen. Gleich wie die Funktion `YPositionAndHeight() aus` Listing 5.2 wird auch hier der Baum rekursiv nach dem Depth-

First Prinzip durchlaufen.

Die Variable `biggestNodeWidth` speichert die längste Breite eines States in einer Ebene des Baumes. Alle Knoten in einer Ebene sind zum Schluss so breit, wie es `biggestNodeWidth` vorgibt. Knoten in einem Baum, die in einer Ebene stehen und schmäler als `biggestNodeWidth` sind, müssen im Nachhinein die Größe `biggestNodeWidth` erhalten. Aus diesem Grund ist ein erneuter rekursiver Durchlauf

des Baumes ab dieser Ebene notwendig. Dafür ist die Funktion `adjustNonLeaveNode()` zuständig, die die Breite der States mit einer neuen Breite überschreibt.

Die folgenden Abbildungen sollen den Algorithmus anschaulich zeigen.

Die Funktion `setHeight()` wurde initial mit dem Wurzelknoten *RootState* aufgerufen. Der zweite Parameter `level` von `setHeight()` ist 0. Die Funktion traversiert nun rekursiv über die Knoten *RootState* und *OrState1* bis zum Blatt-Knoten *State1*. Da *State1* keine Kinder hat, wird die for-Schleife in Zeile 3 nicht mehr betreten und somit kein rekursiver Aufruf mehr durchgeführt. Die Variable `biggestNodeWith` ist 0, deswegen erhält diese den Wert der Konstante `minNodeWith` (Zeile 11). Die for-Schleife in Zeile 13 wird nicht betreten, da *State1* keine Kinder hat. Die Breite von *State1* wird gesetzt (Zeile 18) und diese breite plus zweimal den Abstand (`offsetXToParent`) zum Parent zurückgegeben (Zeile 19). Zweimal deswegen, da auf der rechten und linken Seite ein Abstand zum Parent sein soll.

Anschließend wird die Breite von *State2* auf die gleiche Weise berechnet. *State2* erhält gleich wie *State1* die Breite `biggestNodeWith` (Abbildung 5.14).

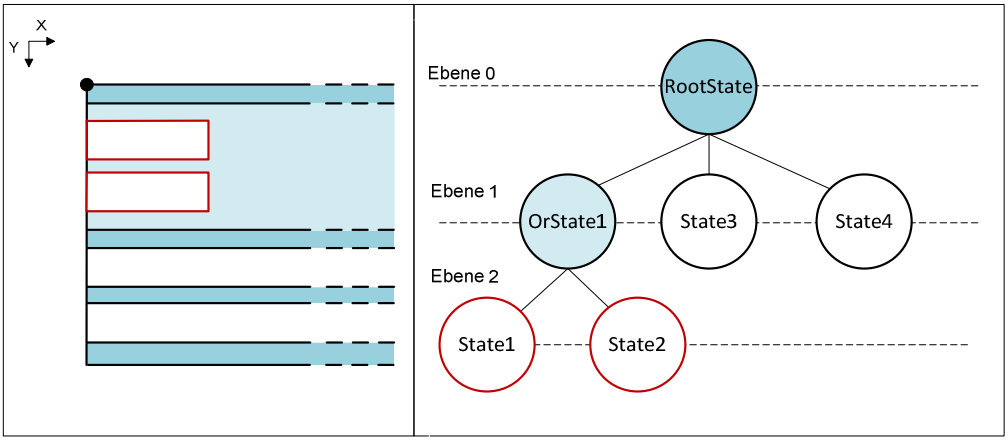

Abbildung 5.14: Breite der einfachen States ist bekannt

Der Funktionsaufruf hat im Rückgabewert die Breite für *OrState1* in der Variable `width` gespeichert. Die Abfrage in Zeile 6 ist wahr, da `biggestNodeWith` noch 0 ist. Die Variable `biggestNodeWith` erhält somit die Breite, die die Kinder von *OrState1* zuvor berechnet haben. Die for-Schleife in Zeile 13 wird nun betreten und die Funktion `adjustNonLeaveNode()` aufgerufen. Jedoch hat der Aufruf keine Auswirkung. Gleich wie vorher wird nun die Breite von Or*State1* gesetzt und die Breite plus den Abstand zum Parent zurückgegeben (Abbildung 5.15).

