Hes·so VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Bereich Ingenieurwissenschaften
Rte du Rawyl 47
CH-1950 Sitten 2
Tel. +41 27 606 85 11
Fax +41 27 606 85 75
info@hevs.ch

www.hevs.ch

Studiengang

Infotronics

Diplom 2011

Ralph Martig

Model based Software Development Tool Chain for Embedded Systems

Dozent	Medard Rieder
Experte	Rico Steiner

Sion, 11. Juli 2011

Da es sich um den nicht korrigierten Originalbericht des Studenten handelt, kann dieser Fehler oder ungenaue Angaben umfassen.

Model based Software Development Tool Chain for Embedded Systems

Diplomarbeit

University of Applied Sciences – Western Switzerland

Systems Engineering

vorgelegt von:	**Ralph Martig**
Studiengang:	***Systemtechnik***
Vertiefungsrichtung:	***Infotronics***
Nr. der Diplomarbeit:	***it/2011/51***
Dozent:	**Medard Rieder**
Assistenten:	**Michael Clausen, Thomas Sterren**
Experte:	**Rico Steiner**

HES-SO Valais

SI	TV
X	X

Données du travail de diplôme
Daten der Diplomarbeit

FO 1.2.02.07.AB
pof/31/01/2009

☒ FSI ☐ FTV	Année académique / *Studienjahr* **2010/2011**	No TD / *Nr. DA* **it/2011/51**
Mandant / *Auftraggeber* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire *Partnerinstitution*	Etudiant / *Student* **Ralph Martig** Professeur / *Dozent* **Medard Rieder**	Lieu d'exécution / *Ausführungsort* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire *Partnerinstitution*
Travail confidentiel / *vertrauliche Arbeit* ☐ oui / ja [1] ☒ non / nein	Expert / *Experte* *(données complètes)* **Steiner Rico** Studer Innotec, Sion	

Titre / *Titel*

Model based Software Development Tool Chain for Embedded Systems

Description et Objectifs / *Beschreibung und Ziele*

Das Ziel dieser Bachelorarbeit besteht darin, einen Modelleditor für UML zu schreiben. Neben der Möglichkeit, Standard UML Elemente zu zeichnen, muss der Editor auch in der Lage sein, Code zu editieren, sprich zum Beispiel den Körper einer Methode zu implementieren.

Während der Semesterarbeit wurden bereits ein Datenmodell sowie das Layout des Editors bereitgestellt. Es gilt nun, den Editor selber zu implementieren. Höchste Priorität haben hierbei die sogenannten Zustandsdiagramme. Aber es besteht nach wie vor das Ziel, auch weitere UML Diagramme wie Klassendiagramme (Pakete, Klassen und Relationen) sowie Objektdiagramme mit Objekten und Links zeichnen zu können.

Die Mechanismen zum speichern bzw. laden eines Models wurden in der Semesterarbeit bereits erschaffen. Es muss jedoch dafür gesorgt werden, dass die Daten eines Projektes unter zwei verschiedenen Dateitypen gespeichert werden. Einer der Dateitypen enthält im Prinzip die logische Information des Projektes oder von Elementen des Projektes und der andere die graphische Darstellung der jeweiligen Elemente.

Als minimale Resultate werden die folgenden erwartet:

- Vollständiger Editor für Zustandsmaschinen
- Abspeicherung in zwei Dateiformaten
- Abschlussbericht
- Präsentation für Verfechtung
- Bedienungsanleitung und Tutorial mit komplexem Beispiel
- Vollständiger Quellcode auf CD.

Délais / *Termine*

Attribution du thème / *Ausgabe des Auftrags:*
16.05.2011

Exposition publique / *Ausstellung Diplomarbeiten:*
02.09.2011

Remise du rapport / *Abgabe des Schlussberichts:*
11.07.2011 | 12.00 Uhr

Défense orale / *Mündliche Verteidigung:*
dès la semaine 36 / *ab Woche 36*

Signature ou visa / *Unterschrift oder Visum*

Responsable de l'orientation
Leiter der Vertiefungsrichtung: .....

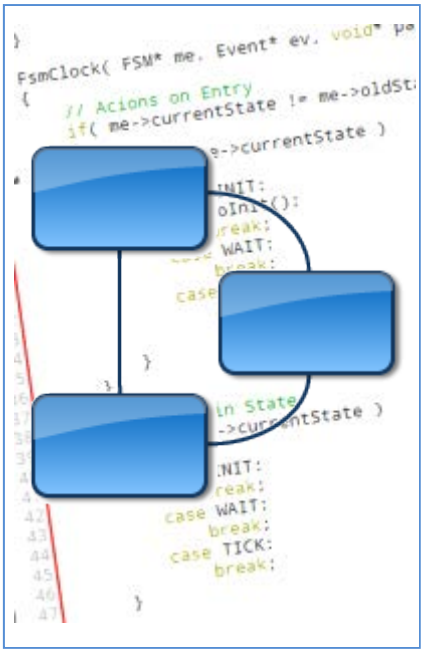
[1] Etudiant/*Student*: .....

[1] *Par sa signature, l'étudiant-e s'engage à respecter strictement la directive et le caractère confidentiel du travail de diplôme qui lui est confié et des informations mises à sa disposition.*
Durch seine Unterschrift verpflichtet sich der Student, die Richtlinie einzuhalten sowie die Vertraulichkeit der Diplomarbeit und der dafür zur Verfügung gestellten Informationen zu wahren.

Rapport reçu le / *Schlussbericht erhalten am* ..... Visa du secrétariat / *Visum des Sekretariats* .....

Hes·so VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Model based Software Development Tool Chain for Embedded Systems



Diplomarbeit
| 2011 |

Diplomand/in Ralph Martig

Ziel des Projekts

Das Ziel der Bachelorarbeit besteht darin, die Basis für ein modellbasiertes Entwicklungstool zu schaffen. In einem ersten Schritt wird zunächst nur das Erstellen von Zustandswechseldiagrammen behandelt.

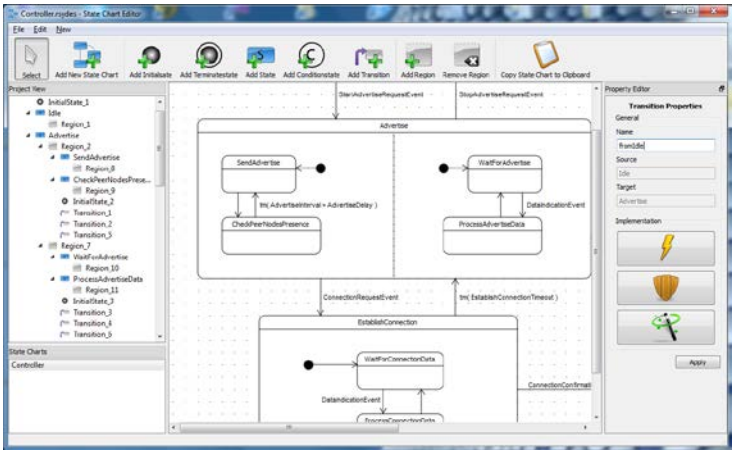
Methoden | Experimente | Resultate

Eine graphische Benutzeroberfläche erlaubt es dem Benutzer Modelle zu erstellen. Zurzeit ist es möglich, mit diesem Tool vollständige Zustandswechseldiagramme zu zeichnen.

Nebst dem Zeichnen der Zustandsmaschinen, kann zudem Programmcode in Zuständen oder auf Transitionen implementiert werden. Die Diagramme, sowie der Code können gedruckt oder zur weiteren Verwendung in die Zwischenablage kopiert werden.

Das Modell wird in eine Archiv-Datei gespeichert, welche die graphischen und logischen Informationen in der Form von XML enthält.
Die logischen Informationen können in einem nächsten Schritt, beispielsweise von einem Code-Generator, weiterverarbeitet werden oder auch in ein gängiges Format wie XMI exportiert werden.

Das Entwicklungstool wurde mit Hilfe der Qt-Library implementiert. Weitere Libraries (z.B. ZLIB) wurden ebenfalls verwendet. Vorgängig an die Implementierung wurden jeweils die UML-Modelle des zu entwickelnden Codes spezifiziert.



Die obenstehende Abbildung zeigt eine mittels des erwähnten Tools entwickelte Zustandsmaschine.

Die graphische Benutzeroberfläche bietet dem Benutzer eine Toolbar zum Zeichnen, Ansichten zur Übersicht des Modells, sowie ein Property-Editor zur Bearbeitung der Eigenschaften der verschiedenen Elemente.

Studiengang
Systemtechnik

Anwendungsbereich
Infotronics

Verantwortliche/r Dozent/in
Medard Rieder
medard.rieder@hevs.ch

Partner
HES-SO Wallis,
Institut für Systemtechnik

HES-SO Wallis
Route du Rawyl 47
1950 Sitten

Tel. 027 606 85 11
Web www.hevs.ch

Inhaltsverzeichnis

Vorwort

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich während dieser Arbeit in irgendwelcher Art und Weise unterstützt haben.

Persönlich danken möchte ich meinem Dozenten Medard Rieder (Dozent für Methodik, Informatik & Telekommunikation) für seine Unterstützung, hilfreichen Anregungen und konstruktive Kritik bei der Realisierung dieser Diplomarbeit.

Weiter möchte ich mich bei Michael Clausen (Wissenschaftlicher Mitarbeiter ISI) bedanken. Immer wieder konnte er mir bei Problemen in Qt Tipps geben oder bei Compiler-Problemen weiterhelfen. Seine kritischen Meinungen waren oft der Grund, nach einer geeigneteren Lösung zu suchen und somit das Resultat zu verbessern.

Ebenfalls danke ich Thomas Sterren (Wissenschaftlicher Mitarbeiter ISI), den ich bei Syntax-Fehlern und Modellierungsproblemen nach Rat fragen konnte. Dank seiner SVN-Dropbox Lösung war meine Arbeit immer bestens gesichert.

Auch meine Mitschüler Fernando Kummer und Thierry Hischier (Diplomanden Infotronics) möchte ich erwähnen. Sie haben sich bereiterklärt, mein Tool zu testen und konnten mir nützliche Verbesserungsvorschläge geben.

Mein ganz besonderer Dank gilt abschliessend meinen Eltern, die mir während des gesamten Studiums stets helfend zur Seiten standen.

Sion, 11. Juli 2011

Ralph Martig

1 Einleitung

Diese Diplomarbeit wird für das Institut für Systemtechnik (ISI) der Fachhochschule Westschweiz HES-SO Wallis entwickelt.

Das Ziel der Diplomarbeit besteht darin, die Basis für ein modellbasiertes Entwicklungstool zu schaffen. In einem ersten Schritt erlaubt dieses Tool das Erstellen von Zustandswechseldiagrammen. Das ISI wird in einem weiterführenden Projekt einen Code-Generator entwickeln, der die gezeichneten Zustandsmaschinen in Code umwandeln kann.

Das modellbasierte Tool wird in erster Linie für die Studenten der HES-SO Wallis (Studiengang Systemtechnik) entwickelt, damit die Studenten das Projekt während der Summer-School mit Hilfe dieses Tools realisieren können. Weitere Anwendungsbereiche sind möglich.

Dieses Dokument beinhaltet die vollständige technische Dokumentation zum realisierten Entwicklungstool und ist in vier Teile gegliedert. In einem ersten Kapitel wird die graphische Benutzeroberfläche des Tools beschrieben. Die nächsten Kapitel befassen sich mit der Entwicklung der Modelle zum Ablegen der Daten. Es folgt ein Abschnitt, der die Realisierung der graphischen Elemente, welche zum Zeichnen der Zustandswechsel-diagramme verwendet werden, aufzeigt. Im letzten Teil wird auf die Implementierung des Entwicklungstools eingegangen, das heisst, die Elemente der vorangehenden Kapitel werden zusammengefügt.

In den nächsten Abschnitten folgt eine detailliertere Beschreibung der Diplomarbeit und die Ziele der Diplomarbeit werden definiert.

1.1 Beschreibung der Diplomarbeit

Das modellbasierte Entwicklungstool, folgend State Chart Editor genannt, erlaubt es, mit Hilfe der Standard UML Elemente, Zustandswechseldiagramme zeichnen zu können.

Nebst dem Zeichnen der Zustandsmaschinen, erlaubt es der State Chart Editor zudem, in den Zuständen oder auf Transitionen Programmcode zu implementieren.

Die Zustandswechseldiagramme werden in zwei Dateien abgespeichert, wobei eine Datei die graphischen Informationen und die Andere die logischen Informationen der Zustandsmaschine enthält.

Der State Chart Editor bietet die Möglichkeit, die logischen Informationen der Zustandsmaschinen zu exportieren, damit diese in einem nächsten Schritt, beispielsweise von einen Code-Generator, weiterverarbeitet werden können.

1.2 Ziele der Diplomarbeit

Die Ziele der Diplomarbeit sind die Implementierung des vollständigen Tools zum Zeichnen von Zustandsmaschinen, das Erstellen einer Dokumentation und einer Bedienungsanleitung mit einem komplexem Beispiel.

2 Die graphische Benutzeroberfläche GUI

Die graphische Benutzeroberfläche erlaubt es dem Benutzer, den State Chart Editor zu bedienen.

Untenstehende Abbildung zeigt das Main-Window des Editors.

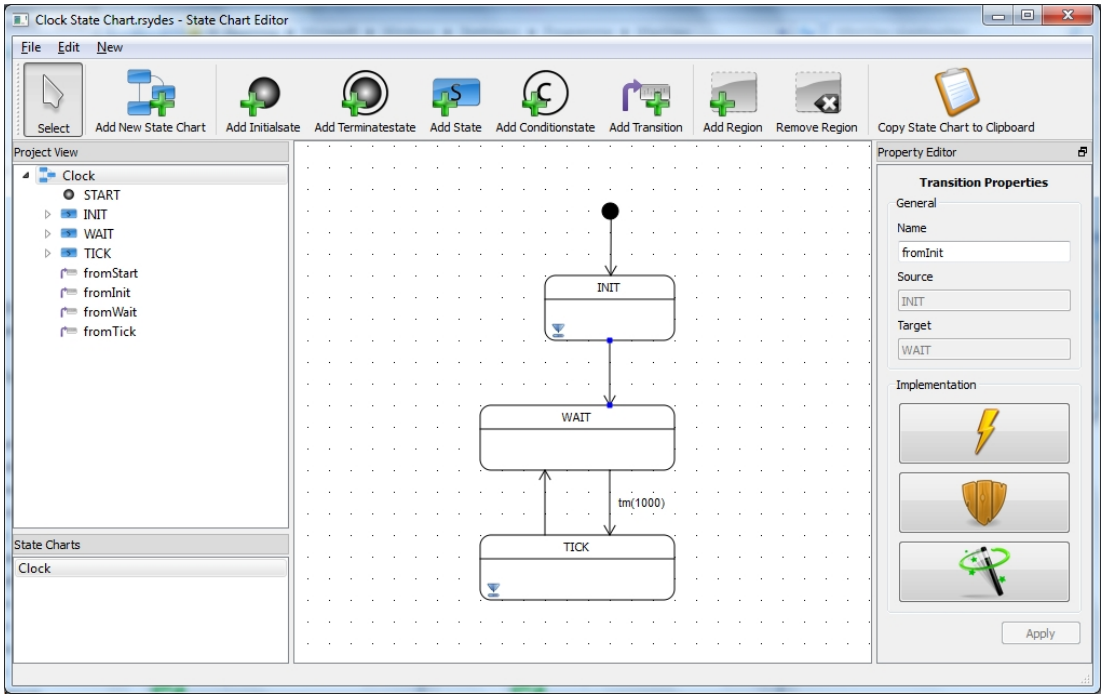


Abbildung 1: Das Main-Window

Das Main-Window ist in drei Bereiche aufgeteilt. Ganz links befindet sich eine Projekt- und State-Chart-Ansicht, die einen Überblick zu den Zustandsmaschinen verschaffen. In der Mitte befindet sich die Arbeitsfläche, auf welcher die Zustandsmaschinen gezeichnet werden. Ganz rechts wird ein Property-Editor angezeigt, welcher die Eigenschaften des ausgewählten Elements anzeigt.

Eine Menüleiste, sowie eine Toolbar bieten dem Benutzer verschiedene Funktionalitäten. Am unteren Fensterrand zeigt eine Statusleiste Informationen (beispielsweise während des Ladevorgangs) an.

Folgend werden die Bereiche und Elemente des Main-Windows kurz beschrieben.

2.1 Die Ansichten

Die Projekt- sowie die State-Chart-Ansicht verschaffen dem Benutzer einen Überblick zu den Zustandsmaschinen.

2.1.1 Die Projekt-Ansicht

Die Projekt-Ansicht listet alle Elemente, die zum Zeichnen der Zustandsmaschine verwendet wurden, hierarchisch auf.

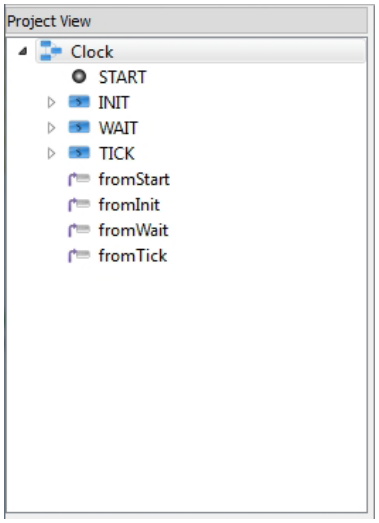


Abbildung 2: Die Projekt-Ansicht

2.1.2 Die State-Chart-Ansicht

Jede Zustandsmaschine wird in der State-Chart-Ansicht aufgelistet. Durch Auswählen einer Zustandsmaschine in dieser Ansicht, kann diese geöffnet und angezeigt werden.

Abbildung 3: Die State-Chart-Ansicht

2.2 Die Arbeitsfläche

Auf der Arbeitsfläche kann der Benutzer die Zustandsmaschinen zeichnen. Es wird jeweils die in der State-Chart-Ansicht ausgewählte Zustandsmaschine angezeigt.

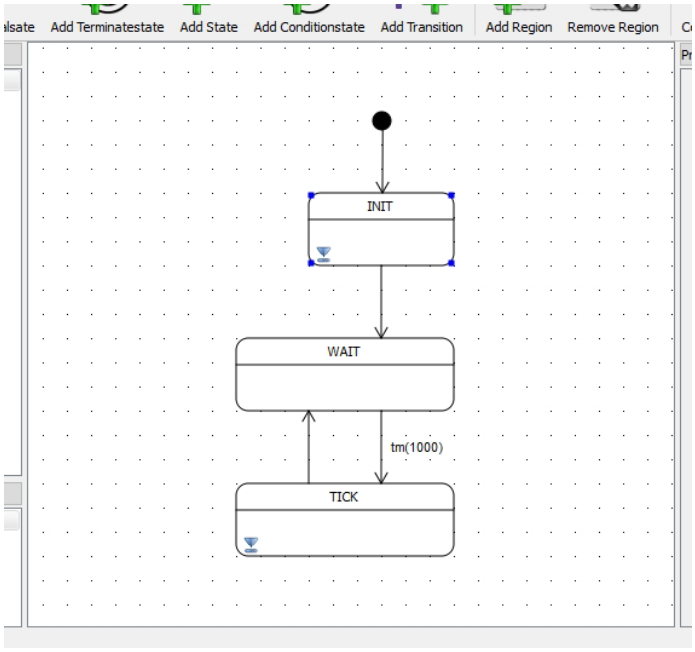


Abbildung 4: Die Arbeitsfläche

2.3 Der Property-Editor

Im Property-Editor werden jeweils die Eigenschaften des ausgewählten Elements angezeigt und können vom Benutzer angepasst werden.

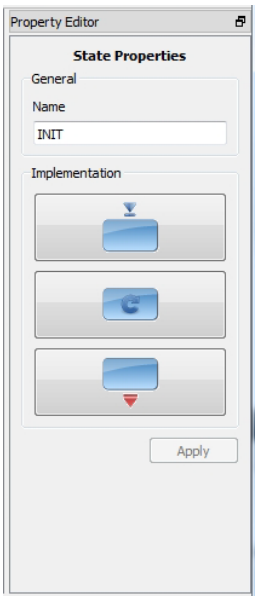


Abbildung 5: Der Property-Editor

2.3.1 Zustandsmaschine /Pseudo-Zustand / Region

Im Property-Editor zu den Zustandsmaschinen, Pseudo-Zuständen und den Regionen kann lediglich der Name des Elements geändert werden.

2.3.2 Zustand

Der Property-Editor der Zustände ermöglicht es, den Namen des Zustands zu ändern. Zudem kann das Verhalten des Zustands (Action On Entry, Action In State und Action On Exit) durch Anklicken des entsprechenden Buttons, im sich öffnenden Code-Editor, implementiert werden.