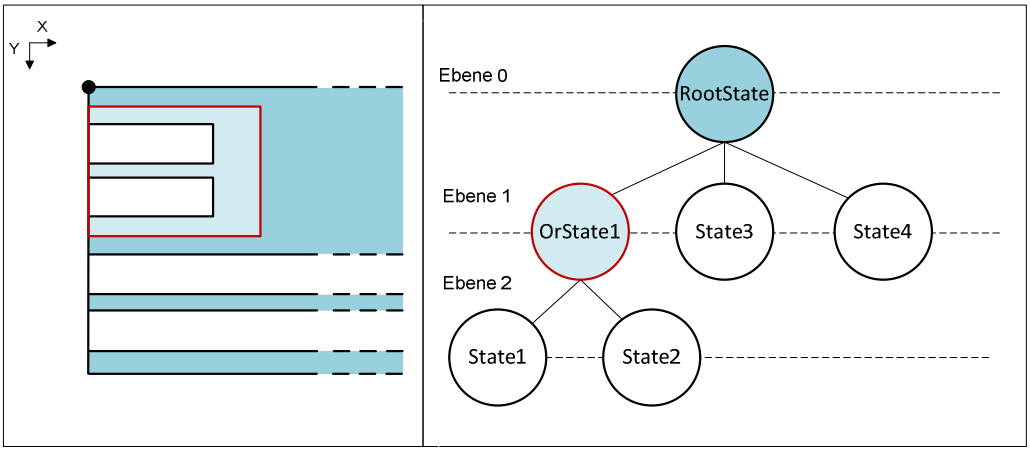

Abbildung 5.15: Breite des Or-States ist bekannt

Die States *State3* und *State4* erhalten die Breite `minNodeWidth`, da sie beide Blätter des Baumes sind. Abbildung 5.16 zeigt jedoch, dass diese zwei States noch keine korrekte Breite besitzen.

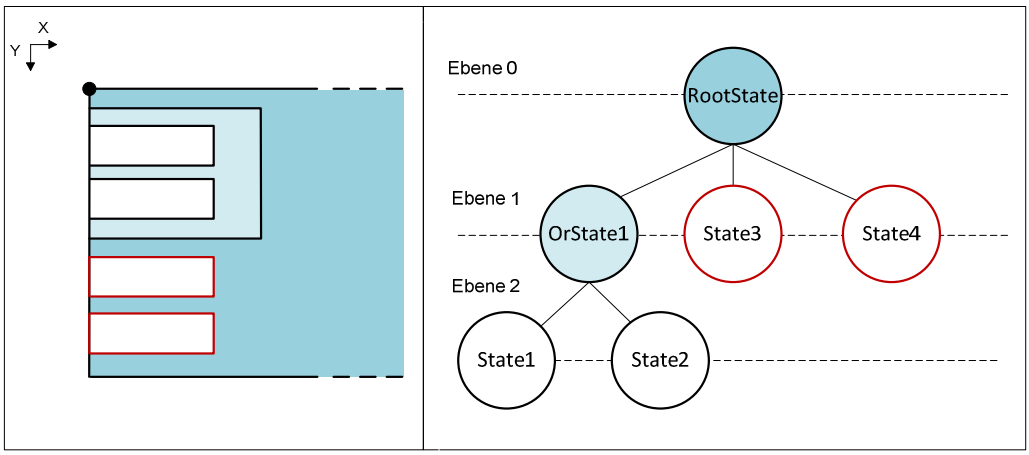

Abbildung 5.16: Festlegung der Breite von State3 und State4

State4 und *State5* müssen die Breite von *State1* erhalten. Für diese Anpassungen ist nun die Funktion `adjustNonLeaveNode()` zuständig. Der aktuell besuchte Knoten ist der Wurzelknoten *RootState*. *RootState* hat seine korrekte Breite bereits erhalten, diese befindet sich in `biggestNodeWidth`. Bevor die Breite in Zeile 18 gesetzt wird, wird `adjustNonLeaveNode()` in Zeile 16 für *OrState1*, *State3* und *State4* aufgerufen. Da *OrState1* der Knoten war, der für *RootState* die Höhe berechnet hat, hat der Aufruf von `adjustNonLeaveNode()` keine Auswirkung auf *RootState1* und seinen Kindern. Jedoch setzt der Aufruf für *State3* und *State4* die korrekten Breiten dieser beiden States. In Abbildung 3.17 wurden beide States grün umrahmt, um zu verdeutlichen, dass sie

im Nachhinein nochmals verändert wurden. Wie man erkennen kann, sind Or*State1*, *State3* und *State4* gleich breit.

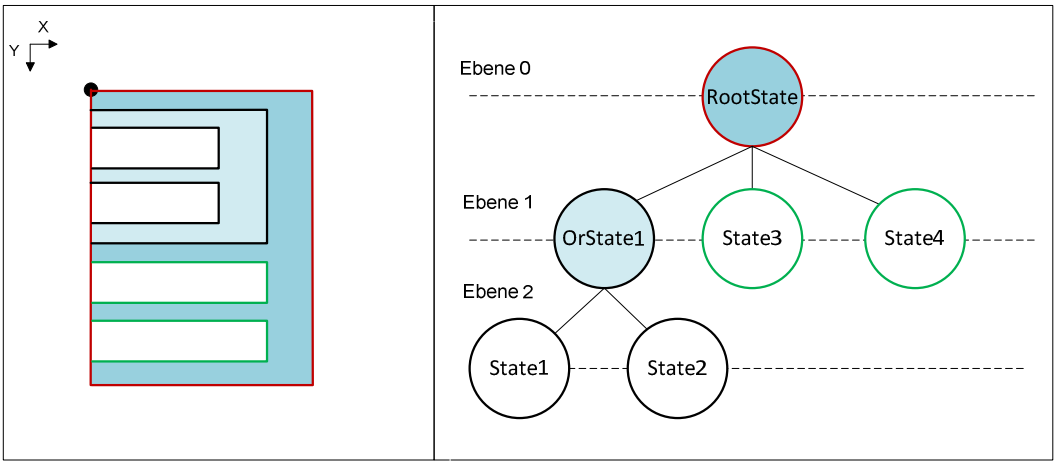

Abbildung 5.17: Nachträgliche Änderung der Breite

Der Graph ist nun fast vollständig aufgebaut. Was noch fehlt ist die korrekte X-Position der States, die im Folgenden erklärt wird.

Bestimmung der X-Position

Dieser Schritt ist sehr einfach und deswegen auch schnell erläutert. Die Aufgabe dieses Algorithmus ist es die Knoten in X-Richtung „einzurücken“, damit sie zentral im Graphen positioniert sind. Listing 5.4 stellt die Funktion `setXPosition()` vor.

```
Annahmen:
- Die Baumstruktur ist wohlgeformt
- Die Baumstruktur besteht aus mindestens einem Element
Input:
- Wurzelknoten der Baumstruktur
1:  function setXPosition(node)
2:    for i := 0 to node.getNumberOfChildren() do
3:      child := node.getChildFromPosition(i)
4:      posX := (node.getStateWith() - child.getStateWith()) / 2
5:      posX := posX + node.getXPosition()
6:      child.setXPosition(posX)
7:      if (node.getNumberOfChildren() == 0) then
8:      setXPosition(child)
9:      end if
10:   end for
11: end function
```

Listing 5.4: Pseudocode zur Ausrichtung der States in X-Richtung

Wie man erkennt ist der Algorithmus sehr kurz. Gleich wie bei den zwei weiter oben vorgestellten Algorithmen traversiert auch dieser wieder in Depth-First Weise über den Baum. Der Unterschied ist jedoch, dass dieser Algorithmus mit dem Layouten der Knoten bei der Wurzel beginnt und nicht bei den Blättern.

Die X-Position des Wurzelknotens muss nicht geändert werden, da dieser immer an der Position 0 steht. Deswegen ist *OrState1* der erste Knoten, der eine X-Position erhält (Abbildung 5.18). Um die X-Position von *OrState1* zu berechnen, wird die Differenz der Breite von *OrState1* und seinem Parent berechnet. Diese Differenz wird halbiert, da rechts und links ein Offset zu *RootState* bestehen soll. (Zeile 4 in Listing 5.4). Anschließend wird in Zeile 5 dieses Offset zum *OrState1* addiert. In Zeile 6 wird schließlich die X-Position des States gesetzt und somit ist dieser Knoten fertig gelayoutet.