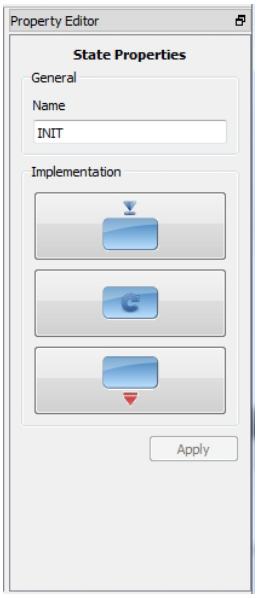


Abbildung 6: Property-Editor eines Zustandes

Folgende Abbildung zeigt den Code-Editor, der durch das Anklicken eines Buttons zur Implementierung des Verhaltens geöffnet wird.

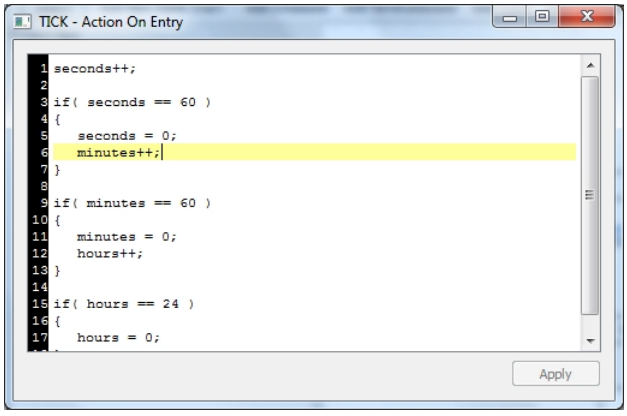


Abbildung 7: Der Code-Editor

2.3.3 Transition

Der Property-Editor der Tranistionen ist ähnlich jenem der Zusände. Es lässt sich der Name ändern und das Verhalten (Trigger, Guard und Action), im sich öffnenden Code-Editor (siehe Abbildung 7), implementieren.

Zusätzlich werden Source und Target der Transition angezeigt.

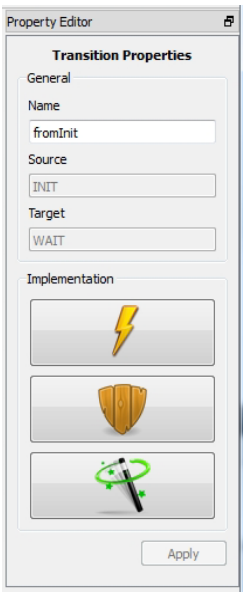


Abbildung 8: Property-Editor einer Transition

2.4 Die Menüleiste

Im Folgenden werden die drei Menüs der Menüleiste beschrieben.

2.4.1 Das Menü «File»

Im Menü File befinden sich Basis-Funktionen, wie zum Beispiel ein neues Projekt erstellen, das aktuelle Projekt speichern oder ein bestehendes Projekt öffnen.

Zusätzlich kann der logische Teil der Zustandsmaschine in ein XML-File exportiert werden, welches später mit dem Code-Generator weiterverarbeitet werden kann.

Ebenfalls eine Druck-Funktion ist implementiert, mit welcher die auf der Arbeitsfläche angezeigte Zustandsmaschine ausgedruckt werden kann.

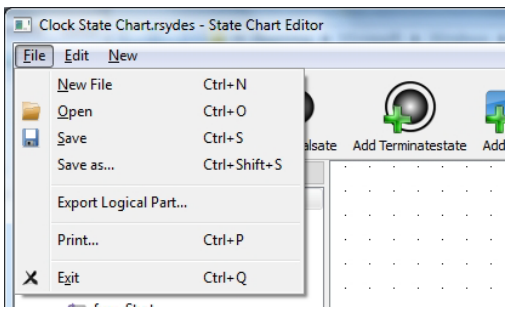


Abbildung 9: Das Menü «File»

2.4.2 Das Menü «Edit»

Im Menü Edit kann in den Selektierungsmodus gewechselt und den Zuständen Regionen hinzugefügt/entfernt werden.

Zudem wird dem Benutzer eine Funktion angeboten, die angezeigte Zustandsmaschine als Bild in die Zwischenablage zu kopieren, damit diese beispielsweise zum Erstellen einer Dokumentation verwendet werden kann.

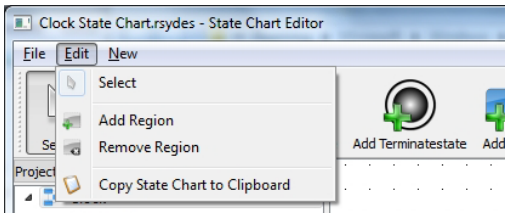


Abbildung 10: Das Menü «Edit»

2.4.3 Das Menü «New»

Sämtliche Elemente die der Zustandsmaschine hinzugefügt werden können, können in diesem Menü ausgewählt werden.

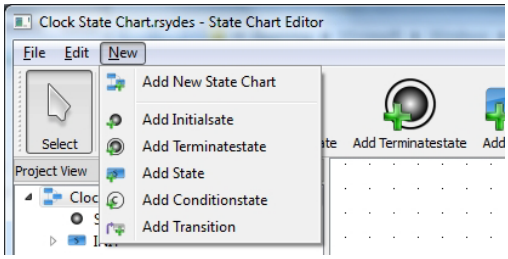


Abbildung 11: Das Menü «New»

2.5 Die Toolbar

Damit der Benutzer sich nicht ständig durch die Menüs quälen muss, bietet die Toolbar einen schnellen Zugriff auf die wichtigsten Funktionen.

Die Toolbar beinhaltet die Aktionen, die in den Menüs Edit und New zu finden sind.

Abbildung 12: Die Toolbar

2.6 Die Statusleiste

Die Statusleiste wird dazu verwendet, um Informationen zu Vorgängen anzuzeigen. So zeigt die Statusleiste beispielsweise einen erfolgreichen bzw. fehlerhaften Speicher-/ Ladevorgang oder Exportvorgang an.

3 Die Entwicklung des Datenmodells

Der State Chart Editor erlaubt es, Zustandsmaschinen mittels unterschiedlichen Elementen zu zeichnen. Die Anzahl der Elemente, aus welchen eine Zustandsmaschine besteht, ist im Voraus nicht bekannt und kann abhängig von der Zustandsmaschine, die gezeichnet wird, ändern.

Jedes Element kann zudem mehrere supplementäre Daten enthalten (Name, Verhalten, usw.) oder sogar weitere Elemente enthalten (Regionen können mehrere Elemente enthalten). Daher ist es wichtig, die Gesamtheit der Elemente sorgfältig zu verwalten.

Um sämtliche Elemente des State Chart Editors zu verwalten, wird ein Datenmodell erstellt. Dazu muss zuerst ein struktureller Aufbau erstellt werden, welcher sämtliche Elemente definiert, welcher aufzeiget, welche Elemente welche Daten bzw. Elemente enthalten können. Auf dieser Basis kann schliesslich eine Schnittstelle (Modell) erstellt werden, welche das Hinzufügen, Entfernen oder Bearbeiten der Elemente in der Struktur ermöglicht.

Bemerkung: Der Begriff Modell hat in diesem Abschnitt zwei Bedeutungen. Einerseits wird mit *Datenmodell* der strukturelle Aufbau der Daten gemeint, andererseits wird die Schnittstelle zu den Daten als *Modell* bezeichnet.

Die untenstehende Abbildung versucht den Nutzen des strukturellen Aufbaus für das Datenmodell aufzuzeigen.

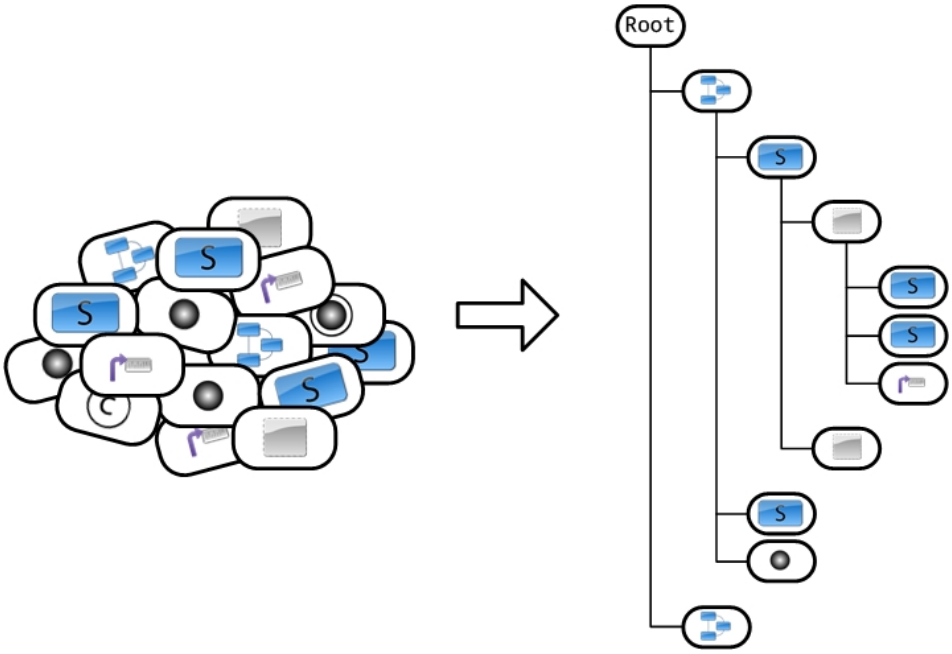


Abbildung 13: Nutzen eines strukturellen Aufbaus für das Datenmodell

In den folgenden Abschnitten wird die Entwicklung und Implementierung des Datenmodells erläutert. Es wird mit dem strukturellen Aufbau der Elemente begonnen, anschliessend werden die Elemente und die Schnittstelle (Modell) beschrieben.

Zudem soll aufgezeigt werden, wie die Daten abgespeichert und wieder geladen werden können.

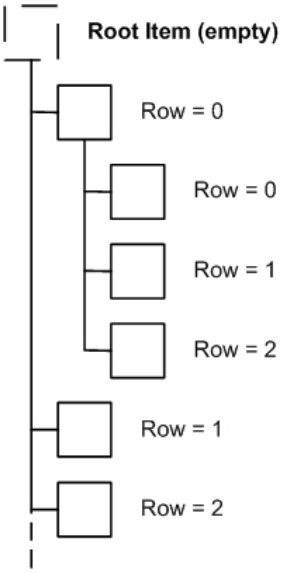
3.1 Der strukturelle Aufbau

Die Elemente (siehe § 3.2 Das Element) werden hierarchisch in einer Baumstruktur abgelegt. Sie werden mittels Zeigern miteinander verlinkt.

Standardmässig hat jedes Element ein Eltern-Element. Einige Elemente können zudem Kinder enthalten. Das Root-Element hingegen besitzt kein Eltern-Element und kann auch nicht ausserhalb des Datenmodells referenziert werden. Es handelt sich um ein „leeres“ Element.

Jedes Element enthält Informationen zu seiner Position in der Struktur. Es kann sein Eltern-Element, sowie seine Position im Eltern-Element (row) mitteilen. Diese Informationen vereinfachen anschliessend die Implementierung der Schnittstelle (Modell).

Die zeigerbasierte Baumstruktur hat den Vorteil, dass die „Adresse“ eines jeden Elements mittels eines Index (*QModelIndex*) erhalten werden kann.



Quelle: Qt Reference Documentation [3]

Abbildung 14: Repräsentierung der Elemente als Baumstruktur

Nun müssen alle Elemente definiert werden, die später in einer Zustandsmaschine enthalten sein können. Folgendes Klassendiagram soll die Elemente aufzeigen.

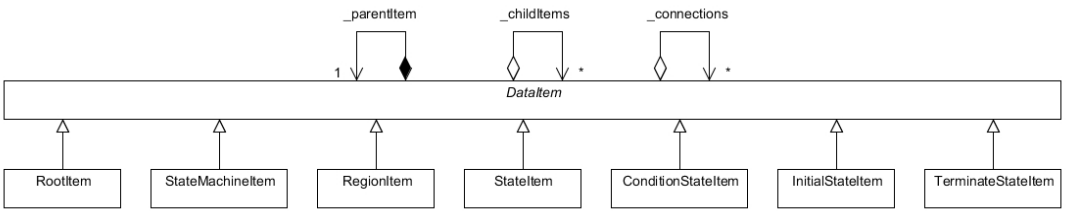


Diagramm 1: Klassendiagramm der Datenstruktur

Schliesslich können die definierten Elemente in einer Baumstruktur dargestellt werden.

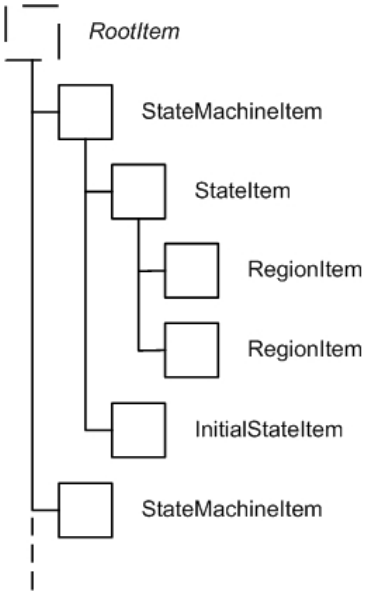


Abbildung 15: Baumstruktur der Elemente

3.1.1 Der Index (*QModelIndex*)

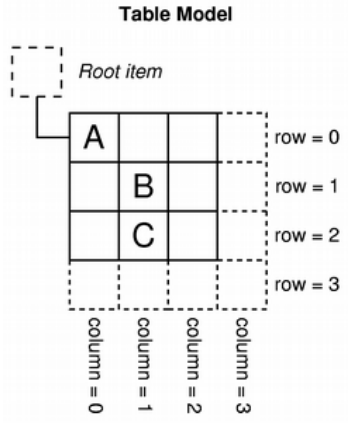
Dieser Abschnitt soll kurz die Indexierung der Elemente mit Hilfe des *QModelIndex* beschreiben.

Ein Index erlaubt es, jedes Element eines Modells zu referenzieren, um auf seine Informationen zugreifen oder diese bearbeiten zu können. Bei einer solchen Referenz handelt es sich um eine temporäre Referenz, da sich die Struktur während der Laufzeit verändern kann und somit der Index nicht mehr aktuell sein könnte.

Um den Index eines Elementes zu erhalten, sind drei Informationen nötig. Es werden ein Zeiger auf das Element und die Position des Elements (Zeile und Spalte) benötigt.

Anschliessend lässt sich mittels der `QAbstractItemModel::createIndex()` Methode des Modells ein Index erstellen. Handelt es sich um das Root-Element, kann mit `QModelIndex::QModelIndex()` ein neuer leerer Index erstellt werden.

Untenstehende Abbildung zeigt ein Beispiel eines Tabellen-Modells.



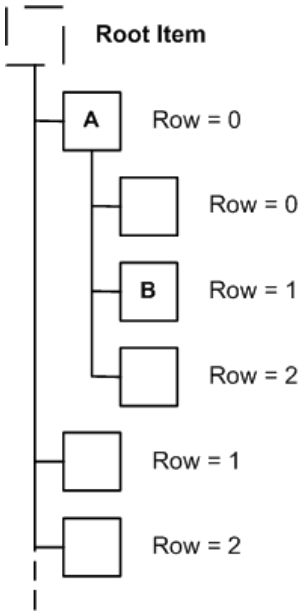
Quelle: Qt Reference Documentation [3]

Abbildung 16: Tabellen-Modell zur Beschreibung der Item-Indexierung

Die Indizes der Elemente A, B und C können nun wie folgt erhalten werden.

```
QModelIndex  indexA = QAbstractItemModel::createIndex(0, 0, <pointer to A>);
QModelIndex  indexB = QAbstractItemModel::createIndex(1, 1, <pointer to B>);
QModelIndex  indexC = QAbstractItemModel::createIndex(2, 1, <pointer to C>);
```

Da es sich aber bei der verwendeten Struktur nicht um eine Tabelle, sondern eine Baumstruktur handelt, vereinfacht sich die Indexierung dadurch, weil das Modell jeweils nur eine Spalte besitzt.



Quelle: Qt Reference Documentation [3]

Abbildung 17: Model Index in einer Baumstruktur

Um nun den Index von A bzw. B zu erhalten, wird wie folgt vorgegangen:

```
QModelIndex  indexA = QAbstractItemModel::createIndex(0, 0, <pointer to A>);
QModelIndex  indexB = QAbstractItemModel::createIndex(1, 0, <pointer to B>);
```

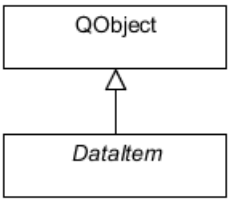
Später kann das Modell somit jedes Element referenzieren. Dafür wird die Funktion `QAbstractItemModel::index()` neu implementiert. Um den Index eines Elements mit dieser Funktion zu erhalten sind drei Informationen nötig. Es werden ein Zeiger auf das Eltern-Element und die Position des Elements (Zeile und Spalte) benötigt.

Wird nochmals die Abbildung 17 betrachtet, würden die Anweisungen zum Referenzieren der beiden Elemente durch einen Index wie folgt aussehen:

```
QModelIndex  indexA = model->index(0, 0, QModelIndex());
QModelIndex  indexB = model->index(1, 0, indexA);
```

Weitere Informationen und detailliertere Beschreibungen können der Qt Reference Documentation [3] entnommen werden.

3.2 Das Element



3.2.1 Beschreibung

Elemente sind Objekte, mit welchen Zustandsmaschinen erstellt werden und eine bestimmte Anzahl Informationen enthalten können.

Alle Elemente besitzen eine gewisse Anzahl an Methoden und Attributen, haben ein Eltern-Element und können teilweise weitere Elemente enthalten. Diese gemeinsamen Punkte werden in einer Basisklasse zusammengefasst. Später kann jedes Element diese Klasse implementieren und seine Besonderheiten hinzufügen.

Diese Basisklasse ist wie folgt definiert.

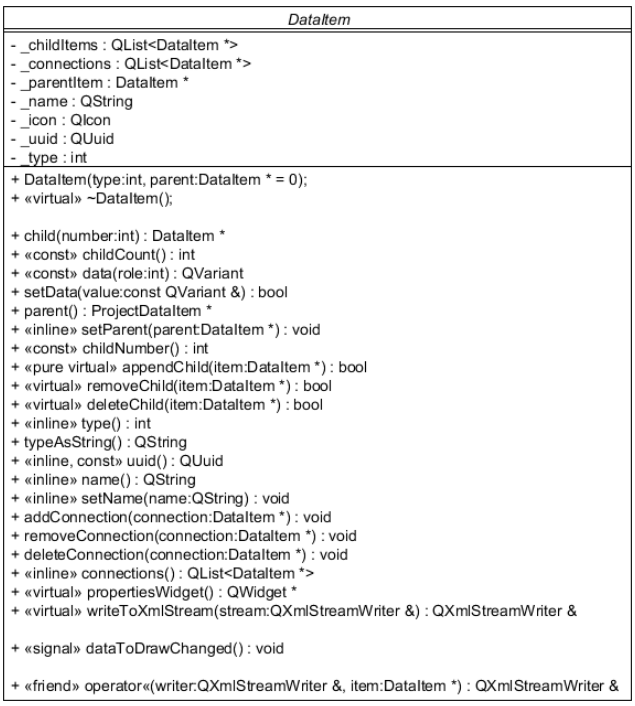


Diagramm 2: DataItem Klasse

Es handelt sich um eine Basisklasse, welche von *QObject* erbt, da der Signal-Slots-Mechanismus verwendet wird.

Das Eltern-Element, der Name, das Icon und die UUID sind in den privaten Member-Variablen `_parentItem`, `_name`, `_icon` und `_uuid` gespeichert. Das Icon wird später zur Anzeige in der Projekt-Ansicht (§ 2.1.1 Die Projekt-Ansicht) verwendet. Mit der UUID soll das Element eindeutig gekennzeichnet werden.

Ebenfalls der Typ des Elements ist in einem privaten Attribut abgelegt. Das Attribut `_childItems` enthält eine Liste von Zeigern auf die Kind-Elemente, die ein Element besitzt. Die Liste `_connections` enthält Zeiger auf die Transitionen, die auf ein Element führen oder es verlassen.

Die Methoden erlauben folgende wichtige Funktionen:

- `child()` und `childCount()` erlauben dem Modell Informationen zu den in einem Element enthaltenen Kind-Elementen zu erhalten.
- Informationen zum Namen und zum Icon können mit `data()` erhalten werden.
- Um einem Element ein Kind hinzuzufügen, wird `appendChild()` verwendet. Zum Entfernen eines Kindes dient `removeChild()` bzw. `deleteChild()`.
- Zum Hinzufügen/Entfernen von eingehenden und ausgehenden Transition wird `addConnection()` und `removeConnection()` bzw. `deleteConnection()` verwendet.
- `propertiesWidget()` erstellt ein Property-Widget (§ 2.3 Der Property-Editor) für das Element.
- Der Typ eines Elements kann mit `type()` ermittelt werden.

3.2.2 Implementierung

In diesem Abschnitt werden sämtliche Methoden der Klasse beschrieben. Wichtige Codesegmente sind direkt in diesem Abschnitt zu finden. Der Programmcode der vollständigen Implementierung dieser Klasse ist elektronisch unter

- ../Software/StateChartEditor/src/DataModel
- ../Software/StateChartEditor/include/DataModel

zu finden.

Der Konstruktor erstellt ein neues Element, setzt dessen Eltern-Element, Typ und UUID.

```
DataItem(int type, QUuid uuid, DataItem *parent = 0);
```

Beim Destruktor dieser Basisklasse handelt es sich um den Default-Destruktor.

```
virtual ~DataItem();
```

Die Funktion `child()` gibt das Kind-Element, das sich an der Position `number` in der Liste des Elements befindet, zurück.

```
DataItem *child(int number);
```

Um die Anzahl Kinder, die ein Element besitzt, zu ermitteln, dient `childCount()`.

```
int childCount() const;
```

`data()` liefert der `role` entsprechend Informationen des Elements. Ist `role` gleich `Qt::DisplayRole` wird der Name des Elements zurückgegeben, ist `role` gleich `Qt::DecorationRole` das Icon.

```
QVariant data(int role) const;
```

`setData()` erlaubt es den Namen des Elements zu setzen.

```
bool setData(const QVariant &value);
```

Die Funktion `parent()` gibt das Eltern-Element des Elements zurück.

```
DataItem *parent();
```

Mit der Funktion `setParent()` lässt sich das Eltern-Element setzen.

```
inline void setParent(DataItem* parent) { _parentItem = parent; }
```

`childNumber()` gibt die Position des Elements in der Liste des Eltern-Elements zurück.

```
int childNumber() const;
```

`appendChild()` fügt dem Element ein Kind hinzu. Diese Funktion ist rein virtuell und muss in der abgeleiteten Klasse implementiert werden, denn nicht jedes Element kann dieselben Typen von Elementen enthalten.

```
virtual bool appendChild(DataItem* item) = 0;
```

Mit `removeChild()` lässt sich ein Kind-Element entfernen, wird aber nicht gelöscht.

```
virtual bool removeChild(DataItem* item);
```

`deleteChild()` entfernt wie `removeChild()` das Kind-Element, dieses wird aber zusätzlich gelöscht.

```
virtual bool deleteChild(DataItem* item);
```

Der Typ des Elements kann mit `type()` ermittelt werden.

```
inline int type() { return _type; }
```

`typeAsString()` liefert den Typ des Elements als String. Diese Methode ist nützlich, wenn das Modell gespeichert wird.

```
QString typeAsString();
```

`uuid()` gibt die UUID des Elements zurück.

```
inline QUuid uuid() const { return _uuid; }
```

`name()` gibt den Namen des Elements zurück.

```
inline QString name() { return _name; }
```

Die Funktion `setName()` erlaubt es, den Namen des Elements zu ändern.

```
inline void setName(QString name) { _name = name; }
```

Mit der Funktion addConnection() lässt sich ein Zeiger auf eine eingehende bzw. ausgehende Transition hinzufügen. Dabei muss überprüft werden, dass es sich beim angegebenen Parameter auch wirklich um eine Transition handelt.

```
void addConnection(DataItem* connection)
{
    if( connection->type() == DataItem::TransitionType )
        _connections.append(connection);
}
```

Mit `removeConnection()` lässt sich ein Zeiger auf eine Transition entfernen, wird aber nicht gelöscht.

```
void removeConnection(DataItem* connection);
```

`deleteConnection()` entfernt wie `removeConnection()` den Zeiger auf eine Transition, diese wird aber zusätzlich gelöscht.

```
void deleteConnection(DataItem* connection);
```

Die Funktion `connections()` gibt eine Liste aller eingehenden bzw. ausgehenden Transitionen des Elements zurück.

```
inline QList<DataItem*> connections() { return _connections; }
```

`propertiesWidget()` gibt den Property-Editor des Elements zurück. Diese Funktion muss für jedes Element entsprechend implementiert werden.

```
virtual QWidget* propertiesWidget();
```

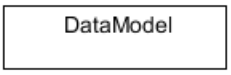
`writeToXmlStream()` schreibt die Informationen des Elements in einen Stream. Um spezifische Informationen eines Elements zu schreiben, muss diese Funktion in der abgeleiteten Klasse neu implementiert werden.

```
virtual QXmlStreamWriter& writeToXmlStream(QXmlStreamWriter& stream);
```

Das Signal `dataToDrawChanged()` wird gesendet, falls Informationen, die im Zustandsdiagramm angezeigt werden, ändern.

```
void dataToDrawChanged();
```

3.3 Die Schnittstelle (Modell)



3.3.1 Beschreibung

Die Schnittstelle (Modell) implementiert die Qt-Klasse *QAbstractItemModel*. Diese Klasse definiert ein Interface, welches es den unterschiedlichen Views in Qt erlaubt, auf die Elemente des Datenmodells zuzugreifen und deren Informationen anzuzeigen.

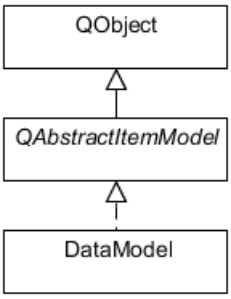


Diagramm 3: Implementierung von *QAbstractItemModel* durch das DataModel

Die Elemente selber werden nicht im Modell gespeichert, sondern sind im Speicher wie im Abschnitt § 3.1 beschriebener Struktur abgelegt.

Das Modell stellt lediglich eine Schnittstelle dar, welche es erlaubt, auf die Elemente in der Datenstruktur zuzugreifen, um diese in einer View anzeigen, bearbeiten oder neue Elemente hinzufügen zu können.

Damit das Modell auf die Datenstruktur Zugriff hat, besitzt das Modell einen Zeiger auf das „leere“ Root-Element der Struktur.

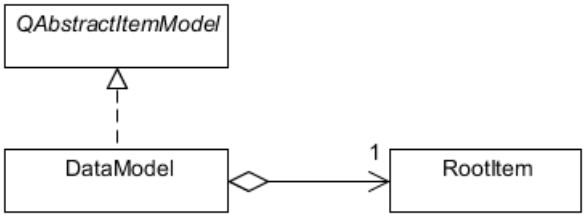


Diagramm 4: Die Schnittstelle (Modell) und ihr Zeiger auf das Root-Element

Folgende Abbildung zeigt die Schnittstelle zusammen mit der Datenstruktur, in welcher die Elemente abgelegt sind.

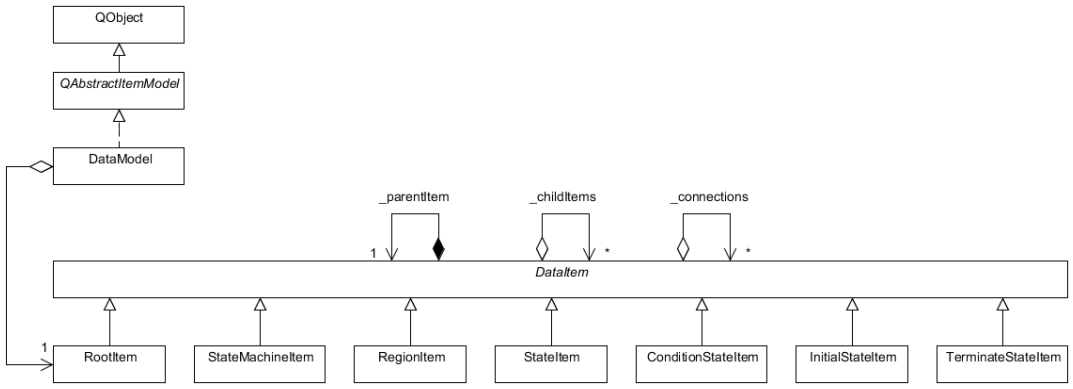


Diagramm 5: Die Schnittstelle zusammen mit der Datenstruktur

Die Schnittstelle ist wie folgt definiert.

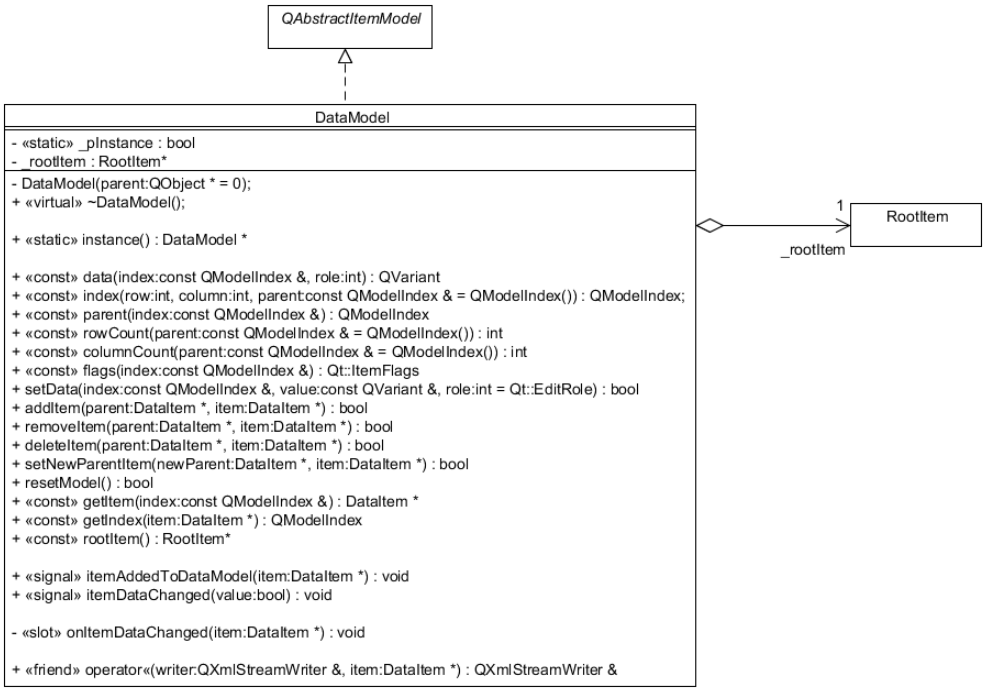


Diagramm 6: DataModel Klasse

3.3.2 Implementierung

Dieser Abschnitt beschreibt kurz die Methoden des Modells, welche von der Interfaceklasse *QAbstractItemModel* implementiert wurden und welche zusätzlich hinzugefügt worden sind.

Die vollständige Implementierung der Modell-Klasse befindet sich elektronisch unter:

- ../Software/SourceCode/src/DataModel
- ../Software/SourceCode/include/DataModel

Der Konstruktor, der ein neues Modell erstellt, ist als private deklariert, da es sich beim Modell um eine Singleton Klasse handelt.

```
DataModel(QObject *parent = 0);
```

Mit der statischen Methode `instance()` kann die Instanz des Modells erhalten werden.

```
static DataModel* instance();
```

Der Destruktor löscht das Root-Element und somit sämtliche Elemente. Der Speicher wird dadurch wieder freigegeben.

```
virtual ~DataModel();
```

Die Funktion `data()` erlaubt es, die der `role` entsprechenden Daten des Elementes, gegeben durch den `index`, zu erhalten. `Qt::DisplayRole` und `Qt::DecorationRole` sind möglich.

```
QVariant data(const QModelIndex &index, int role) const;
```

Ein Modell muss `index()` implementieren. Diese Funktion wird von den Views verwendet, um auf die Daten zugreifen zu können. Ein Index wird durch die Angabe der Zeile und Spalte, sowie durch den Index des Eltern-Elements erstellt (siehe § 3.1.1 Der Index (*QModelIndex*)).

Alle erstellen Indizes enthalten einen Zeiger auf das Element, wodurch der Zugriff auf dieses Element sichergestellt wird.

```
QModelIndex index(int row, int column, const QModelIndex &parent = QModelIndex())
const;
```

`parent()` gibt den Index des Eltern-Elements des Elements, welches durch `index` referenziert ist, zurück.

```
QModelIndex parent(const QModelIndex &index) const;
```

Die Funktion `rowCount()` retourniert die Anzahl Kind-Elemente, die das mit `parent` angegebene Element enthält.

```
int rowCount(const QModelIndex &parent = QModelIndex()) const;
```

`columnCount()` gibt immer 1 zurück, da es sich beim strukturellen Aufbau um eine Baumstruktur handelt.

```
int columnCount(const QModelIndex &parent = QModelIndex()) const;
```

`flags()` liefert die Eigenschaften, die das Modell unterstützt. Beispielsweise, dass die Elemente selektierbar sind. Diese Methode wird von den Views verwendet.

```
Qt::ItemFlags flags(const QModelIndex &index) const;
```

Die Funktion `setData()` wird zum Setzen der Daten, entsprechend der `role`, des Elements, angegeben durch `index`, verwendet.

```
bool setData(const QModelIndex &index, const QVariant &value, int role = Qt::EditRole);
```

`addItem()` fügt das Element `item` dem Element `parent` hinzu.

```
bool addItem(DataItem* parent, DataItem* item);
```

Mit `removeItem()` kann das Element `item` vom Element `parent` entfernt werden. Das Element `item` wird jedoch nicht gelöscht.

```
bool removeItem(DataItem* parent, DataItem* item);
```

`deleteItem()` entfernt gleich wie `removeItem()` das Element `item`. Zusätzlich wird das Element `item` gelöscht und dessen Speicher freigegeben.

```
bool deleteItem(DataItem* parent, DataItem* item);
```

Die Funktion `setNewParentItem()` erlaubt es, das Element `item` im Modell zu verschieben. Das heisst, es wird einem anderen Element (`newParent`) hinzugefügt.

```
bool setNewParentItem(DataItem* newParent, DataItem* item);
```

`resetModel()` entfernt alle Elemente. Nach dem Funktionsaufruf enthält das Modell nur noch das „leere“ Root-Element.

```
bool resetModel();
```

Ist der Index eines Elements bekannt, liefert `getItem()` einen Zeiger auf das Element, das durch `index` referenziert ist.

```
DataItem *getItem(const QModelIndex &index) const;
```

Umgekehrt wie bei `getItem()` kann mit `getIndex()` der Index eines Elements herausgefunden werden.

```
QModelIndex getIndex(DataItem* item) const;
```

`rootItem()` liefert einen Zeiger auf das „leere“ Root-Element.

```
inline RootItem* rootItem() const { return _rootItem; }
```

Das Signal `itemAddedToDataModel()` wird gesendet, wenn dem Datenmodell ein Element hinzugefügt wurde.

```
void itemAddedToDataModel(DataItem* item);
```

`itemDataChanged()` gibt an, dass die Daten eines Elementes geändert haben.

```
void itemDataChanged(bool value);
```

Dieser Slot dient dazu, das Datenmodell zu aktualisieren, falls die Daten eines Elements geändert haben, wie beispielsweise der Name.

```
void onItemDataChanged(DataItem* item);
```

3.4 Speichern und Laden des Datenmodells

Ein Datenmodell ist nutzlos, wenn seine Daten nicht gespeichert und zu einem späteren Zeitpunkt wieder geladen werden können.

Dieser Abschnitt zeigt, wie das Datenmodell mittels eines StreamWriters in ein IODevice geschrieben werden kann und wie das Datenmodell später mit einem DOM-Parser wieder geparst wird.

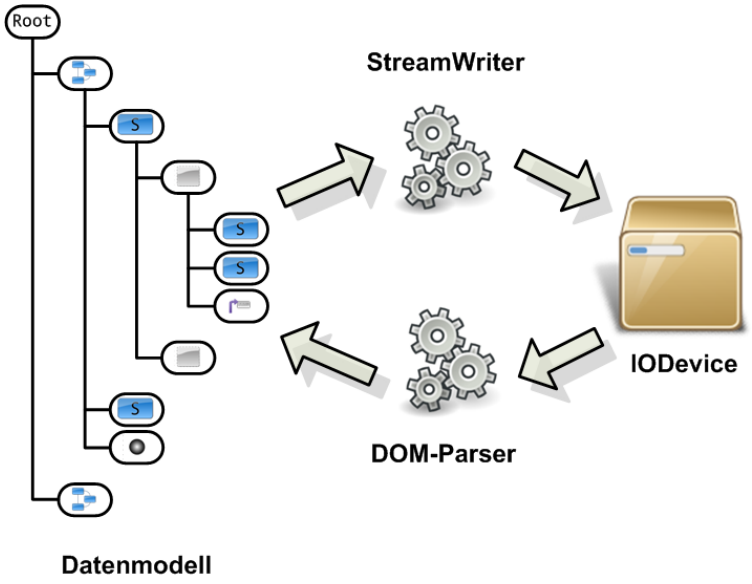


Abbildung 18: Speichern/Laden des Datenmodells

3.4.1 Speichern

Zum Speichern wird ein StreamWriter verwendet. Da es sich um ein strukturiertes Modell handelt, wird das Modell im XML-Format abgespeichert.

Damit das Datenmodell gespeichert werden kann, wird in der DataModel Klasse und in der Basisklasse der Elemente (DataItem) der <<-Operator des *QXmlStreamWriters* als Freundmethode hinzugefügt und überladen.

Folgender Code zeigt den <<-Operator des Modells. Es ist ersichtlich, dass vom <<-Operator der Basisklasse der Elemente Gebrauch gemacht wird.

```
QXmlStreamWriter& operator<<(QXmlStreamWriter& writer, DataModel* model)
{
    writer << model->rootItem();
    return writer;
}
```

Zudem wurde in der Basisklasse der Elemente die virtuelle Methode `writeToXmlStream()` erstellt, welche von jedem Elemente neu implementiert wird. In der Implementierung dieser Methode wird jeweils ein Tag für das Element erstellt und die Informationen geschrieben.

In der Implementierung des <<-Operators der Basisklasse der Elemente wird nun `writeToXmlStream()` aufgerufen und es wird die dem Element entsprechende Methode verwendet.

```
QXmlStreamWriter& operator<<(QXmlStreamWriter& writer, DataItem* item)
{
    return item->writeToXmlStream(writer);
}
```

Um schliesslich das gesamte Datenmodell in ein IODevice zu schreiben, sind folgende Anweisungen nötig.

```
DataModel* dataModel = DataModel::instance();

QXmlStreamWriter writer( &ioDevice );
writer << dataModel;
```

Die Implementierung der `writeToXmlStream()` Funktion ist jeweils in der Element Klasse zufinden. Die Klassen befinden sich elektronisch unter:

- ../Software/SourceCode/src/DataModel
- ../Software/SourceCode/include/DataModel

Im Anhang § 12.1.1 wird der Inhalt es IODevices erläutert.

3.4.2 Laden

Zum Laden des Datenmodells aus einem IODevice, wurde ein DOM-Parser (DataDomParser) erstellt. Dieser Parser erstellt anhand des IODevices ein DOM-Dokument, durchläuft dieses Dokument und erstellt Schritt für Schritt das Datenmodell.

Der DOM-Parser für das Datenmodell ist wie folgt definiert:

```
class DataDomParser
{
public:
    DataDomParser(DataModel* model);

    bool readFile(QIODevice* ioDevice);
    inline QHash<QUuid, DataItem*> itemsLoaded() { return _itemsLoaded; }

private:
    void parseProjectDataElement(const QDomElement &element);
    void parseStateMachineElement(const QDomElement &element,
                                  StateMachineItem* stateMachine);
    void parseStateElement(const QDomElement &element, StateItem* state);
    void parseRegionElement(const QDomElement &element, RegionItem* region);
    void parsePseudoStateElement(const QDomElement &element, DataItem* pseudoState);
    void parseTransitionElement(const QDomElement &element,
                                TransitionItem* transition);
    int getType(const QDomAttr attr);

private:
    DataModel* _model;                              //!< Reference to the data model.
    QHash<QUuid, DataItem*> _itemsLoaded;           //!< Map containing all items read.
};
```

Wird ein DOM-Parser erstellt, muss ein Zeiger auf das Modell mitgegeben werden, in welches die geladenen Daten geschrieben werden sollen. Danach kann mit `readFile()` der Ladevorgang gestartet werden.

Die Anweisungen um ein Datenmodell zu laden, würden wie folgt aussehen.

```
DataModel* dataModel = DataModel::instance();

DataDomParser parser( dataModel );
parser.readFile( &ioDevice );
```

Wiederum ist die vollständige Implementierung elektronisch unter

- ../Software/SourceCode/src/DataModel
- ../Software/SourceCode/include/DataModel

zu finden.

4 Die Entwicklung des Diagrammmodells

Es ist möglich, mit dem State Chart Editor mehrere Zustandsmaschinen zu zeichnen. Jede Zustandsmaschine entspricht einem Diagramm. Die Anzahl der Zustandsmaschinen ist vorgängig noch nicht bekannt.