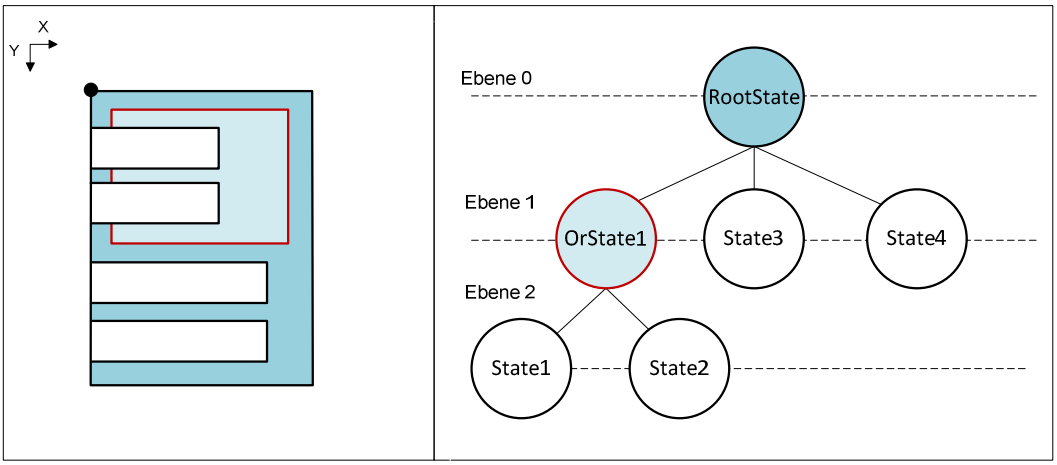

Abbildung 5.18: Endgültige Positionierung von OrState1

In Zeile 8 wird rekursiv für die Knoten *State1* und *State2* weitergemacht. Das Verfahren ist dabei genau dasselbe wie eben beschrieben. Der einzige Unterschied ist nur, dass nun *OrState1* der Referenzknoten ist und von diesem aus die X-Position für *State1* und *State2* festgelegt wird.

Nachdem alle Knoten in Ebene 2 fertig gelayoutet wurden, kehrt die Funktion auf Ebene 1 zurück. Hier müssen noch die Knoten *State3* und *State4* gelayoutet werden. Das Verfahren ist dabei wieder das Gleiche wie bei *OrState1*.

Das Ergebnis des Layouting-Vorganges zeigt Abbildung 5.19.

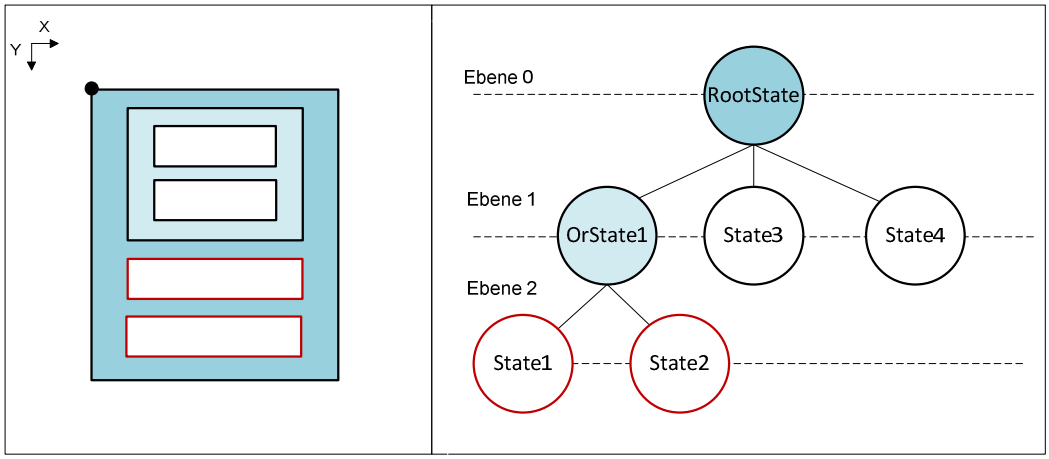

Abbildung 5.19: Fertiges Layout des State-Graphen

5.2.3 Algorithmen für Connections

Wie die einzelnen States positioniert werden, wurde im vorigen Kapitel ausführlich erklärt. Viele State-Graphen müssen neben den States auch noch Transitionen (Connections) anzeigen. Deswegen soll dieses Kapitel zeigen, wie StateGraphViewer die Visualisierung von Transitionen implementiert.

Anforderungen an die Transitionspfeile

Abbildung 5.20 zeigt einen State-Graphen mit mehreren Transitionspfeilen. Der orange vertikale Pfeil soll andeuten, dass die pfeillosen Enden der Transitionen einen Abstand voneinander haben müssen, damit sich die Transitionspfeile nicht überdecken oder überschneiden. Desweiteren sollen die orangen horizontalen Pfeile zeigen, dass auch die vertikalen Linien der Transitionspfeile einen Abstand von den States und einen Abstand zu anderen Transitionspfeilen einhalten müssen, um ein korrektes Layout zu besitzen. Die grünen und roten Pfeile werden weiter unten erklärt. Transitionen werden im Folgenden mit der Notation *Source-State->Target-State* angegeben.

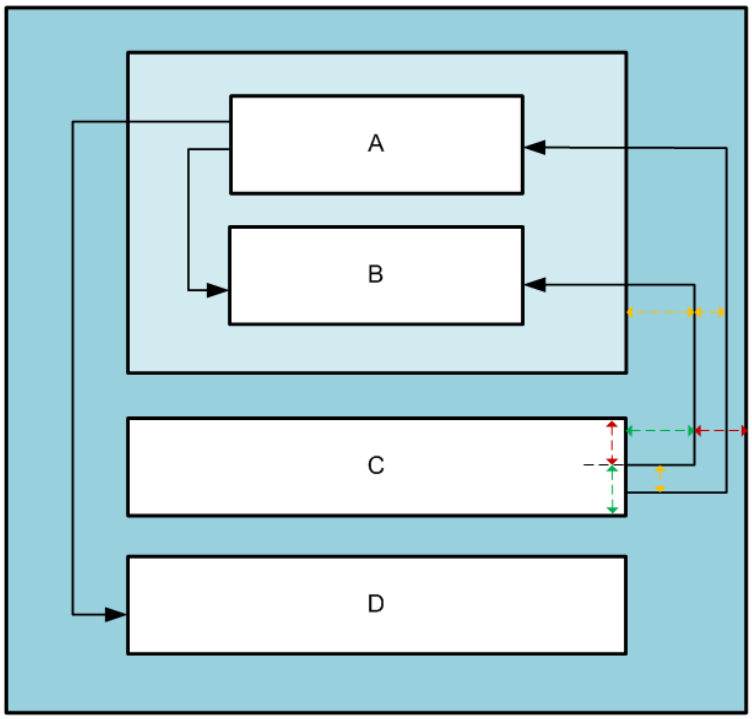

Abbildung 5.20: Ausrichtung der Transitionspfeile

Implementierung

Als erstes wird für jeden Source-State eine leere Liste erstellt. Alle Transitionen, die einen gemeinsamen Source-State haben werden in die Liste aufgenommen und der vertikalen Länge nach geordnet. Im Falle der Abbildung 5.20 gäbe es somit zwei Listen *L1* und *L2*, wobei *L1* die Liste von State *C* und *L2* die Liste von State *A* ist. Die Transition *C->B* ist kürzer als die Transition *C->A*. Somit wäre *L1={C->B, C->A}*. Auf der rechten Seite ist *A->B* kürzer als *A->D* und somit ist *L2={A->B, A->D}*.

Nachdem nun die Listen erstellt wurden, wird eine der zwei Listen ausgewählt und die erste Transition in der Liste gelayoutet. Welche der beiden Listen als erstes genommen wird ist egal, da *L1* die Transitionen rechts und *L2* die Transitionen links speichert und sich somit beide Listen nicht gegenseitig stören.

Als Beispiel wird die Liste *L1* genommen und die erste Transition *C->B* gelayoutet. Dazu wird zuerst der vertikale Abstand, welcher eine Konstante ist, der Transition zugewiesen (Der vertikale Abstand wird in Abbildung 5.20 als grüner horizontaler Pfeil dargestellt). Der nächste Schritt ist das pfeillose Ende der Transition *C->B* am rechten Rand des States *C* zu positionieren. Der horizontale Abstand der Transition (roter vertikaler Pfeil) spielt hier eine entscheidende Rolle, da es bei falscher Positionie-

rung zu Überschneidungen mit anderen Transitionen kommen kann. Um die Position zu bestimmen wird folgender, Trick benützt: Es wird das Verhältnis zwischen dem vertikalen Abstand der Transition und dem horizontalen Abstand zum Parent-State gebildet. Dieses Verhältnis wird mit der Höhe des Source-States multipliziert. Somit erhält man die exakten Positionen der Pfeilenden.

Beim pfeillosen Ende wird das Verhältnis zwischen dem Abstand zum Source-State *C* (grüner horizontaler Pfeil) und dem äußeren Ende des Root-States (roter horizontaler Pfeil) gebildet.