Um die unterschiedlichen Diagramme (Zustandsmaschinen) besser verwalten zu können, wird ähnlich wie bei den Elementen (Datenmodell) ein Modell für die verschiedenen Diagramme entworfen.

Beim Diagrammmodell handelt es sich um ein einfaches Listenmodell, das sämtliche Diagramme als Liste darstellen lässt.

Bemerkung: Anders als beim Datenmodell, befinden sich die Daten beim Diagrammmodell direkt im Modell. Die Daten und Schnittstelle befinden sich in derselben Klasse.

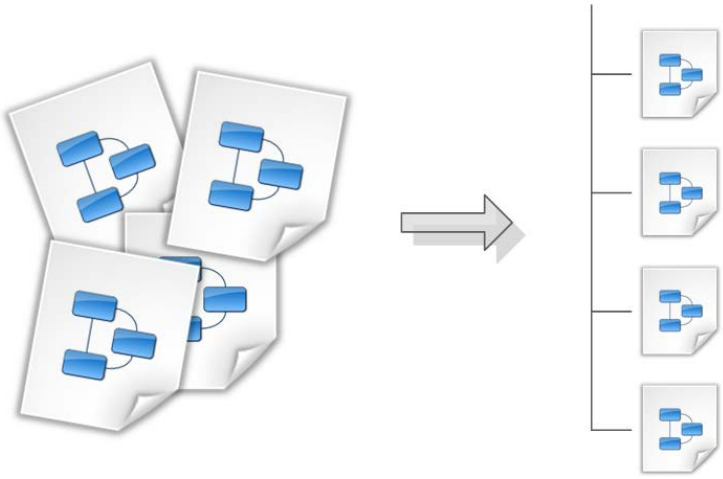


Abbildung 19: Prinzip des Diagrammmodells

Anschliessend wird kurz auf die Implementierung des Diagrammmodells eingegangen. Ähnlich wie beim Datenmodell wird zuerst der strukturelle Aufbau und schliesslich die Schnittstelle (Modell) beschrieben.

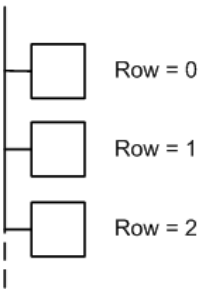
Zum Schluss wird gezeigt, wie die Daten des Diagrammmodells gespeichert und geladen werden können.

4.1 Der strukturelle Aufbau

Die Diagramme werden in einer Liste abgelegt. Anders als beim Datenmodel handelt es sich bei den Diagrammen immer um Top-Level Elemente und besitzen daher kein Eltern-Element.

Jedes Element kann durch einen Index (siehe § 3.1.1 Der Index (*QModelIndex*)) referenziert werden. Somit können die Views in Qt selbstständig auf die Daten zu greifen, um diese anzuzeigen.

Die untenstehende Abbildung zeigt den Aufbau der Liste.



Quelle: Qt Reference Documentation [3]

Abbildung 20: Repräsentierung der Diagramme in einer Liste

Die Diagramme werden direkt in einer Liste der Schnittstelle (Modell) abgelegt, wie das folgende Klassendiagramm zeigt.

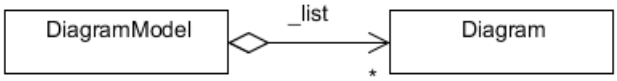
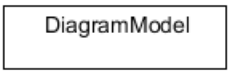


Diagramm 7: Klassendiagramm der Struktur des Diagrammmodells

4.2 Das Diagramm

Das Diagramm wird weiter unten im Abschnitt § 5 Die graphischen Elemente näher beschrieben.

4.3 Die Schnittstelle (Modell)



4.3.1 Beschreibung

Die Schnittstelle (Modell) implementiert die Qt-Klasse *QAbstractListModel*. Diese Klasse definiert ein Interface, das es den unterschiedlichen Views in Qt erlaubt, auf den Inhalt des Diagrammmodells zuzugreifen und diesen anzuzeigen.

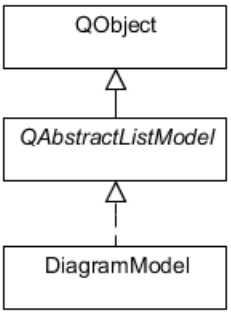


Diagramm 8: Implementierung von *QAbstractListModel* durch das DiagramModel

Folgendes Klassendiagramm zeigt das vollständige Diagrammmodell.

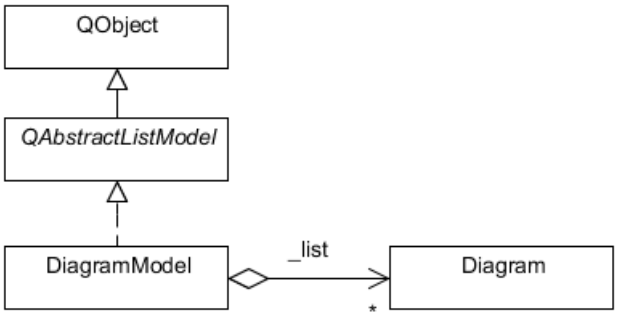


Diagramm 9: Klassendiagramm des Diagrammmodells

Das Diagrammmodell ist wie folgt definiert.

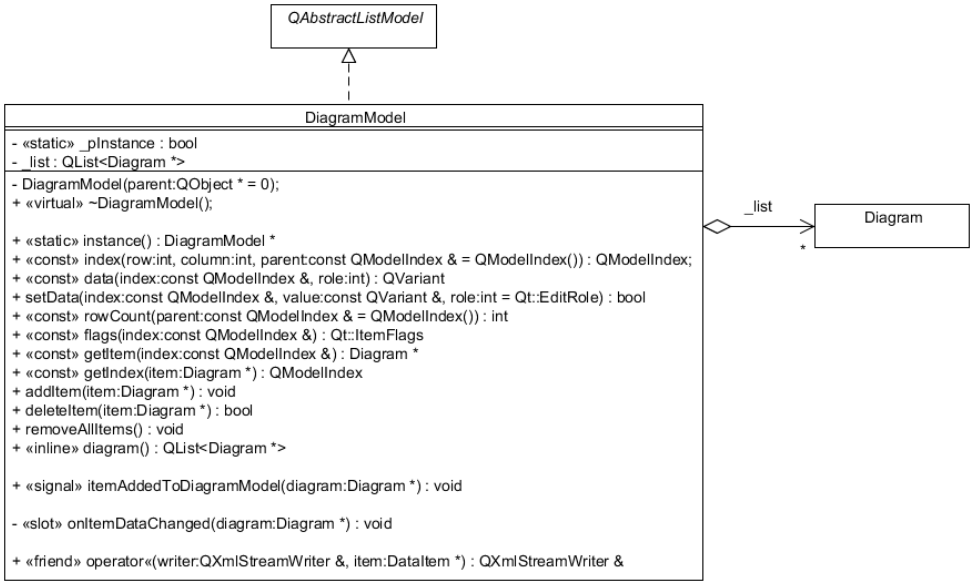


Diagramm 10: DiagramModel Klasse

4.3.2 Implementierung

Die einzelnen Methoden der DiagramModel Klasse werden in diesem Abschnitt kurz beschrieben.

Die vollständige Implementierung dieser Klasse ist elektronisch unter

- ../Software/SourceCode/src/DiagramModel
- ../Software/SourceCode/include/DiagramModel

zu finden.

Der Konstruktor, der ein neues Diagrammmodell erstellt, ist als private deklariert, da es sich beim Modell um eine Singleton Klasse handelt.

```
DiagramModel(QObject *parent = 0);
```

Die statische Methode `instance()` gibt die Instanz des Modells zurück.

```
static DiagramModel* instance();
```

Der Destruktor löscht das Diagrammmodell, sowie alle Diagramme und gibt deren Speicher wieder frei.

```
virtual ~DiagramModel();
```

Das Modell muss `index()` implementieren. Diese Funktion wird von den Views verwendet, um auf die Daten des Diagrammmodells zugreifen zu können. Alle erstellten Indizes enthalten einen Zeiger auf das Diagramm, wodurch der Zugriff auf dieses Diagramm sichergestellt wird.

Nähere Informationen zum Index wurden bereits im Kapitel § 3.1.1 Der Index (*QModelIndex*) gegeben.

```
QModelIndex index(int row, int column, const QModelIndex &parent) const;
```

Die Funktion `data()` gibt die der `role` entsprechenden Daten zurück. Im Fall des Diagrammmodells handelt es sich bei den Daten lediglich um den Namen (`Qt::DisplayRole`).

```
QVariant data(const QModelIndex &index, int role) const;
```

`setData()` ermöglicht es, den Namen des Modells zu setzen. Die entsprechende `role` dafür ist `Qt::DisplayRole`.

```
bool setData(const QModelIndex &index, const QVariant &value, int role = Qt::EditRole);
```

Die Funktion `rowCount()` retourniert die Anzahl Diagramme im Diagrammmodell. Als Parameter wird immer ein „leerer“ Index übergeben, da sämtliche Elemente der Liste kein Eltern-Element besitzen.

```
int rowCount(const QModelIndex &parent = QModelIndex()) const;
```

`flags()` liefert die Eigenschaften, die das Modell unterstützt. Beispielsweise, dass die Diagramme selektierbar sind. Diese Methode wird von den Views verwendet.

```
Qt::ItemFlags flags(const QModelIndex &index) const;
```

Ist der Index eines Diagramms bekannt, liefert `getItem()` einen Zeiger auf das Diagramm, das durch `index` referenziert ist.

```
Diagram *getItem(const QModelIndex &index) const;
```

Umgekehrt wie bei `getItem()` kann mit `getIndex()` der Index eines Diagramms ermittelt werden.

```
QModelIndex getIndex(Diagram* item);
```

Mit `addItem()` lässt sich dem Diagrammmodell ein Diagramm `item` hinzufügen.

```
void addItem(Diagram* item);
```

`deleteItem()` entfernt das Diagramm `item` vom Diagrammmodell und löscht es.

```
bool deleteItem(Diagram* item);
```

`removeAllItems()` entfernt alle Diagramme und löscht diese.

```
void removeAllItems();
```

Die Funktion `diagrams()` gibt eine Liste der Diagramme, die sich im Diagrammmodell befinden, zurück.

```
inline QList<Diagram*> diagrams() { return _list; }
```

Der Slot `onItemDataChanged()` dient dazu, das Diagrammmodell zu aktualisieren, falls die Daten eines Diagramms geändert haben, beispielsweise der Name.

```
void onItemDataChanged(bool value);
```

Das Signal `itemAddedToDiagramModel()` wird gesendet, wenn dem Diagrammmodell ein Diagramm hinzugefügt wurde.

```
void itemAddedToDiagramModel(Diagram* diagram);
```

4.4 Speichern und Laden

Gleich wie beim Datenmodell ist es nutzlos, wenn die Daten des Diagrammmodells nicht gespeichert werden können, um sie später wieder zu verwenden.

Folgend soll der Speicher- bzw. Ladevorgang des Diagrammmodells erläutert werden. Beide Vorgänge sind identisch zu jenen des Datenmodells. Das Diagrammmodell wird mittels eines StreamWriters in ein IODevice geschrieben und kann anschliessend mit einem DOM-Parser wieder gelesen werden.

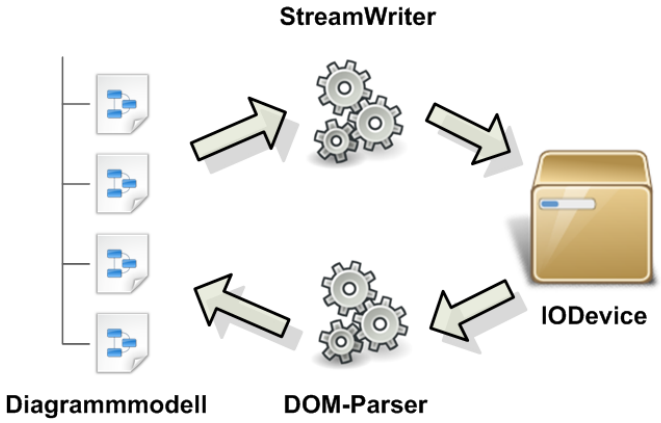


Abbildung 21: Speichern/Laden des Diagrammmodells

4.4.1 Speichern

Wie beim Datenmodell wird das Diagrammmodell im einem XML-Format abgespeichert. Dazu wird ein StreamWriter verwendet.

Damit das Diagrammmodell abgespeichert werden kann, wird in der DiagramModel Klasse und in der Diagram Klasse der <<-Operator des *QXmlStreamWriter* als Freundmethode hinzugefügt und überladen.

Folgende Code-Ausschnitte zeigen den <<-Operator des Diagrammmodells und des Diagramms.

```
QXmlStreamWriter& operator<<(QXmlStreamWriter& writer, DiagramModel* model)
{
    // start xml document
    writer.writeStartDocument( "1.0" );
    writer.setAutoFormatting(true);
    // create root item tag element
    writer.writeStartElement( "DiagramData" );
    // create new element tag for each child
    foreach(Diagram* item, model->_list)
    {
        writer << item;
    }
    writer.writeEndElement();
    writer.writeEndDocument();
    return writer;
}
```

```
QXmlStreamWriter& operator<<(QXmlStreamWriter& writer, Diagram* diagram)
{
    writer.writeStartElement( "Diagram" );
    writer.writeAttribute("uuid", diagram->_uuid.toString());
    // create new element tag for each child
    foreach(QGraphicsItem* item, diagram->items())
    {
        DrawItem* drawItem = dynamic_cast<DrawItem*>(item);
        if(drawItem && drawItem->parentItem() == NULL)
            writer << drawItem;
    }
    writer.writeEndElement();
    return writer;
}
```

Um schliesslich das gesamte Diagrammmodell in ein IODevice zu schreiben, sind folgende Anweisungen nötig.

```
DiagramModel* diagramModel = DiagramModel::instance();

QXmlStreamWriter writer( &ioDevice );
writer << diagramModel;
```

Die beiden Klassen DiagramModel und Diagram befinden sich elektronisch unter

- ../Software/SourceCode/src/DiagramModel
- ../Software/SourceCode/include/DiagramModel

bzw.

- ../Software/SourceCode/src/Diagram
- ../Software/SourceCode/include/Diagram

Im Anhang § 12.1.2 wird der Inhalt es IODevices erläutert.

4.4.2 Laden

Zum Laden des Diagrammmodells aus einem IODevice, wurde ebenfalls ein DOM-Parser (DiagramDomParser) erstellt. Dieser Parser erstellt anhand des IODevices ein DOM-Dokument, durchläuft es und erstellt Schritt für Schritt die Diagramme und fügt sie zum Diagrammmodell hinzu.

Der DOM-Parser (DiagramDomParser) ist wie folgt definiert:

```
class DiagramDomParser
{
public:
    DiagramDomParser(DiagramModel* model);

    bool readFile(QIODevice* ioDevice);
    inline QHash<QUuid, DrawItem*> itemsLoaded() { return _itemsLoaded; }
    inline QHash<QUuid, Region*> regionsLoaded() { return _regionsLoaded; }

private:
    void parseDiagramDataElement(const QDomElement &element);
    void parseDiagramElement(const QDomElement &element, Diagram* diagram);
    void parseDiagramItemElement(const QDomElement &element, Diagram* diagram,
                                 State* parentState = 0, Region* parentRegion = 0);
    void parseRegionItemElement(const QDomElement &element, Diagram* diagram,
                                State* parentState = 0, Region* parentRegion = 0);
    void parseConnectionItemElement(const QDomElement &element, Diagram* diagram,
                                    State* parentState = 0, Region* parentRegion = 0);
    int getType(const QDomAttr attr);
    bool linkConnection(ConnectionItem* connection,
                        QUuid startUuid, QPointF startPoint,
                        QUuid endUuid, QPointF endPoint);

private:
    DiagramModel* _model;                   //!< Pointer to the diagram model.
    QHash<QUuid, DrawItem*> _itemsLoaded;   //!< Map containing all items read.
    QHash<QUuid, Region*> _regionsLoaded;   //!< Map containing all regions read.
};
```

Dem DOM-Parser muss ein Zeiger auf das Diagrammmodell mitgegeben werden, in welches die geladenen Daten geschrieben werden sollen. Anschliessend kann mit `readFile()` der Ladevorgang gestartet werden.

Die Anweisungen um ein Diagrammmodell zu laden, sehen wie folgt aus.

```
DiagramModel* diagramModel = DiagramModel::instance();

DiagramDomParser parser( diagramModel );
parser.readFile( &ioDevice );
```

Die Implementierung des DiagramDomParser ist elektronisch unter

- ../Software/SourceCode/src/DiagramModel
- ../Software/SourceCode/include/ DiagramModel

zu finden.

5 Die graphischen Elemente

Dieser Abschnitt beschreibt, wie die Arbeitsfläche aufgebaut ist, um auf dieser Zustandsmaschinen zeichnen zu können. Zudem werden die graphischen Elemente, die zum Zeichnen verwendet werden, erläutert.

Bei der Arbeitsfläche, die sich im Central-Widget des Main-Windows befindet, handelt es sich um eine *QGraphicsView*, die es erlaubt, graphische Elemente anzuzeigen. Eine solche View zeigt jeweils ein Diagramm (*QGraphicsScene*) an. Falls mehrere Diagramme existieren, ist es nötig, der View anzugeben welches Diagramm anzuzeigen ist.

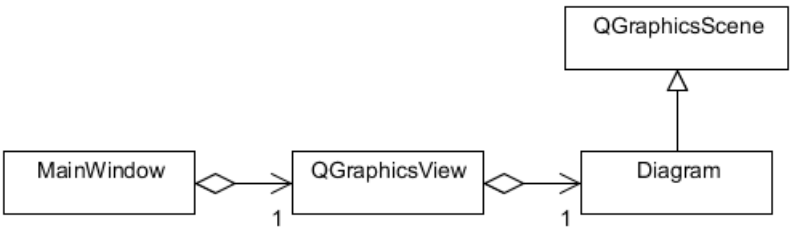


Diagramm 11: Klassendiagramm zum Aufbau der Arbeitsfläche des Main-Windows

5.1 Die Arbeitsfläche (QGraphicsView)

Die *QGraphicsView* Klasse erlaubt es, den Inhalt einer *QGraphicsScene* anzuzeigen. Um eine Szene zu visualisieren, muss dem *QGraphicsView* Objekt die Adresse der Szene mittels `setScene()` angegeben werden. Um die anzuzeigende Szene zu ändern, wird ebenfalls `setScene()` verwendet. [3]

Damit die View schliesslich angezeigt wird, ist der Aufruf von `show()` nötig.

Zum Beispiel:

```
QGraphicsScene scene;
...
<add something to the scene>
...

QGraphicsView view( &scene );
view.show();
...
view.setScene( &otherScene );       // set another scene
```

[3]

5.2 Das Diagramm (QGraphicsScene)

Folgend wird das Diagramm beschrieben und kurz auf dessen Implementierung eingegangen.

5.2.1 Beschreibung

Jede Zustandsmachine wird in einem Diagramm gezeichnet. Ein Diagramm wird durch die Klasse Diagram, welche von *QGraphicsScene* abgeleitet ist, repräsentiert.

Die Diagramme können auf der Arbeitsfläche angezeigt werden, wie es der vorherige Abschnitt beschreibt.

Eine *QGraphicsScene* kann *QGraphicsItem* enthalten, wie es das untenstehende Diagramm zeigt. Die in der Szene enthaltenen Elemente werden später in der View angezeigt.

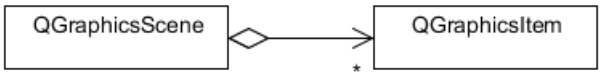


Diagramm 12: Eine *QGraphicsScene* kann *QGraphicsItem* enthalten

Somit ist es möglich, dem Diagramm später die verschiedenen Elemente, zum Zeichnen der Zustandsmaschine, hinzuzufügen.

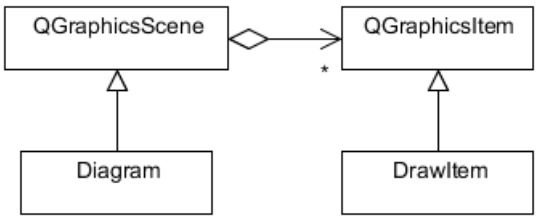


Diagramm 13: Ein Diagramm kann verschiedene graphische Elemente enthalten

5.2.2 Implementierung

Die Diagram Klasse ist wie folgt definiert.