6 Implementierungs-beschreibung

Dieses Kapitel gibt einen Überblick über den internen Aufbau von StateGraphViewer, indem die wichtigsten Komponenten genauer beschrieben werden.

6.1 Paketdiagramm

Abbildung 6.1 zeigt die wichtigsten Pakete und die darin befindlichen Klassen, die StateGraphViewer aufbauen.

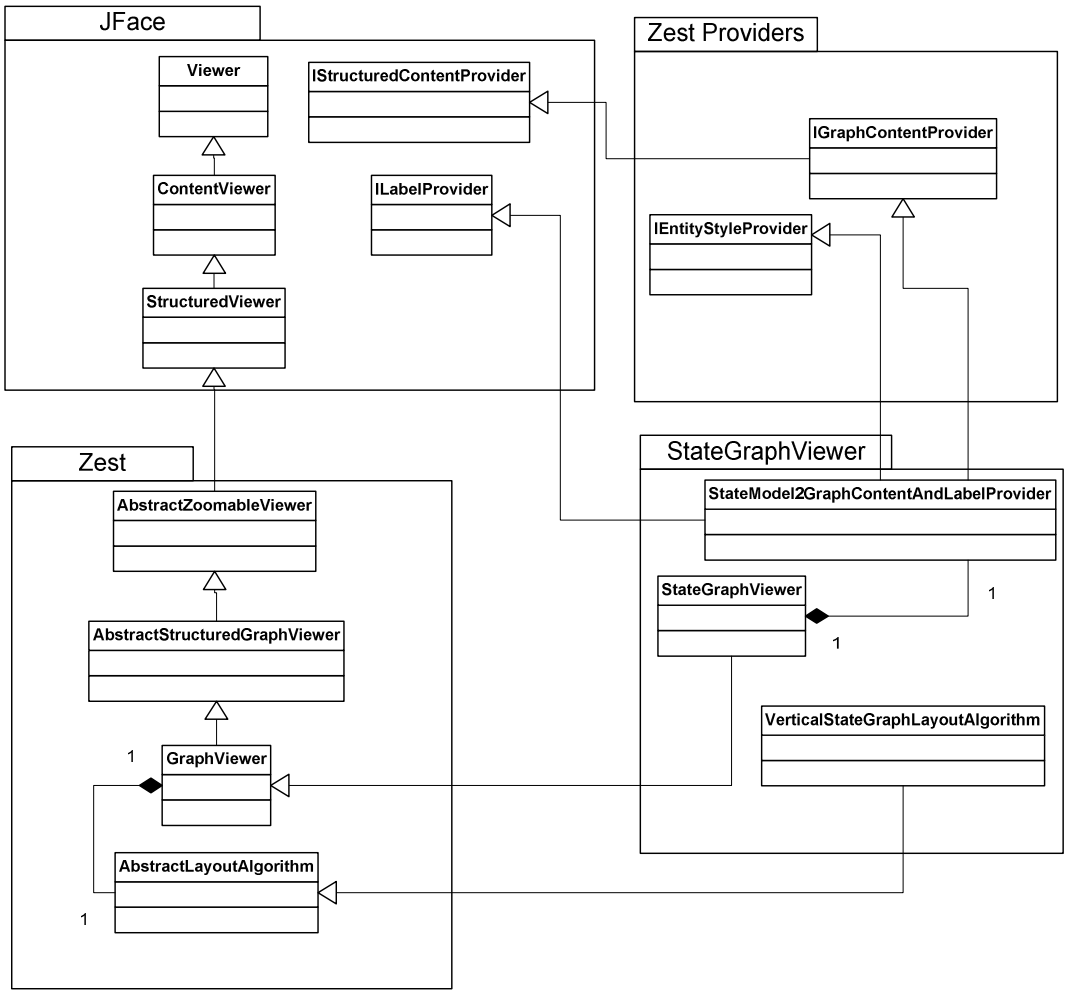

Abbildung 6.1: Paketdiagramm von Zest und StateGraphViewer

Zest verwendet die `StructuredViewer`-Klasse von JFace, um seine eigenen Viewer zu implementieren. `AbstractZoomableViewer` leitet von dieser Klasse ab, um Zooming zu ermöglichen. Jedoch sollte nicht direkt von dieser Klasse abgeleitet werden, um seine eigenen Zest-Views zu erstellen. Dafür ist vielmehr die Klasse `GraphViewer` gedacht, die volle Unterstützung zur Erstellung von eigenen Graphen-Views anbietet. Deswegen leitet StateGraphViewer `GraphViewer` ab und implementiert des Weiteren die entsprechenden Provider-Klassen von Zest und JFace mittels der Klasse `StateModel2GraphContentAndLabelProvider`, welche in Kapitel 6.2 genauer dargestellt wird. `VerticalStateGraphLayoutAlgorithm` leitet von der abstrakten Zest-Klasse `AbstractLayoutAlgorithm` ab und implementiert die in Kapitel 5.2.2 sowie Kapitel 5.2.3 vorgestellten Algorithmen zur Darstellung von States und Connections.

6.2 Provider-Implementierung

Zur Definition des Datenmodells implementiert StateGraphViewer eine Adapterklasse mit dem sprechenden Namen `StateModel2GraphContentAndLabelProvider`, welche das Datenmodell der State-Graphen „umformt“, um vom Layout-Algorithmus dargestellt werden zu können. `StateModel2GraphContentAndLabelProvider` implementiert dabei 4 Provider Interfaces:

- `IGraphContentProvider`: Erzeugt eine Beziehung zwischen einem Source- und einem Destination-State.
- `ILabelProvider`: Stellt den Text für die States zur Verfügung.
- `IEntityStyleProvider`: Gibt style-spezifische Parameter wie die Hintergrundfarbe oder den Tooltip-Text an.
- `IConnectionStyleProvider`: Gibt ähnlich wie `ILabelprovider` style-spezifische Parameter für Connections an.

```
public class StateModel2GraphContentAndLabelProviderAdapter implements
IGraphContentProvider,
ILabelProvider,
IEntityStyleProvider,
IConnectionStyleProvider {

      private Relation[] relArray;
      private List<Relation> relList;
      private IEntityStyleProvider styleProvider;