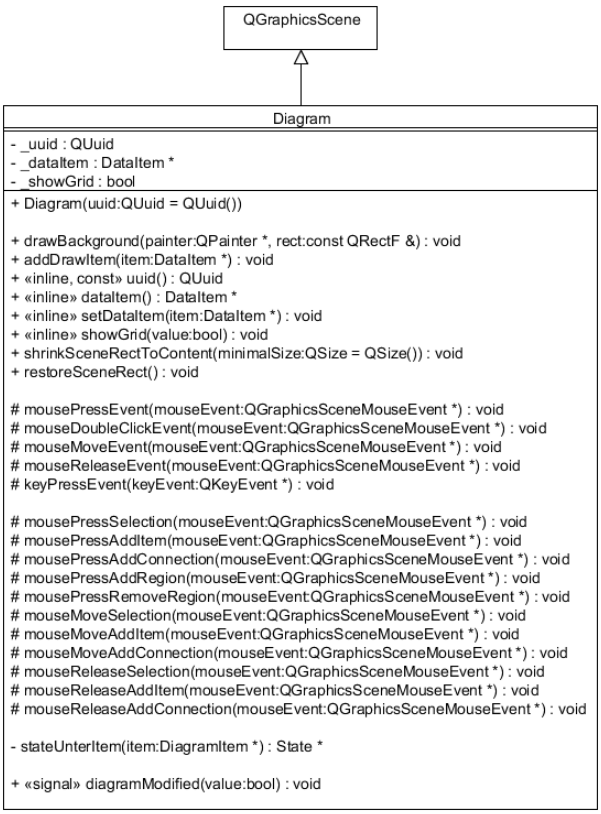


Diagramm 14: Diagram Klasse

Die logischen Eigenschaften des Diagramms sind in der StateMaschineItem Klasse abgelegt. Daher besitzt das Diagramm einen Zeiger `_dataItem` auf den logischen Teil.

Jedes Diagramm besitzt eine eindeutige UUID `_uuid`, die bei der Erstellung erzeugt wird, oder beim Laden gesetzt wird. Mit Hilfe dieser UUID ist es nach einem Ladevorgang möglich, den Link zwischen dem graphischen und logischen Element wiederherzustellen.

Der Konstruktor erstellt ein neues Diagramm mit der gegebenen UUID. Falls keine UUID angegeben wird, wird eine neue eindeutige UUID erzeugt.

```
Diagram(QUuid uuid = QUuid());
```

Die Funktion `drawBackground()` der *QGraphicsScene* Klasse wurde neu implementiert, um in Funktion des `_showGrid` Flags ein Raster im Diagramm anzuzeigen oder nicht. [6]

```
void drawBackground(QPainter *painter, const QRectF &rect);
```

Um dem Diagramm ein graphisches Element hinzuzufügen, wird `addDrawItem()` verwendet. Gleichzeitig wird das Diagramm aktualisiert.

```
void addDrawItem(DrawItem* item);
```

`uuid()` gibt die UUID des Diagramms zurück.

```
inline QUuid uuid() const { return _uuid; }
```

Mit `dataItem()` erhält man einen Zeiger auf das Objekt, das die logischen Informationen des Diagramms enthält.

```
inline DataItem* dataItem() { return _dataItem; }
```

Die Funktion `setDataItem()` erlaubt es, den Zeiger auf das Objekt, das den logischen Teil enthält, zu setzen.

```
inline void setDataItem(DataItem* item) { _dataItem = item; }
```

`showGrid()` erlaubt es, den Raster ein- bzw. auszuschalten.

```
inline void showGrid(bool value) { _showGrid = value; }
```

Jede *QGraphicsScene* besitzt eine Eigenschaft, die das Rechteck der Szene, das Begrenzungsrechteck, enthält. Dieses Rechteck definiert den Umfang der Szene und wird in erster Linie durch *QGraphicsView* verwendet, um den scrollbaren Bereich zu bestimmen. [3]

Wird das Rechteck der Szene nicht gesetzt, ist das Rechteck so gross, wie das Begrenzungsrechteck aller Elemente, die sich in der Szene befinden. Beim Hinzufügen von weiteren Elementen, wird das Rechteck automatisch vergrössert. Das Rechteck wird aber nie automatisch verkleinert. [3]

`shrinkSceneRectToContent()` erlaubt es nun, das Rechteck der Szene auf den Inhalt zu verkleinern.

```
void shrinkSceneRectToContent(QSize minimalSize = QSize());
```

Wurde das Rechteck der Szene einmal gesetzt, wird die automatische Vergrösserung nicht mehr unterstützt. Durch den Aufruf von `restoreSceneRect()`, kann diese Funktion wieder aktiviert werden.

```
void restoreSceneRect();
```

Die Methoden zu den Handlern von Mouse- und Key-Events werden im Abschnitt über die Implementierung des Verhaltens (§ 6.3) näher beschrieben.

Die vollständige Implementierung der Diagram Klasse ist elektronisch unter

- ../Software/SourceCode/src/Diagram
- ../Software/SourceCode/include/Diagram

zu finden. Die Klasse (StateMaschineItem) die den logischen Teil des Diagramms beinhaltet, unter:

- ../Software/SourceCode/src/DataModel
- ../Software/SourceCode/include/DataModel

5.3 Die Elemente – Eine allgemeine Beschreibung

Die graphischen Elemente werden von *QGraphicsItem* abgeleitet. *QGraphicsItem* ist die Basisklasse aller graphischen Elemente in einer *QGraphicsScene*.

Um die Implementierung der graphischen Elemente zu vereinfachen, wurde eine Basisklasse (DrawItem) für diese Elemente erstellt. Dazu kommen zwei weitere Klassen, die die Zustandsmaschinenelemente und Verbindungen (Transitionen) zusammenfassen.

In den folgenden Abschnitten werden die Basisklassen und anschliessend sämtliche Elemente beschrieben.

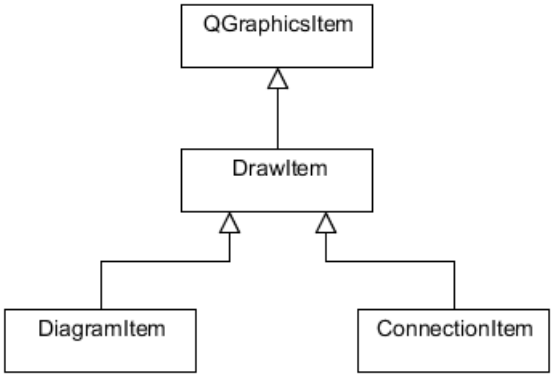


Diagramm 15: Basisklassen der graphischen Elemente

5.3.1 Das *QGraphicsItem*

Das *QGraphicsItem* ist die Basisklasse aller graphischen Elemente, die einer *QGraphicsScene* hinzugefügt werden können. Es werden nun kurz die meistbenutzten Funktionen dieser Klasse beschrieben.

Alle geometrischen Informationen eines *Items* basieren auf dem lokalen Koordinatensystem des Systems. Die Position des *Items*, `pos()`, ist die einzige Funktion, die nicht im lokalen Koordinatensystem arbeitet, sondern die Position im Koordinatensystem des *Parent Item*, falls es eines besitzt, zurückgibt. `scenePos()` gibt jedoch immer die Position des *Items* im lokalen Koordinatensystem zurück.

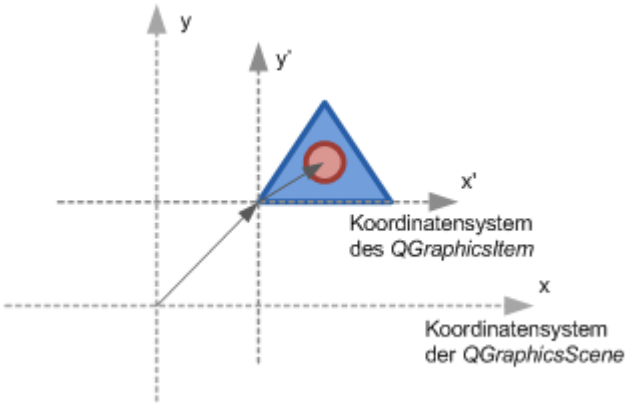


Abbildung 22: Die Koordinatensysteme

Mit `setFlag()` lassen sich Eigenschaften des *QGraphicsItem* setzen. Zum Beispiel, dass das *Item* auswählbar ist (`QGraphicsItem::ItemIsSelectable`) oder das es verschiebbar ist (`QGraphicsItem::ItemIsMovable`).

Um ein eigenes graphisches Element zu erstellen, wird die Klasse von *QGraphicsItem* abgeleitet. Anschliessend müssen mindestens die zwei rein virtuellen Funktionen `boundingRect()` und `paint()` implementiert werden.

Die Funktion `boundingRect()` definiert das Begrenzungsrechteck des Elements. Das Element muss innerhalb dieses Rechtecks gezeichnet werden. *QGraphicsView* nutzt diese Funktion, um festzustellen, ob das Element neu gezeichnet werden muss.

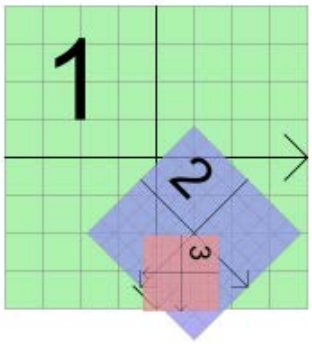
Abbildung 23: Begrenzungsrechteck eines graphischen Elements (boundingRect)

Optional ist es möglich `shape()` zu implementieren. Diese Funktion gibt, anders als `boundingRect()`, die exakte Kontur des Elements zurück. Sie wird von der *QGraphicsScene* verwendet, um beispielsweise festzustellen, ob ein *Item* angeklickt wurde. Die Standardimplementierung von `shape()` gibt das Begrenzungsrechteck zurück.

Abbildung 24: Kontur eines graphischen Elements (shape)

Die Funktion `paint()`, welche hauptsächlich von *QGraphicsView* aufgerufen wird, zeichnet den Inhalt des *Items* im lokalen Koordinatensystem.

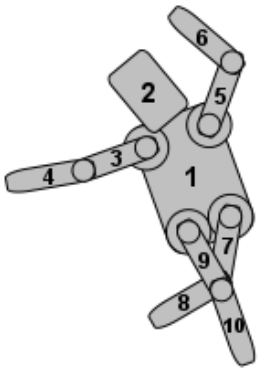
Items können andere *Items* enthalten, sowie auch von anderen *Items* enthalten werden. Alle *Items* haben ein *Parent Item* und können eine Liste von *Child Items*. Ein *Item*, ausser wenn es kein *Parent Item* besitzt, ist immer im lokalen Koordinatensystem des *Parent Items* positioniert. *Parent Items* propagieren ihre Position und Transformation für alle *Child Items*.



Quelle: Qt Reference Documentation [3]

Abbildung 25: Die Propagation der Koordinatensysteme

Ein *Child Item* wird zuoberst auf das *Parent Item* gestapelt und *Sibling Items* werden in der Reihenfolge, wie sie eingefügt worden sind, gestapelt.



Quelle: Qt Reference Documentation [3]

Abbildung 26: Stapelung der graphischen Elemente

`setZValue()` erlaubt es *Items* explizit auf dem Stapel zu positionieren.

[3]

5.3.2 Die Basisklasse der graphischen Elemente

Beschreibung der DrawItem Klasse

Für alle graphischen Elemente, die in einer Zustandsmaschine gezeichnet werden können, wurde die Basisklasse DrawItem, welche von *QGraphicsItem* abgeleitet ist entworfen.

Diese Klasse ermöglicht es, den Eltern-Teil eines Elements und die Region, falls sich das Element in einer Region befindet, zu setzen. Ebenfalls ist der <<-Operator der *QXmlStreamWriter* Klasse überladen, damit sich die graphischen Elemente, wie im Abschnitt § 4.4.1 beschrieben, speichern lassen. Die rein virtuelle Methode `writeToXmlStream()`, welche von den einzelnen Elementen implementiert wird, dient zum Schreiben der Informationen des Elements in ein IODevice.

Die DrawItem Klasse ist wie folgt definiert.

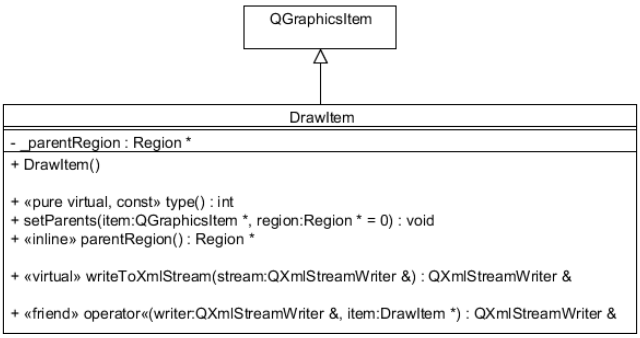


Diagramm 16: DrawItem Klasse

Die Implementierung dieser Klasse befindet sich elektronisch unter

- ../Software/SourceCode/src/Diagram
- ../Software/SourceCode/include/Diagram

Beschreibung der DiagramItem Klasse

Sämtliche Elemente einer Zustandsmaschine (State, Condition-State, …) werden von der Klasse DiagramItem abgeleitet.

Diese Klasse fasst die gemeinsamen Eigenschaften (z.B. Position und Dimension) zusammen, beinhaltet den relativen Ursprung des Elements, definiert die verschiedenen Elementtypen und die Methoden zum Zeichnen der Elemente. Sie enthält einen Zeiger auf die logischen Informationen des Elements und besitzt ein Attribut, das die UUID des Elements enthält.

Die Dimensionen eines Elements, unabhängig der Form, sind in der folgenden Abbildung dargestellt.

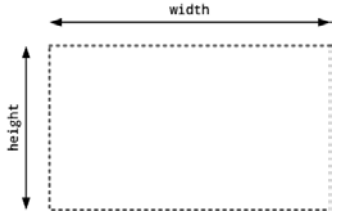
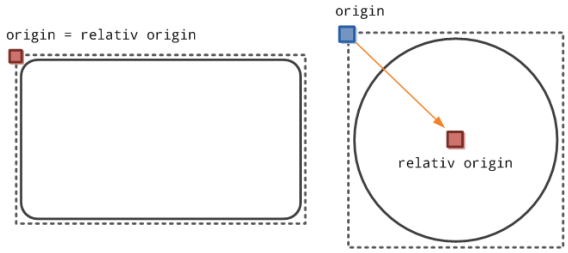


Abbildung 27: Dimensionen eines graphischen Elements



Der Ursprung befindet sich für alle Elemente in der oberen linken Ecke. Abhängig der Form, ist es eventuell erwünscht, den Ursprung anders zu wählen. Mit Hilfe des relativen Ursprungs kann nun angegeben werden, wie der Ursprung von der linken oberen Ecke entfernt ist. Folgende Abbildung versucht dies zu veranschaulichen.

Abbildung 28: Relativer Ursprung der graphischen Elemente

Bei Rechteckigen Elementen ist der Ursprung jeweils die linke obere Ecke und bei kreisförmigen Elementen der Mittelpunkt.

Implementierung der DiagramItem Klasse

Die Klasse ist, wie untenstehend gezeigt, definiert.

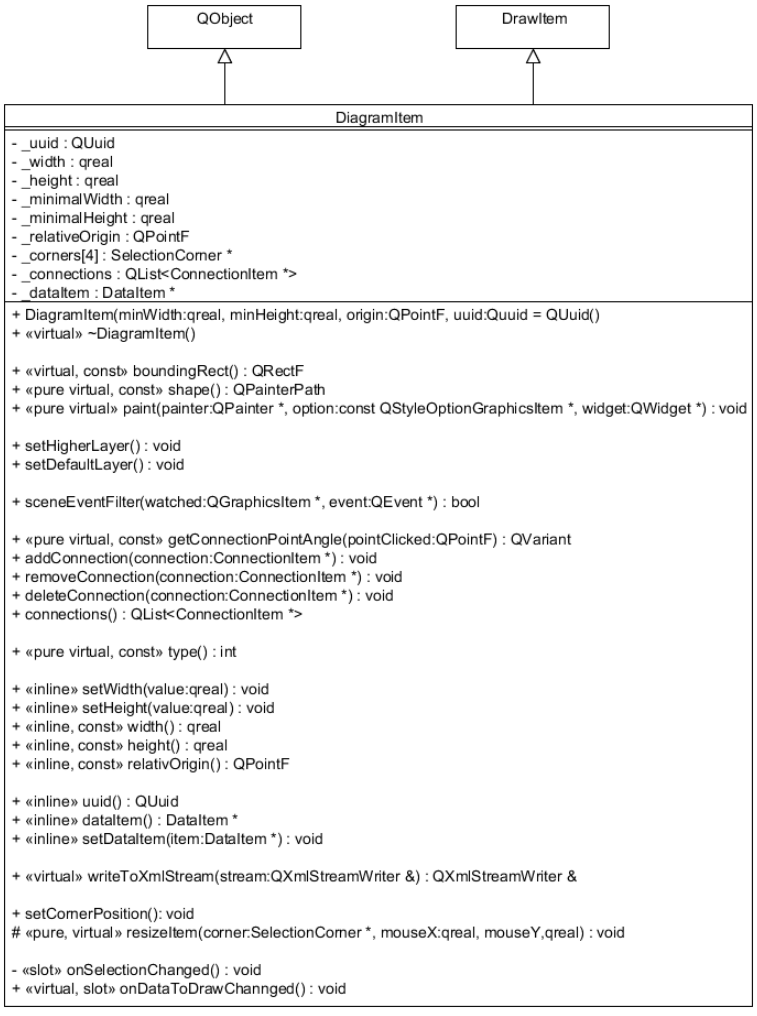


Diagramm 17: DiagramItem Klasse

Im Folgenden werden die Methoden der DiagramItem Klasse kurz beschrieben. Die vollständige Implementierung dieser Klasse ist elektronisch unter

- ../Software/SourceCode/src/Items
- ../Software/SourceCode/include/Items

zu finden.

Der Konstruktor erstellt ein neues graphisches Element. Die minimale Breite und Höhe, sowie der relative Ursprung des Elements werden gesetzt. Zudem kann dem Element eine UUID gegeben werden, wird keine angegeben, so wird eine neue UUID erstellt.

```
DiagramItem(qreal minWidth, qreal minHeight, QPointF origin, QUuid uuid = QUuid());
```

Der Destruktor löscht das Element und alle seine ein- / ausgehenden Transition, sofern vorhanden.

```
virtual ~DiagramItem();
```

Die Methoden `boundingRect()`, `shape()` und `paint()`, wie sie bereits im Abschnitt § 5.3.1 beschrieben wurden, werden von den abgeleiteten Klassen implementiert.

```
virtual QRectF boundingRect() const;
virtual QPainterPath shape() const = 0;
virtual void paint( QPainter *painter , const QStyleOptionGraphicsItem *option ,
                    QWidget *widget ) = 0;
```

Mit `setHigherLayer()` kann das Element in den Vordergrund gebracht werden und `setDefaultLayer()` platziert das Element auf den entsprechenden Layer des Element-Typs.

```
void setHigherLayer();
void setDefaultLayer();
```

Der `sceneEventFilter()` wird dazu verwendet, um Events eines anderen *QGraphicsItem* zu verarbeiten. Im Fall des DiagramItem werden die Events der Selection-Corner (siehe § 5.3.3) abgefangen, um die Grösse des Items zu verändern.

```
bool sceneEventFilter (QGraphicsItem * watched, QEvent * event);
```

`getConnectionPointAngle()` liefert in Funktion des gegebenen Punktes `pointClicked`, welcher sich innerhalb des Elements befindet, den Ankerpunkt bzw. den Winkel zur Berechnung des Ankerpunkts, um eine Transition anzufügen. Eine detailliertere Beschreibung folgt im Abschnitt § 6.3.3 Transitionen hinzufügen.

```
virtual QVariant getConnectionPointAngle(QPointF pointClicked) const = 0;
```

`addConnection()` dient dazu, um den Zeiger auf eine ein- / ausgehende Transition dem Element hinzuzufügen.

```
void addConnection(ConnectionItem* connection);
```

Die Funktion `removeConnection()` entfernt den Zeiger auf eine ein- / ausgehende Transition vom Element.

```
void removeConnection(ConnectionItem* connection);
```

`deleteConnection()` entfernt ebenfalls den Zeiger auf eine ein- / ausgehende Transition. Zusätzlich wird die Transition gelöscht.

```
void deleteConnection(ConnectionItem* connection);
```

`connections()` gibt eine Liste von allen Zeigern auf ein- / ausgehende Transitionen zurück.

```
QList<ConnectionItem*> &connections();
```

Der Typ des graphischen Elements kann mittels `type()` ermittelt werden.

```
virtual int type () const = 0;
```

Um die Breite und Höhe des Elements zu lesen bzw. schreiben, dienen die folgenden Getter- und Setter-Methoden.

```
inline void setWidth(qreal value) { _width = value; }
inline void setHeight(qreal value) { _height = value; }
inline qreal width() const { return _width; }
inline qreal height() const { return _height; }
```

`relativeOrigin()` gibt den relativen Ursprung des Elements zurück.

```
inline QPointF relativOrigin() const { return _relativOrigin; }
```

`setCornerPosition()` platziert die Selection-Corner (siehe § 5.3.3) des Elements.

```
void setCornerPosition();
```

Die Funktion `uuid()` gibt die UUID des Elements zurück.

```
inline QUuid uuid() const { return _uuid; }
```

Mit `dataItem()` und `setDataItem()` kann der Zeiger auf den logischen Teil des Elements gelesen bzw. gesetzt werden.

```
inline DataItem* dataItem() { return _dataItem; }
inline void setDataItem(DataItem* dataItem) { _dataItem = dataItem; }
```

`writeToXmlStream()` schreibt die Informationen des Elements in einen Stream. Um spezifische Informationen eines Elements zu schreiben, muss diese Funktion in der abgeleiteten Klasse neu implementiert werden.

```
virtual QXmlStreamWriter& writeToXmlStream(QXmlStreamWriter& stream);
```

Die Methode `resizeItem()` ändert die Grösse des Elements in Funktion des aktiven Selection-Corners und der aktuellen Mausposition. Diese Methode muss in den abgeleiteten Klassen implementiert werden.

```
virtual void resizeItem(SelectionCorner* corner, qreal mouseX, qreal mouseY) = 0;
```

Dieser Slot wird aufgerufen, wenn die Auswahl des Elements ändert. Die Selection-Corner (siehe § 5.3.3) werden angezeigt oder nicht.

```
void onSelectionChanged();
```

Dieser Slot wird aufgerufen, falls logische Informationen geändert haben, welche vom graphischen Element angezeigt werden. Das Element wird entsprechend aktualisiert.

```
virtual void onDataToDrawChanged();
```

Beschreibung der ConnectionItem Klasse

Die ConnectionItem Klasse fasst die wichtigsten Eigenschaften und Methoden einer Transition zusammen.

Die Klasse beinhaltet eine Struktur welche, abhängig von der Form der Source- und Target-Elementen, Informationen zu den Ankerpunkten, sowie Zeiger zu dem Source- bzw. Target-Element enthält. Weiter besitzt diese Klasse eine Textbox, um das Verhalten der Transition anzuzeigen, eine UUID und einen Zeiger auf die logischen Informationen der Transition.

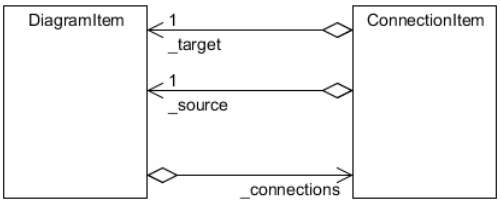


Diagramm 18: ConnectionItem mit Zeigern auf Source- und Target-Element

Die Informationen zu den Ankerpunkten sind abhängig von der Form des Elements. Bei rechteckigen Elementen wird der Ankerpunkt durch die Position des Elements und einem Faktor, der die Distanz zum Ankerpunkt in Funktion der Höhe / Breite des Elements angibt, festgelegt.

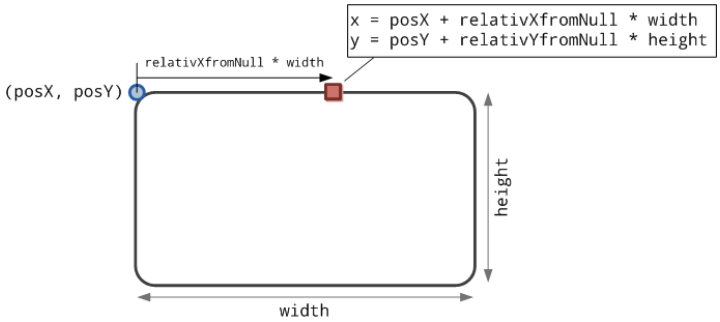


Abbildung 29: Ankerpunkt einer Transition bei einem rechteckigen Element

Bei kreisförmigen Elementen ergibt sich der Ankerpunkt aus Mittelpunkt und Radius des Elements, plus einem bestimmten Winkel.

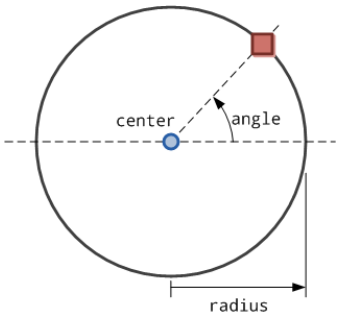


Abbildung 30: Ankerpunkt einer Transition bei einem kreisförmigen Element

Implementierung der ConnectionItem Klasse

Nachfolgend sollen die Methoden der ConnectionItem Klasse kurz beschrieben werden. Die vollständige Implementierung der Klasse befindet sich elektronisch unter

- ../Software/SourceCode/src/Connections
- ../Software/SourceCode/include/Connections

Die ConnectionItem Klasse ist, wie untenstehend gezeigt, definiert.

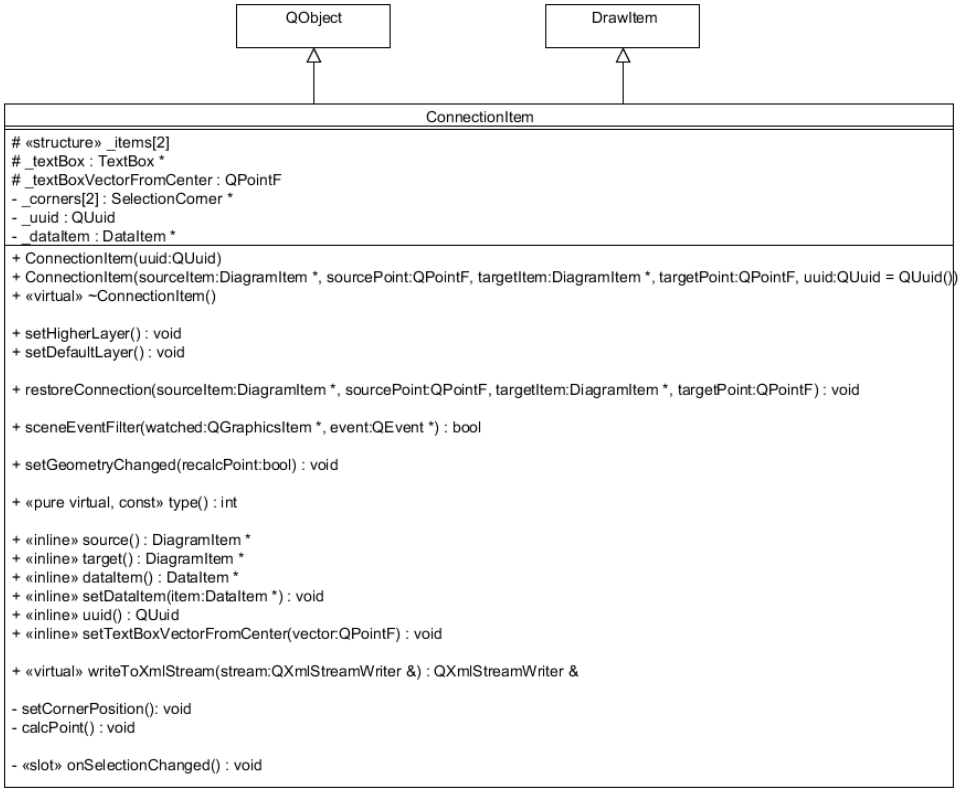


Diagramm 19: ConnectionItem Klasse

Der Konstruktor erstellt eine neue Verbindung mit der gegebenen UUID. Die Zeiger auf das Source- / Target-Element werden nicht gesetzt. Dieser Konstruktor wird verwendet, wenn ein Diagrammmodell geladen wird.

```
ConnectionItem(QUuid uuid);
```

Dieser Konstruktor erstellt ebenfalls eine neue Verbindung mit der gegebenen UUID, falls keine UUID gegeben ist, wird eine neue erstellt. Zudem werden die Ankerpunkte und die Zeiger zu Source und Target angegeben.

```
ConnectionItem(DiagramItem* sourceItem, QPointF sourcePoint,
               DiagramItem* targetItem, QPointF targetPoint,
               QUuid uuid = QUuid());
```

Der Destruktor löscht die Verbindung und wird aus den betroffenen Elementen entfernt.

```
virtual ~ConnectionItem();
```

Mit `setHigherLayer()` kann die Verbingung in den Vordergrund gebracht werden und `setDefaultLayer()` platziert die Verbindung Element auf den Layer der Verbindungen.

```
void setHigherLayer();
void setDefaultLayer();
```

`restoreConnection()` die dazu, nach einem Ladevorgang die Zeiger zu Source und Target, sowie die Ankerpuntke zu setzen.

```
void restoreConnection(DiagramItem* sourceItem, QPointF sourcePoint,
                       DiagramItem* targetItem, QPointF targetPoint);
```

Der `sceneEventFilter()` wird dazu verwendet, um Events eines anderen *QGraphicsItem* zu verarbeiten. Im Fall des ConnectionItem werden die Events der Selection-Corner (siehe § 5.3.3) abgefangen, um die Verbindung verschieben zu können.

```
bool sceneEventFilter(QGraphicsItem * watched, QEvent * event);
```

`setGeometryChanged()` aktualisiert die Verbindung. Mit `recalcPoint` kann zudem angegeben werden, ob die Ankerpunkte neu berechnet werden sollen.

```
void setGeometryChanged(bool recalcPoint);
```

`type()` liefert den Typ der Verbindung.

```
virtual int type () const = 0;
```

Mit `dataItem()` und `setDataItem()` kann der Zeiger auf den logischen Teil des Elements gelesen bzw. gesetzt werden.

```
inline DataItem* dataItem() { return _dataItem; }
inline void setDataItem(DataItem* dataItem) { _dataItem = dataItem; }
```

Die folgenden Getter-Methoden dienen zum Lesen des Zeigers von Source / Target und der UUID.

```
inline DiagramItem* source() { return _items[0].item; }
inline DiagramItem* target() { return _items[1].item; }
inline QUuid uuid() const { return _uuid; }
```

Mit der Funktion `setTextBoxVectorFromCenter()` lässt sich der Vektor vom Mittelpunkt der Verbindung zur Textbox setzen (siehe § 5.3.4 Textbox).

```
inline void setTextBoxVectorFromCenter(QPointF vector)
{ _textBoxVectorFromCenter = vector; }
```

`writeToXmlStream()` schreibt die Informationen der Verbindung in einen Stream. Um spezifische Informationen einer Transition zu schreiben, muss diese Funktion in der abgeleiteten Klasse neu implementiert werden.

```
virtual QXmlStreamWriter& writeToXmlStream(QXmlStreamWriter& stream);
```

`setCornerPosition()` platziert die Selection-Corner (siehe § 5.3.3).

```
void setCornerPosition();
```

Mit der Funktion `calcPoint()` werden die Ankerpunkte berechnet.

```
void calcPoint();
```

Dieser Slot wird aufgerufen, wenn die Auswahl der Verbindung ändert. Die Selection-Corner (siehe § 5.3.3) werden angezeigt oder nicht.

```
void onSelectionChanged();
```

5.3.3 Selection-Corner

Um Anzeigen zu können, ob ein Element oder eine Transition markiert ist, wurden die Selection-Corner eingeführt.

Bei den Selection-Corners handelt es sich um Markierungen, die in den Ecken eines Elements oder an den Enden einer Transition angezeigt werden, wenn das Element oder die Transition markiert ist.

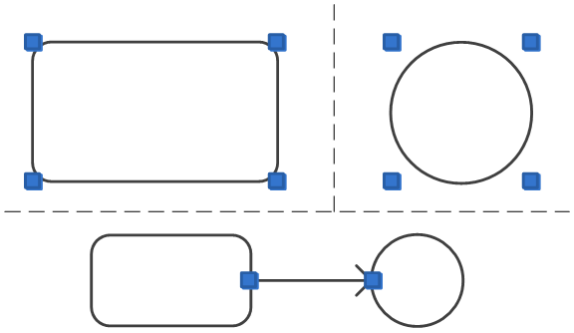


Abbildung 31: Die Selection-Corner bei markierten Elementen / Transitionen

Wie die obenstehende Abbildung zeigt, besitzen die Elemente 4 und die Transitionen 2 Selection-Corner.

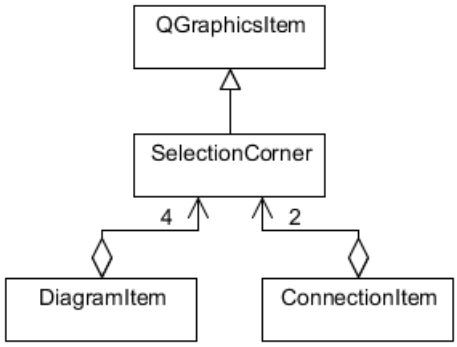


Diagramm 20: Elemente und Transitionen mit Selection-Corner

Die Selection-Corner besitzen zudem eine Mouse-Over Detektion, um anzuzeigen, dass sich die Maus über einem Corner befindet. Der Corner färbt sich rot.

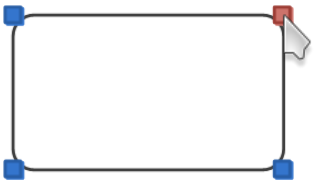


Abbildung 32: Selection-Corner und Mouse-Over Detektion

Zusätzlich werden die Selection-Corner dazu verwendet, um die Grösse eines Elements zu ändern. Beim Drücken der linken Maustaste auf einem Selection-Corner wird die Mausposition im Corner abgespeichert. Durch installieren eines *SceneEventFilter* in der Element-Klasse, kann nun die Grössenänderung implementiert werden. Nähere Informationen zum Ändern der Elementgrösse sind im Abschnitt § 6.3.6 (S. 60) zu finden.

Die Selection-Corner wurden auf der Basis des Beispiels in der Referenz [5] erstellt.

Die Implementierung der SelectionCorner Klasse ist elektronisch unter

- ../Software/SourceCode/src/Diagram
- ../Software/SourceCode/include/Diagram

zu finden.

5.3.4 Textbox

Um auf einer Transition Text anzuzeigen, kann der Transition eine Textbox hinzugefügt werden.

Die Textbox wird standardmässig auf den Mittelpunkt der Transition gesetzt. Sie kann aber mit Hilfe der Maus an eine beliebige Stelle verschoben werden. Die Position der Textbox ist somit durch den Mittelpunkt einer Transition und einen Vektor definiert (Abbildung 33).

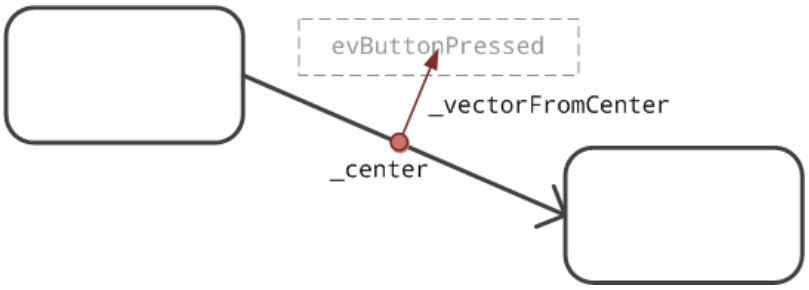


Abbildung 33: Positionierung der Textbox

Die Klasse TextBox ist wie folgt definiert:

```
class TextBox : public QObject, public QGraphicsItem
{
    Q_OBJECT
    Q_INTERFACES( QGraphicsItem )

public:
    TextBox(QGraphicsItem* parent);
    QRectF boundingRect() const;
    QPainterPath shape() const;
    void paint( QPainter *painter , const QStyleOptionGraphicsItem *option ,
                QWidget *widget );
    void setText(QString text);
    void adjustPosition(QPointF center);
    inline QPointF getVectorFromCenter() { return _vectorFromCenter; }
    inline void setVectorFromCenter(QPointF vector) { _vectorFromCenter = vector; }

protected:
    void mouseMoveEvent(QGraphicsSceneMouseEvent *mouseEvent);

private:
    QString _text;                          //!< The text contained by the text box.
    qreal _width;                           //!< Width of the text box.
    qreal _height;                          //!< Height of the text box.
    QPointF _center;                        //!< Transition's center point.
    QPointF _vectorFromCenter;              //!< Vector from transition's center to the
                                                text box.
};
```

In den Attributen `_center` und `_vectorFromCenter` sind die Informationen zur Positionierung der Textbox abgelegt. `_width` und `_heigth` geben die Dimension der Textbox an und `_text` enthält den anzuzeigenden Text.

`boundingRect()` und `shape()` wurden bereits im Abschnitt § 5.3.1 erklärt. Die Methode `paint()` zeigt den Text an der entsprechenden Position an, dazu werden zuerst die Dimensionen des Texts berechnet.

Wird die Transition verschoben oder geändert, so kann mit `adjustPosition()` die Textbox wieder richtig ausgerichtet werden.

Zudem beinhaltet die Klasse einige Getter- und Setter-Methoden.

Die Implementierung der Klasse befindet sich elektronisch unter:

- ../Software/SourceCode/src/Connections
- ../Software/SourceCode/include/Connections

5.4 Die Elemente

Aufbauend auf die Basisklassen DiagramItem und ConnectionItem können nun die verschiedenen Elemente und Transitionen erstellt werden.

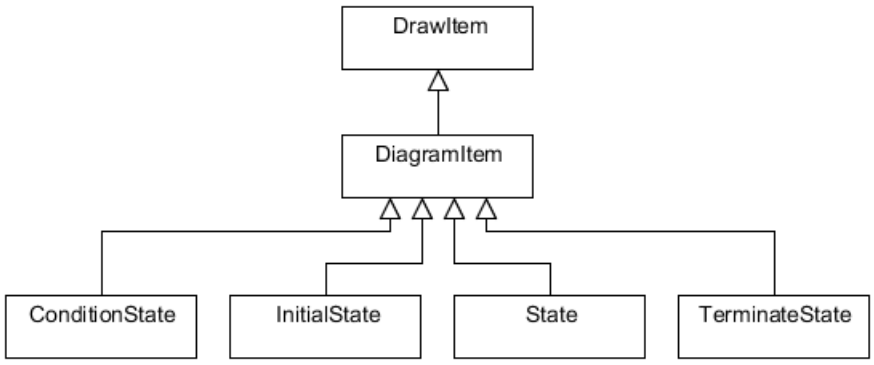


Diagramm 21: Die graphische Elemente abgeleitet von DiagramItem

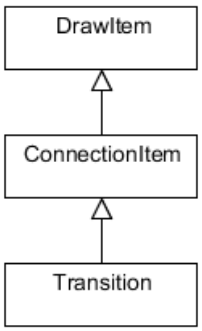


Diagramm 22: Die Transition abgeleitet von ConnectionItem

Die einzelnen Elemente und die Transition werden in den folgenden Abschnitten beschrieben. Bei der Implementierung wird nur auf die wichtigsten Punkte eingegangen. Die Implementierung der Elemente und der Transition sind elektronisch unter

- ../Software/SourceCode/src/Items
- ../Software/SourceCode/include/Items

bzw.

- ../Software/SourceCode/src/Connections
- ../Software/SourceCode/include/Connections

zu finden.

Zudem befindet sich elektronisch unter

- ../Software/SourceCode/documentation

eine Doxygen-Dokumentation zur gesamten Implementierung des State Chart Editors.

5.4.1 Initial-State, Terminate-State und Condition-State

Die Elemente Initial-State, Terminate-State und Condition-State weisen keine besonderen Eigenschaften auf.

Abbildung 34: Initial-, Terminate- und Condition-State

Zum Erstellen des graphischen Elements wird lediglich die `paint()` Methode überladen.

Die Methode `getConnectionPointAngle()` zur Ermittlung des Ankerpunkts der Transition wird im Abschnitt § 6.3.3 (S. 56) beschrieben.

5.4.2 State

Der Zustand wird durch ein abgerundetes Rechteck dargestellt, welches den Namen des Zustands enthält. Ein Zustand besitzt standardmässig eine Region, es besteht aber die Möglichkeit, weitere Regionen hinzuzufügen. Besitzt ein Zustand mehrere Regionen, werden diese durch eine vertikale Linie voneinander getrennt.

Ist das Verhalten eines Zustands implementiert, wird dieses durch kleine Indikatoren in der linken unteren Ecke angezeigt.

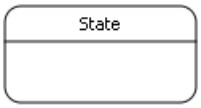


Abbildung 35: Standard-Zustand mit einer Region

Zum Zeichnen des Zustands wurde wiederum die `paint()` Methode überladen.

Action-Indikatoren

Wie bereits erwähnt, kann das Verhalten des Zustandes implementiert werden. Um im Diagramm anzuzeigen, dass das Verhalten (Action On Entry, usw.) implementiert ist, werden in der linken unteren Ecke Indikatoren angezeigt.

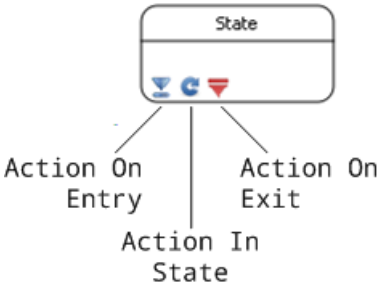


Abbildung 36: Zustand mit Action-Indikatoren

Um die Indikatoren anzuzeigen, wird in der `paint()` Methode überprüft, ob das Verhalten vorhanden ist oder nicht.