...
```

Listing 6.1: Providerklasse für das Datenmodell

Zur Erzeugung der Relationen zwischen States wurde eine Klasse `Relation` implementiert, der eine Beziehung zwischen einem Parent- und einem Child-State kapselt. Der Layouting-Algorithmus benötigt diese Relationen, um das korrekte Layouting der States im späteren Verlauf vorzunehmen.

```
public class Relation {
            private Object source, dest;
            Relation(Object source, Object dest) {
                  this.source = source;
                  this.dest = dest;

            }
```

Listing 6.2: Hilfsklasse zur Darstellung von Relationen

Das Feld `source` gibt den Parent-State an, das Feld `dest` den Child-State. Die Felder wurden bewusst nicht `parent` und `child` benannt, da eine Relation eine allgemeine Beziehung zwischen zwei beliebigen Objekten darstellt und nicht auf Beziehungen zwischen States beschränkt sein soll.

Zur Erzeugung der Relationen wird die Methode `getElements()` vom Interface `IGraphContentProvider` verwendet. Als Parameter bekommt die Methode den Root-Node eines State-Graphen übergeben.

```
@Override
public Object[] getElements(Object input) {
	if(relArray == null) {
		createRelationsRec((StateModelNode)input);
		relArray = relList.toArray(new Relation[relList.size()]);
	}
	return relArray;
}
```

Listing 6.3: Methode, um Zest die erstellten Relationen zu liefern

Der State-Graph wird rekursiv durchlaufen. Der Elternknoten ist immer der Parentstate und der Kindknoten immer der Child-State. Eine Relation wird in die Relationsliste `relist` aufgenommen. Da `getElements()` ein Array zurückgibt, wird die Liste `relList` in ein Array (`relArray`) umgewandelt. Die Methode `getElements()` liefert Zest anschließend die Liste mit den Relationen.

```
private void createRelationsRec(StateModelNode parent) {
// Firstly, create all relations between the parent and his children
	for(int i = 0; i < parent.getChildren().length; i++) {
		relList.add(new Relation(parent, parent.getChildren()[i]));
// secondly, go recursively downwards and create relation between
// a child and his child
		createRelationsRec(parent.getChildren()[i]);
	}
}
```

Listing 6.4: Erzeugt die Relationen

Alle anderen Methoden in StateModel2GraphContentAndLabelProvider sind einfach und werden deswegen nicht weiter erläutert.