Regionen

Ein Zustand kann mehrere Regionen enthalten. Um dem Zustand eine Region hinzuzufügen, wird eine Region erstellt und mit `addRegion()` dem Zustand hinzugefügt. Mit `removeRegion()` bzw. `deleteRegion()` kann eine Region wieder vom Zustand entfernt werden.

Eine Region ist definiert durch eine Breite. Um die Regionen im Zustand anzuzeigen, werden alle Regionen, die der Zustand enthält, in der `paint()` Methode durchlaufen und mit Hilfe der Breite die vertikalen Begrenzungen gezeichnet.

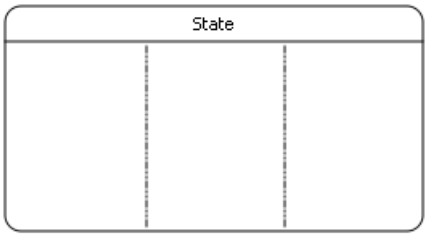


Abbildung 37: Zustand mit drei Regionen

Zum Ändern der Breite der Region wurden die Mouse-Events in der State Klasse überladen. Beim Drücken der linken Maustaste wird überprüft, ob sich die Maus auf einer Begrenzung befindet. Die Begrenzung wird danach mit der Maus zusammen verschoben.

Grössenänderung

Um die Grösse des Zustands zu ändern, wurde die Methode `resizeItem()` implementiert. Eine Beschreibung zur Grössenänderung folgt im Abschnitt § 6.3.6 (S. 60).

Zudem implementiert auch der Zustand die Methode `getConnectionPointAngle()` neu, welche zur Ermittlung des Ankerpunkts des Zustands dient. Diese wird im Abschnitt § 6.3.3 (S. 56) beschrieben.

5.4.3 Transition

Eine Transition wird durch eine Linie mit einer Pfeilspitze am Ende (Target-Element) dargestellt und verbindet ein Source- mit einem Target-Element.

Bei einer Transition können Trigger, Guard und Action implementiert werden. Falls eine Implementierung vorhanden ist, wird eine Textbox erstellt, die diese auf der Transition anzeigt.

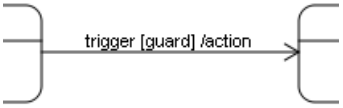


Abbildung 38: Transition mit Text

Die überladene `paint()` Methode zeichnet mit Hilfe der Ankerpunkte die Transition.

Die Implementierung zum Erstellen einer Transition wird im Abschnitt § 6.3 genauer behandelt.

6 Implementierung des State Chart Editors

Der letzte Abschnitt wird nun noch kurz auf die Implementierung der wichtigsten Elemente eingehen.

Zu Beginn wird aufgezeigt, wie das Main-Window (siehe Abschnitt § 2) implementiert wird. Es wird gezeigt, wie die Menüleiste, Toolbar und die verschiedenen Ansichten erstellt werden.

In einem nächsten Schritt wird erklärt, wie die verschiedenen Diagramme auf der Arbeitsfläche angezeigt werden können.

Zum Schluss wird die Implementierung der Bedienung erläutert. Es wird gezeigt, wie Diagramme hinzugefügt/entfernt und Zustandsmaschinen gezeichnet werden können. Es wird der Property-Editor beschrieben, auf den Speicher-/Ladevorgang und den Export eingegangen, ebenfalls auf das Kopieren in die Zwischenablage und das Drucken.

Bemerkung:
Der gesamte Source-Code des State Chart Editors ist elektronisch unter

- ../Software/SourceCode
- ../Software/SourceCode

zu finden.

Zudem befindet sich elektronisch unter

- ../Software/SourceCode/documentation

eine Doxygen-Dokumentation zum Source-Code des State Chart Editors.

Ein Windows-Release des State Chart Editors ist unter

- ../Software/Release (Win32)

zu finden.

6.1 Main-Window des State Chart Editors

Das Main-Window besitzt, wie es bereits beschrieben wurde, eine Menü- und Statusleiste, eine Toolbar, sowie verschiedene Ansichten. Zur Implementierung des Main-Windows wurde ausschliesslich die Qt-Library verwendet.

Das Main-Window wurde mit Hilfe der *QMainWindow* Klasse erstellt. Zur Implementierung der Menüleiste und den Menüs, wurde *QMenuBar* bzw. *QMenu* verwendet. Die Toolbar wurde mit *QToolBar* erstellt und bei den verschiedenen Ansichten handelt es sich um *QDockWidgets*. Die Statusleiste wird mittels *QStatusBar* dargestellt.

Folgend wird beschrieben wie die Menüleiste, Toolbar, die Ansichten und die Statusleiste in Qt erstellt werden.

Siehe auch:

- ../Software/SourceCode/src/UI
- ../Software/SourceCode/include/UI

6.1.1 Menüleiste

Das Main-Window (*QMainWindow*) besitzt bereits eine Menüleiste. Auf diese Menüleiste kann mittels der Methode `menuBar()` zugegriffen werden.

Mit Hilfe von `addMenu()` können dieser Menüleiste die verschiedenen Menüs (*QMenu)* hinzugefügt werden. Dazu werden die einzelnen Menüs im Konstruktor des Main-Windows erstellt und der Menüleiste hinzugefügt, wie es untenstehendes Beispiel zeigt.

```
QMenu *file = new QMenu(tr("&File"), this);
menuBar()->addMenu(file);
```

Die Menüeinträge werden schliesslich mit *QAction* erstellt und dem Menü hinzugefügt. Diese *QActions* können mit einem Slot verbunden und später, falls es erwünscht ist, auch in einer Toolbar verwendet werden. Einer *QAction* kann mit `setShortcut()` eine Tastenkombination zugewiesen werden.

```
QAction *exit = file->addAction(tr("E&xit"), this, SLOT(close()));
exit->setShortcut(Qt::CTRL + Qt::Key_Q);
```

6.1.2 Toolbar

Um dem Main-Window eine Toolbar hinzuzufügen wird im Konstruktor eine neue *QToolBar* erstellt. Dieser Toolbar können nun die *QActions*, die für die Menüeinträge erstellt wurden, angegeben werden.

```
QToolBar *toolbar = new QToolBar("Add Item", this);
toolbar->addAction(insertState);
```

Mit `setToolButtonStyle()` lässt sich das Erscheinungsbild der Toolbar bestimmen. So kann gewählt werden, dass in der Toolbar der Icon und der Text angezeigt werden.

```
toolbar->setToolButtonStyle(Qt::ToolButtonTextUnderIcon);
```

`addToolBar()` fügt die Toolbar im angegebenen Bereich schliesslich dem Main-Window hinzu.

```
addToolBar(Qt::TopToolBarArea, toolbar);
```

6.1.3 Die Ansichten

Die Projekt-Ansicht, State-Chart-Ansicht und der Property-Editor wurden mit Hilfe von *QDockWidgets* realisiert. Ein *QDockWidget* ist ein Fenster, das am Rand innerhalb des Main-Windows angedockt werden kann. Dem *QDockWidget* kann schliesslich ein beliebges Widget/View angegeben werden, das/die angezeigt werden soll.

Folgend ein Beispiel das zeigt, wie ein *QDockWidget* im Konstruktor des Main-Windows erstellt und wie es dem Main-Window hinzugefügt wird.

```
QDockWidget *dockWidget = new QDockWidget("Project View", this);
```

Die Funktion `setAllowedAreas()` gibt an, wo das *QDockWidget* angedockt werden darf. Mittels `addDockWidget()` wird das *QDockWidget* dem Main-Window hinzugefügt.

```
dockWidget->setAllowedAreas(Qt::LeftDockWidgetArea);

addDockWidget(Qt::LeftDockWidgetArea, dockWidget);
```

Mit `setWidget()` kann dem *QDockWidget* ein Widget zur Anzeige gesetzt werden.

Die Modelle

Die Projekt-Ansicht und State-Chart-Ansicht zeigen die beschriebenen Modelle an (siehe Abschnitt § 3 und § 4). Zum Anzeigen der Modelle muss zuerst das Modell und eine View erstellt werden. Schliesslich kann der View mit `setModel()` das Modell, das angezeigt werden soll, angegeben werden.

Die Projekt-Ansicht wird als Baumstruktur angezeigt. Die Anweisungen sehen wie folgt aus.

```
DataModel *dataModel = DataModel::instance();
QTreeView *treeView = new QTreeView();
treeView->setModel(dataModel);
projectViewDock->setWidget(treeView);
```

Um die Zustandsmaschinen anzuzeigen wird eine Listenansicht erstellt.

```
DiagramModel *diagramModel = DiagramModel::instance();
QListView *listView = new QListView();
listView->setModel(diagramModel);
stateChartViewDock->setWidget(listView);
```

6.1.4 Statusleiste

Auf die Statusleiste des Main-Windows kann mit `statusBar()` zugegriffen werden. Mittels `show()` wird diese angezeigt.

```
statusBar()->show();
```

Um Meldungen in der Statusleiste anzuzeigen, dient die Funktion `showMessage()`. Es kann der Text, sowie die Anzeigedauer angegeben werden.

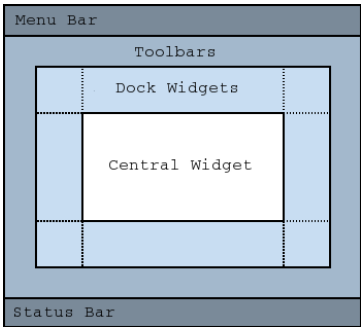
```
statusBar()->showMessage(tr("File saved"), 2000);
```

6.2 Die Arbeitsfläche

Die Arbeitsfläche wird im Central-Widget des Main-Windows erstellt. Es wird eine *QGraphicsView* erstellt, um den Inhalt einer *QGraphicsScene* (Diagramm) zu visualisieren.

Mit `setCentralWidget()` kann dem Central-Widget des Main-Windows diese View angegeben werden. Mit `setScene()` kann die *QGraphicsScene* angegeben werden.

```
QGraphicsView *view = new QGraphicsView(this);
setCentralWidget(view);
```



Quelle: Qt Reference Documentation [3]

Abbildung 39: Bereiche des Main-Windows

6.3 Die Bedienung des State Chart Editors

Dieser Abschnitt beschreibt die Hauptelemente der Implementierung der Bedienung des State Chart Editors.

Es wird beschrieben wie Diagramme erstellt, wie diese im Diagrammmodell abgelegt und wie sie wieder entfernt werden können.

Um das Zeichnen der Zustandsmaschinen zu vereinfachen, wurde die Editor Klasse entwickelt. Diese Klasse dient zum Bestimmen des Bearbeitungsmodus (Selektionieren, Hinzufügen, ...) und beinhaltet beim Hinzufügen von Elementen oder Transitionen eine Vorschau des Elements/Transition.

Mit Hilfe der Editor Klasse wird gezeigt, wie eine Zustandsmaschine in einem Diagramm gezeichnet werden kann. Dazu gehören das Hinzufügen von Elementen mit parallelem Hinzufügen des logischen Teils zum Datenmodell, das Erstellen von Transitionen, das Verschieben von Elementen, sowie das Entfernen von Elementen und Transitionen.

Ebenfalls das Ändern der Grösse eines Zustands und das Hinzufügen / Entfernen von Regionen in einem Zustand wird erläutert.

Am Ende wird beschrieben, wie der Property-Editor der einzelnen Elemente erstellt wird.

Bemerkung:

Zur Interaktion zwischen sämtlichen Komponenten des State Chart Editors wurde der Signal-Slot-Mechanismus von Qt verwendet. Sämtliche Signale und Slots sind in der Doxygen-Dokumentation (elektronisch unter ../Software/SourceCode/documentation) beschrieben.

6.3.1 Diagramm hinzufügen / entfernen / anzeigen

Um ein Diagramm hinzuzufügen, wird im Menü oder der Toolbar *Add New State Chart* gewählt.

Zum Entfernen eines Diagramms, wird dieses in der State-Chart-Ansicht ausgewählt und die Taste *Delete* gedrückt.

Hinzufügen

Beim Wählen von *Add New State Chart* wird der Slot `onNewStateChart()` der MainWindow Klasse aufgerufen.

Zum Hinzufügen eines Diagramms muss zuerst ein neues Diagramm (*QGraphicsScene*) und der logische Teil des Diagramms (*StateMachineItem*) erstellt und gelinkt werden. Anschliessend werden das Diagramm und der logische Teil in den entsprechenden Modellen abgelegt, Signale und Slots miteinander verbunden und das erstellte Diagramm in der Arbeitsfläche angezeigt.

Das `windowModified` Flag wird gesetzt, um dem Main-Window eine Änderung zu signalisieren.

Der folgende Programmcode zeigt die Implementierung des beschriebenen Slots.

```
void MainWindow::onNewStateChart()
{
    // create new diagram
    Diagram* diagram = new Diagram();

    DataModel* dataModel = DataModel::instance();

    // create new statemachine
    StateMachineItem* sm = new StateMachineItem(diagram->uuid());
    diagram->setDataItem(sm);
    // add diagram to data model
    dataModel->addItem(dataModel->rootItem(), sm);

    // add diagram to diagram model
    _diagramModel->addItem(diagram);
    _diagramView->selectionModel()->setCurrentIndex(_diagramModel->getIndex(diagram),
                                                    QItemSelectionModel::ClearAndSelect);

    connect(diagram, SIGNAL(selectionChanged()),
            this, SLOT(onItemSelectedInDiagramScene()));
    connect(diagram, SIGNAL(diagramModified(bool)),
            this, SLOT(setWindowModified(bool)));

    setWindowModified(true);

    _view->setScene(diagram);
}
```

Entfernen

Zum Entfernen eines Diagramms wurde der *Key-Event-Handler* der State-Chart-Ansicht implementiert.

Es wird überprüft, ob die *Delete* Taste gedrückt wurde. Beim Drücken der *Delete* Taste wird das selektionierte Diagramm aus beiden Modellen entfernt.

```
void DiagramView::keyPressEvent(QKeyEvent *keyEvent)
{
    QListView::keyPressEvent(keyEvent);
```

```
    if( keyEvent->key() == Qt::Key_Delete )
    {
        DiagramModel* diagramModel = DiagramModel::instance();
        Diagram* diagram = diagramModel->getItem(currentIndex());
        if( diagram )
        {
            DataModel* dataModel = DataModel::instance();
            dataModel->deleteItem(dataModel->rootItem(), diagram->dataItem());
            diagramModel->deleteItem(diagram);
            emit diagramDeleted();
        }
    }
}
```

Angezeigtes Diagramm wählen

Um ein Diagramm auf der Arbeitsfläche anzuzeigen, wählt man dieses in der State-Chart-Ansicht aus. Beim Selektionieren eines Diagramms in dieser Ansicht wird der Slot `onItemSelectedInDiagramModel()` aufgerufen. In diesem Slot wird auf das selektionierte Diagramm zugegriffen, dessen Property-Editor wird angezeigt und es wird der *QGraphicsView* der Arbeitsfläche zum Anzeigen angegeben.

Der untenstehende Programmcode zeigt, wie der `onItemSelectedInDiagramModel()` Slot implementiert ist.

```
void MainWindow::onItemSelectedInDiagramModel(QModelIndex index)
{
    Diagram *diagram = static_cast<Diagram*>(index.internalPointer());
    if ( diagram )
    {
        _propertyEditor->editStateMachine(diagram);
        _view->setScene(diagram);
        _view->scene()->clearSelection();
    }
}
```

6.3.2 Elemente hinzufügen

Um dem Zustandsdiagramm ein Element hinzufügen zu können, muss sich der State Chart Editor in einem der „Platzierungsmodi“ befinden. Dieser Modus wird, im Slot der jeweiligen *QAction* des Menü *New,* in der Editor Klasse gesetzt.

Beispielsweise beim Drücken von *Add State*:

```
void MainWindow::onSwitchToPlacementStateMode()
{
    setToolButtonsUnchecked();
    _addStateAction->setChecked(true);
    emit changeEditMode(Editor::PlacementState);
}
```

Das Hinzufügen eines Elements zur Zustandsmaschine erfolgt immer in drei Etappen. Beim Drücken der linken Maustaste im Diagramm wird das graphische Element erstellt und als Vorschau angezeigt. Durch das Bewegen der Maus kann das Element platziert werden und beim Loslassen der Maustaste wird das Element am Raster ausgerichtet, der dazugehörige logische Teil erstellt und beiden Modellen hinzugefügt.

Linke Maustaste wird gedrückt

Befindet sich der Editor in einem der „Platzierungsmodi“, so wird das entsprechende graphische Element erstellt, als Vorschau in der Editor Klasse abgelegt und die Position der Vorschau der Mausposition gleichgesetzt. Das graphische Element kann dem Diagramm mit `addItem()` hinzugefügt werden.

```
void Diagram::mousePressAddItem( QGraphicsSceneMouseEvent * mouseEvent )
{
    if (mouseEvent->button() != Qt::LeftButton)
        return;

    Editor* editor = Editor::instance();
    switch( editor->mode() )
    {
        case Editor::PlacementInitialState:
            QGraphicsScene::mousePressEvent( mouseEvent );
            editor->setItemPreview(new InitialState());
            editor->itemPreview()->setPos( newX, newY );
            editor->itemPreview()->setHigherLayer();
            addItem(editor->itemPreview());
            mouseEvent->accept();
            break;

    ...
```

Maus wird bewegt

Beim Bewegen der Maus wird die erstellte Vorschau zusammen mit der Maus verschoben.

```
void Diagram::mouseMoveAddItem( QGraphicsSceneMouseEvent * mouseEvent )
{
    Editor* editor = Editor::instance();
    if( editor->itemPreview() )
    {
        editor->itemPreview()->setPos(mouseEvent->scenePos());
    }
    mouseEvent->accept();
}
```

Maustaste wird losgelassen

Wird die Maustaste losgelassen, muss das Element am Raster ausgerichtet und auf dem Diagramm platziert werden. Dabei muss überprüft werden, ob sich das Element über einem Zustand befindet. Falls sich das Element (vollständig) über einem Zustand befindet, wird der Zustand als *Parent Item* des Elements gesetzt und das Element wird der entsprechenden Region des Zustands hinzugefügt.

Nach dem Platzieren wird der dazugehörige logische Teil des graphischen Elements erstellt und mit dem graphischen Element verlinkt. Anschliessend werden die beiden Teile den entsprechenden Modellen hinzugefügt.

```
void Diagram::mouseReleaseAddItem( QGraphicsSceneMouseEvent * mouseEvent )
{
    qreal newX, newY;

    // place on grid
    newX = (qRound(mouseEvent->scenePos().x()/20))*20;
    newY = (qRound(mouseEvent->scenePos().y()/20))*20;

    ...
```

```
            // if a preview exists, place item
            if( editor->itemPreview() )
            {
                emit diagramModified(true);

                // create item
                InitialState* initialState
                                = dynamic_cast<InitialState*>(editor->itemPreview());
                initialState->setPos(newX, newY);

                ...

                State* parentState = stateUnderItem(initialState);
                Region* parentRegion = NULL;
                if( parentState )
                {
                    parentRegion = parentState->getRegionUnderItem(initialState);
                }

                // if a parent exists, update position to parent's coordinates
                if( parentState && parentRegion )
                {
                    QPointF oldPoint = initialState->scenePos();
                    parentRegion->addItem(initialState);
                    initialState->setParents(parentState, parentRegion);
                    initialState->setPos( parentState->mapFromScene(oldPoint) );
                }

                // set item in data model
                DataModel* dataModel = DataModel::instance();
                InitialStateItem* is = new InitialStateItem(initialState->uuid());
                initialState->setDataItem(is);

                if( parentState && parentRegion )
                    dataModel->addItem(parentRegion->dataItem(), is);
                else
                    dataModel->addItem(this->dataItem(), is);

                // delete preview
                editor->setItemPreview(NULL);
            }

    ...
```

6.3.3 Transitionen hinzufügen

Um eine Transition hinzufügen zu können, muss sich der Editor im entsprechenden Modus befinden.

Gleich wie das Hinzufügen eines Elements, erfolgt das Hinzufügen einer Transition in drei Etappen. Wird die linke Maustaste innerhalb eines Elements gedrückt, wird überprüft, ob die Transition von diesem Element aus starten darf. Handelt es sich um ein gültiges Element, so wird die Position der Maus (Startpunkt) im Editor abgelegt. Beim Verschieben der Maus wird eine Vorschau der Transition als Linie angezeigt (Line ausgehend vom Startpunkt zum Mauszeiger). Wird die Maustaste innerhalb eines Elements losgelassen, so wird erneut überprüft, ob die Transition in diesem Element enden darf. Wenn die Transition in diesem Element enden darf, werden die Ankerpunkte der Transition berechnet, die Transition und der dazugehörige logische Teil erstellt und in den Modellen abgelegt. Falls die Transition nicht in diesem Element enden darf, so wird die Vorschau verworfen und keine Transition erstellt.