7 Zusammenfassung

In dieser Arbeit wurde ein System zur graphischen Repräsentation von Zustandsautomaten entwickelt, welches es erlaubt, textbasierte Zustandsmodelle in Eclipse darzustellen. StateGraphViewer kann dabei mit einer beliebigen Menge von Zuständen umgehen und diese sauber in Eclipse darstellen. Es wurde ein spezieller Layout-Algorithmus implementiert, welcher die Zustände und Transitionen auf die richtige Größe skaliert und anschließend im Eclipse-Ansichtsfenster an der richtigen Position darstellt.

Es stellte sich während der Entwicklung heraus, dass Zest ein sehr fortgeschrittenes und robustes Framework ist, welches schon sehr viel an Funktionalität in Bezug auf das Darstellen von Graphen bereitstellt. Durch die klare Strukturierung des Quellcodes von Zest war die Einarbeitungszeit entsprechend kurz und es konnten in kurzer Zeit gute Ergebnisse erzielt werden.

Der schwierigste Teil der Entwicklung von StateGraphViewer war die Implementierung des Layout-Algorithmus. Dieser Teil des Projektes hat den größten Komplexitätsgrad und somit auch die meiste Entwicklungszeit in Anspruch genommen.

Rückwirkend gesehen war es ein Fehler, dass ich mein Projekt während der Entwicklung zu wenig getestet habe. Dadurch hat sich die Entwicklungszeit unnötig erhöht. Es wäre von Vorteil gewesen, wenn ich von Anfang an Unit-Tests geschrieben hätte, um die meisten Fehlerfälle im Vorhinein ausschließen zu können.

Literaturverzeichnis

[Zitzke 2011] Zitzke Stefan, statecharts.net
http://statecharts.net/wassind.htm#guard
Zugriff am 3.6.2011

[Hitz & Kappel 1999] Hitz Martin, Kappel Gerti: UML@Work – Von der Analyse zur Realisierung. dpunkt.verlag, Heidelberg et al., 1999

[Vogel 2011] Vogel Michael, Eclipse RCP Tutorial
http://www.vogella.de/articles/EclipseRCP/article.html
Zugriff am 5.7.2011

[Arndt 2007] Arndt Daniel, Eclipse RCP – Entwicklung von Java-Anwendungen für den Desktop
http://www.contentmanager.de/magazin/artikel_1492_java_eclipse_rcp.html
Zugriff am 5.7.2011

[SWT 2011] Eclipse Foundation
http://www.eclipse.org/swt/docs.php
Zugriff am 6.7.2011

[ECL 2011] Eclipse Foundation
http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/jface.htm
Zugriff am 6.7.2011

[Ebert 2011] Ebert Ralf, JFace Überblick, JFace Structured Viewers
http://www.ralfebert.de/eclipse_rcp/jface
Zugriff am 9.7.2011

[Kocher 2005] Kocher Hartmut, JFS 2005: Java SWT und JFace in der Praxis
http://www.java-forum-stuttgart.de/jfs/2005/folien/E2_Cortex-Brainware.pdf
Zugriff am 6.7.2011

[GEF1 2011] Eclipse Wiki
http://wiki.eclipse.org/GEF_Description
Zugriff am 3.7.2011

[DRAW2D 2011] Draw2d Programmer's Guide
http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.draw2d.doc.isv/guide/guide.html
Zugriff am 5.7.2011

[ZEST 2011]: GEF Zest Visualization
http://wiki.eclipse.org/GEF_Zest_Visualization
Zugriff am 3.6.2011

[GEF2 2011] Eclipse Foundation
http://www.eclipse.org/gef
Zugriff 17.2.2011

Abbildungsverzeichnis

Codeverzeichnis

Danksagung

Ich möchte mich zuerst bei meinem Betreuer Dr. Herbert Prähofer bedanken, der mich während der Entstehung dieser Arbeit tatkräftig unterstützt hat. Des Weiteren möchte ich mich bei Herrn Florian Kölbleitner bedanken, welcher mir mit guten Ratschlägen sehr geholfen hat.

Des Weiteren möchte ich mich bei meiner Freundin Nina für ihr Verständnis bedanken, da ich während der Ausarbeitung dieser Arbeit weniger Zeit für sie hatte.

Außerdem möchte ich mich noch bei meinem Bruder Herrn Florian Ennemoser, Herrn Johannes Schwarzenberger und Herrn Martin Fartek bedanken, die mir durch ihre moralische Unterstützung sehr geholfen haben.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, Monat Jahr Name