Linke Maustaste wird gedrückt

Beim Drücken der linken Maustaste werden das Source-Element und der Startpunkt gesetzt und im Editor abgelegt.

Anschliessend wird überprüft, ob das Source-Element existiert und ob es sich um ein gültiges Element handelt. Falls die Überprüfung erfolgreich war, wird die Vorschau der Transition erstellt und ebenfalls im Editor abgelegt.

```
void Diagram::mousePressAddConnection( QGraphicsSceneMouseEvent * mouseEvent )
{
    if (mouseEvent->button() != Qt::LeftButton)
        return;

    Editor* editor = Editor::instance();
    QGraphicsScene::mousePressEvent( mouseEvent );
    editor->setFrom(dynamic_cast<DiagramItem*>(itemAt(mouseEvent->scenePos())));
    editor->setConnectionStart(new QPointF(mouseEvent->scenePos()));

    // check if source exists and type is ok
    // if type not ok, clear preview
    if( editor->from() && editor->from()->type() != DiagramItem::StateType &&
        editor->from()->type() != DiagramItem::ConditionStateType &&
        editor->from()->type() != DiagramItem::InitialStateType )
    {
        editor->setFrom(NULL);
        editor->setConnectionStart(NULL);
    }

    // if all ok, set preview
    else if( editor->from() &&
             ( editor->from()->type() != DiagramItem::StateType ||
               editor->from()->type() != DiagramItem::ConditionStateType ||
               editor->from()->type() != DiagramItem::InitialStateType ) )
    {
        editor->setConnectionPreview(addLine(QLineF(*editor->connectionStart(),
                                                    mouseEvent->scenePos())));
        editor->connectionPreview()->setZValue(Diagram::HighestLayer);
    }
    mouseEvent->accept();
}
```

Maus wird bewegt

Beim Bewegen der Maus wird die Vorschau der Transition aktualisiert.

```
void Diagram:: mouseMoveAddConnection( QGraphicsSceneMouseEvent * mouseEvent )
{
    Editor* editor = Editor::instance();
    if( editor->from() && editor->connectionPreview() )
    {
        editor->connectionPreview()->setLine(QLineF(*editor->connectionStart(),
                                                    mouseEvent->scenePos()));
    }
    mouseEvent->accept();
}
```

Maustaste wird losgelassen

Beim Loslassen der Maustaste wird überprüft, ob das oberste graphische Element als Target verwendet werden darf und ob Source- und Target-Element dasselbe *Parent Item* besitzen.

Falls die Überprüfung erfolgreich war, können die Ankerpunkte berechnet und die Transition erstellt werden. Besitzen beide Elemente dasselbe *Parent Item*, wird die Transition demselben *Parent Item* hinzugefügt, ansonsten dem Diagramm. Gleich wie bei

den Elementen wird der logische Teil erstellt, mit dem graphischen Element verlinkt und anschliessend dem Modell hinzugefügt.

Am Ende wird die Vorschau der Transition verworfen.

```
void Diagram::mouseReleaseAddConnection( QGraphicsSceneMouseEvent * mouseEvent )
{
    Editor* editor = Editor::instance();
    DiagramItem* to = NULL;

    ...

    // if source and destination are available and destination type is ok,
    // then create connection
    if( editor->from() &&
        to &&
        to->parentRegion() == editor->from()->parentRegion() &&
        to != editor->from() &&
        ( to->type() == DiagramItem::StateType ||
          to->type() == DiagramItem::ConditionStateType ||
          to->type() == DiagramItem::TerminateStateType ) )
    {
        ...
        // if destination has no parent, add connection to scene
        if( to->parentRegion() == NULL )
        {
            Transition* transition = new Transition(editor->from(),
                                                    *editor->connectionStart(),
                                                    to, mouseEvent->scenePos());
            ...
            addItem(transition);

            // add transition to data model
            TransitionItem* t = new TransitionItem(transition->uuid());
            transition->setDataItem(t);
            t->setSourceTarget(editor->from()->dataItem(), to->dataItem());
            dataModel->addItem(this->dataItem(), t);
            ...
        }
        // if destination has parent, add connection to the parent
        else
        {
            State* parentState = dynamic_cast<State*>(to->parentItem());
            Region* parentRegion = NULL;
            if( parentState )
            {
                parentRegion = to->parentRegion();
            }
            if( parentState && parentRegion )
            {
                Transition* transition
                          = new Transition(editor->from(),
                                parentState->mapFromScene(*editor->connectionStart()),
                                to, parentState->mapFromScene(mouseEvent->scenePos()));
                ...
                parentRegion->addConnectionItem(transition);
                transition->setParents(parentState, parentRegion);

                // add transition to data model
                TransitionItem* t = new TransitionItem(transition->uuid());
                transition->setDataItem(t);
                t->setSourceTarget(editor->from()->dataItem(), to->dataItem());
                dataModel->addItem(parentRegion->dataItem(), t);
                ...
            }
        }
    }
    ...
    mouseEvent->accept();
}
```

Berechnung der Ankerpunkte

`getConnectionPointAngle()` berechnet aus dem Punkt, wo die linke Maustaste gedrückt bzw. losgelassen wurde, die Ankerpunkte der Transition.

Bei einem rechteckigen Element wird der Punkt, wo die Maustaste geklickt/losgelassen wurde, an den Elementrand verschoben und zurückgegeben.

Beim kreisförmigen Element wird der Winkel zwischen der x-Achse und der Verlängerung vom Mittelpunkt und dem Punkt, wo die Maustaste geklickt/losgelassen wurde, zurückgegeben. Anschliessend kann der Ankerpunkt mit Hilfe des Mittelpunkts, des Radius und des Winkels berechnet werden.

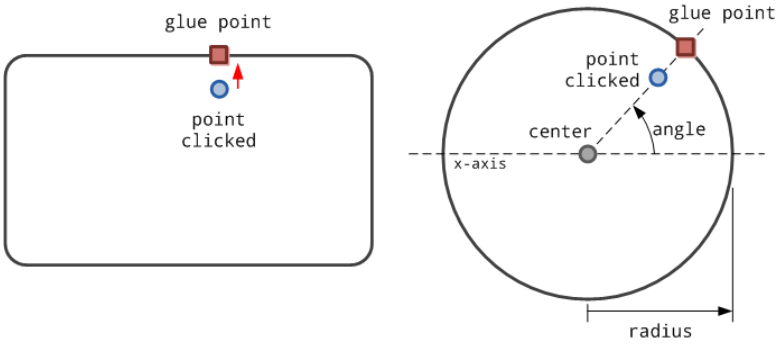


Abbildung 40: Rückgabe von `getConnectionPointAngle()`

6.3.4 Elemente/Transitionen entfernen

Zum Entfernen von Elementen oder Transition wurde der *Key-Event-Handler* in der Diagram Klasse implementiert.

Wird die *Delete* Taste gedrückt, so wird eine Liste der zu löschenden Elemente bzw. Transitionen erstellt (es handelt sich um die markierten Elemente). Anschliessend wird die Liste durchlaufen und Element für Element aus den Modellen entfernt und gelöscht. Dabei wird die Liste der zu löschenden Element aktualisiert, um zu verhindern, das ein Element gelöscht werden will, das schon zusammen mit seinem *Parent Item* gelöscht wurde.

Die Implementierung des Handlers ist in der Diagram Klasse zu finden, diese Klasse befindet sich elektronisch unter

- ../Software/SourceCode/src/Diagram
- ../Software/SourceCode/include/Diagram

6.3.5 Elemente verschieben

Das Verschieben von Elementen erfolgt in drei Etappen. Zuerst werden die zu verschiebenden Elemente markiert und in den Vordergrund gebracht. Mit gedrückter linker Maustaste können die Elemente verschoben werden. Wird die Maustaste losgelassen, werden die Elemente an der neuen Stelle, am Raster ausgerichtet, positioniert.

Linke Maustaste wird geklickt

Beim Klicken der Maustaste werden alle selektionierten Elemente / Transition in den Vordergrund gebracht.

```
foreach(QGraphicsItem* selectedItem, selectedItems())
{
    DiagramItem* item = dynamic_cast<DiagramItem*>(selectedItem);
    ConnectionItem* connection = dynamic_cast<ConnectionItem*>(selectedItem);
    // set selected items on highest layer
    if( item )
        item->setHigherLayer();
    if( connection )
        connection->setHigherLayer();
}
```

Maus wird bewegt

Das Verschieben von *QGraphicsItems* wird vom *Default-Mouse-Event-Handler* der *QGraphicsScene* bereits unterstützt. Wird die Mausbewegt, wird der *Default-Handler* aufgerufen und die Transitionen der verschobenen Elemente neu berechnet.

```
void Diagram::mouseMoveSelection( QGraphicsSceneMouseEvent * mouseEvent )
{
    QGraphicsScene::mouseMoveEvent(mouseEvent);
    foreach(QGraphicsItem* item, selectedItems()) {
        // update all connections involved while moving
        DiagramItem* diagramItem = dynamic_cast<DiagramItem*>(item);
        if ( diagramItem ) {
            foreach(ConnectionItem *c, diagramItem->connections()) {
                c->setGeometryChanged(true);
            }
        }
    }
    mouseEvent->accept();
}
```

Maustaste wird losgelassen

Beim Loslassen der Maustaste muss für jedes Element überprüft werden, ob sein *Parent Item* geändert hat. Falls das *Parent Item* geändert hat, muss das Element dem neuen *Parent Item* hinzugefügt und vorhandene Transitionen entfernt werden. Dieser Vorgang ist ähnlich jenem, der beim Hinzufügen von Elementen erklärt wurde.

Die vollständige Implementierung des `mouesReleaseEvent()` der Diagram Klasse ist elektronisch unter

- ../Software/SourceCode/src/Diagram

zu finden.

6.3.6 Die Grösse eines Zustands ändern

Zum Ändern der Grösse des Zustands werden die Selection-Corner verwendet.

Beim Erstellen der Selection-Corner in der DiagramItem Klasse wird auf den Selection-Corners ein *SceneEventFilter* installiert und in der DiagramItem Klasse der `sceneEventFilter()` implementiert.

```
_corners[0] = new SelectionCorner(this, SelectionCorner::TopLeft);
_corners[0]->installSceneEventFilter(this);
...
```

Der `sceneEventFilter()` wird dazu verwendet, um Events eines anderen *QGraphicsItem* in einer Klasse zu verarbeiten. Zum Ändern der Elementgrösse wird `sceneEventFilter()` wie folgt implementiert.

```
bool DiagramItem::sceneEventFilter (QGraphicsItem * watched, QEvent * event)
{
    SelectionCorner* corner = dynamic_cast<SelectionCorner*>(watched);
    if ( corner == NULL) return false;

    QGraphicsSceneMouseEvent* mouseEvent =
                                        dynamic_cast<QGraphicsSceneMouseEvent*>(event);

    if ( mouseEvent == NULL) return false;
    switch (event->type())
    {
        case QEvent::GraphicsSceneMousePress:
            corner->setMouseState(SelectionCorner::Pressed);
            corner->mouseDownX(mouseEvent->pos().x());
            corner->mouseDownY(mouseEvent->pos().y());
            break;
        case QEvent::GraphicsSceneMouseRelease:
            corner->setMouseState(SelectionCorner::Released);
            break;
        case QEvent::GraphicsSceneMouseMove:
            corner->setMouseState(SelectionCorner::Moving );
            break;
        default:
            return false;
            break;
    }

    if ( corner->mouseState() == SelectionCorner::Moving )
    {
        resizeItem(corner, mouseEvent->pos().x(), mouseEvent->pos().y());
    }

    return true;
}
```

Beim Aufruf von `sceneEventFilter()` wird zuerst überprüft, ob das Event von einem Selection-Corner stammt.

Stammt das Event von einem Selection-Corner, so wird die Mausposition beim Drücken der linken Maustaste im Corner abgespeichert, damit während dem Bewegen der Maus die Änderung der Grösse berechnet werden kann.

Beim Bewegen der Maus wird `resizeItem()` mit Angabe des Corners und der aktuellen Mausposition aufgerufen. Da beim Drücken der Maustaste die Position im Corner gespeichert wurde und die aktuelle Mausposition bekannt ist, kann diese Methode die Verschiebung berechnen, anhand der Corner-Position das Vorzeichen der Grössenänderung bestimmen und somit die Grösse des Elements ändern.

Die vollständige Implementierung der `resizeItem()` Methode ist in der State Klasse elektronisch unter

- ../Software/SourceCode/src/Items

zu finden.

Die Grössenänderung wurde auf der Basis des Beispiels in der Referenz [5] realisiert.

6.3.7 Einem Zustand Regionen hinzufügen / entfernen

Um dem Zustand eine Region hinzuzufügen oder zu entfernen, muss sich der State Chart Editor im entsprechenden Modus befinden.

Hinzufügen (`mousePressAddRegion`)

Zum Hinzufügen einer Region wird überprüft, ob die Maus innerhalb eines Zustands gedrückt wurde. Falls die Maustaste innerhalb eines Zustands gedrückt wurde, wird der Zustand verbreitert, eine Region und ihr dazugehöriger logischer Teil erstellt, die Region dem Zustand hinzugefügt und das graphische und logische Element den Modellen hinzugefügt.

Entfernen (`mousePressRemoveRegion`)

Zum Entfernen einer Region wird überprüft, ob die Region, in welche geklickt wurde, leer ist und ob diese Region nicht die einzige im Zustand ist. Falls die Überprüfung erfolgreich war, wird die Region aus den Modellen entfernt und gelöscht. Die Breite wird der entfernten Region wird an die Nachbarregionen verteilt. War die Überprüfung nicht erfolgreich, so wird eine Fehlermeldung ausgegeben.

Die Implementierung der beiden Methoden, `mousePressAddRegion()` und `mousePressRemoveRegion()`, ist elektronisch in der Diagram Klasse unter

- ../Software/SourceCode/src/Diagram

zu finden.

6.3.8 Property-Editor

Wird ein Element im Diagramm, der Projekt-Ansicht oder der State-Chart-Ansicht ausgewählt, so wird dessen Properties-Widget im Property-Editor angezeigt.

Mit `editDataItem()` des Property-Editors kann das ausgewählte Element angegeben werden, um dessen Properties-Widget anzuzeigen.

Jedes Element verfügt über eine Methode `propertiesWidget()`, die das zum Element zugehörige Properties-Widget zurückgibt.

Bei jedem Element lässt sich mindestens der Name ändern. Um zu verhindern, dass ein Name eines Elements zweimal existiert, wurde die sogenannte NameChecker Klasse entwickelt, welche beim Ändern des Namens überprüfen kann, ob der Name bereits existiert.

NameChecker

Die NameChecker Klasse beinhaltet eine Map, die sämtliche Elementnamen enthält. So kann überprüft werden, ob ein Name bereits existiert oder nicht.

Mit `setItemName()` kann der Name eines Elements geändert werden. Ist der Name bereits vorhanden, wird er nicht geändert und die Methode retourniert `false`.

Zudem besitzt die Klasse die Methode `setDefaultName()`, welche verwendet wird, wenn ein neues Element erstellt wird. Der Elementname wird wie folgt konstruiert: `<Elementtyp_#>`.

Die NameChecker Klasse befindet sich elektronisch unter:

- ../Software/SourceCode/src/DataModel
- ../Software/SourceCode/include/DataModel

Code-Editor

In den Zuständen oder auf Transitionen kann das Verhalten implementiert werden. Zur Implementierung des Verhaltens wird ein kleiner Code-Editor geöffnet. Dieser Code-Editor wurde auf der Basis des „Code Editor Example“ der Qt Reference Documentation (siehe [3]) entwickelt.

Der Source-Code des Code-Editors ist elektronisch unter

- ../Software/SourceCode/src/UI
- ../Software/SourceCode/include/UI

zu finden

6.4 Weitere Funktionen

Im letzten Abschnitt wird noch kurz auf die Implementierung des Speicher- und Ladevorgangs, den Export, das Kopieren in die Zwischenablage und das Drucken eingegangen.

6.4.1 Speichern / Laden

In den Kapiteln der Modelle (§ 3 und § 4) wurde bereits beschrieben, wie ein Modell in ein IODevice geschrieben und wieder von einem IODevice geladen werden kann. Beim Abspeichern eines Projekts des State Chart Editors ist es jedoch von Vorteil, nur in eine Datei abzuspeichern. So wird garantiert, dass die graphischen und logischen Informationen immer zusammenbleiben.

Es wird nun gezeigt, wie die beiden Teile bei einem Speichervorgang in eine Archiv-Datei verpackt und wie die Archiv-Datei bei einem Ladevorgang entpackt werden kann.

Um Archiv-Dateien zu erstellen, wurde die *ZLIB* [8] zusammen mit *QuaZip* [7] verwendet.

Bemerkung: Auf der Windows-Plattform ist die *ZLIB* standardmässig nicht installiert und muss daher zuerst installiert werden.

Speichern

Beim Speichervorgang wird zunächst eine Archiv-Datei (*QuaZip*) erstellt.

```
// create new zip archive
QFile zipFile(fileName);
QuaZip zip(&zipFile);
if( !zip.open(QuaZip::mdCreate) )
{
    QApplication::restoreOverrideCursor();
    QMessageBox::critical(0,
                          tr("Error while saving..."),
                          tr("An error occured while creating file. Error-Code: %1")
                          .arg(zip.getZipError()));
    return false;
}
```

Anschliessend wird für die logischen und graphischen Informationen ein IODevice (*QuaZipFile*) erstellt, welche in der Archiv-Datei verpackt werden. Wie in den Abschnitten § 3.4.1 und § 4.4.1 beschrieben, werden die Daten in das IODevice geschrieben.

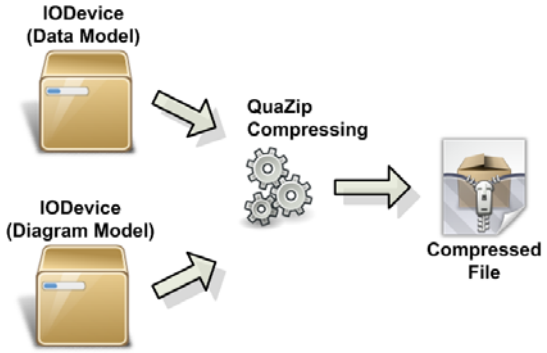


Abbildung 41: Speichern in eine Archiv-Datei

Hier das Beispiel für die graphischen Informationen.

```
// create graphical archive file (file in the archive)
QuaZipFile fileGraphic(&zip);
if( !fileGraphic.open(QIODevice::WriteOnly, QuaZipNewInfo("graphical.xml")) )
{
    QApplication::restoreOverrideCursor();
    QMessageBox::critical(0,
                          tr("Error while saving..."),
                          tr("An error occured while saving graphical data. Code: %1")
                          .arg(fileGraphic.getZipError()));
    return false;
}

QXmlStreamWriter writerG( &fileGraphic );
// write model to file
writerG << _diagramModel;
fileGraphic.close();
```

Laden

Beim Laden wird die Archiv-Datei geöffnet und entpackt. Die Dateien in der Archiv-Datei werden durchlaufen und mit den beschriebenen (§ 3.4.2 und § 4.4.2) DOM-Parsern gelesen.

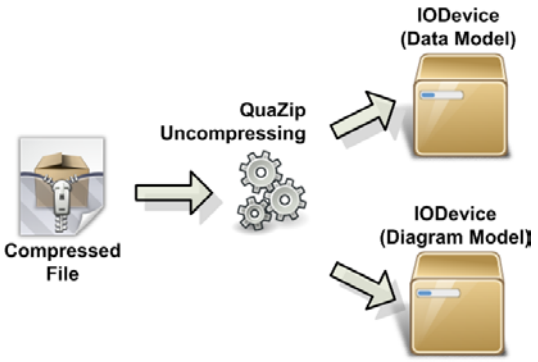


Abbildung 42: Laden aus einer Archiv-Datei

Der untenstehende Programmcode zeigt das Öffnen der Archiv-Datei.

```
// create zip archive
QFile zipFile(fileName);
QuaZip zip(&zipFile);
// open zip archive
if( !zip.open(QuaZip::mdUnzip) )
{
    QApplication::restoreOverrideCursor();
    QMessageBox::critical(0,
                          tr("Error while loading..."),
                          tr("An error occured while loading file. Error-Code: %1")
                          .arg(zip.getZipError()));
    return false;
}
```

Die Dateien in der Archiv-Datei können wie folgt durchlaufen werden:

```
// create new archive file (file in the archive)
QuaZipFile file(&zip);
DiagramDomParser diagramParser(_diagramModel);
DataDomParser dataParser(_dataModel);

// get files in the zip archive
for( bool more = zip.goToFirstFile(); more; more = zip.goToNextFile() )
{
    ...
```

Folgend das Beipsiel zum Parsen der graphischen Informationen.

```
    if( !file.open(QIODevice::ReadOnly) )
    {
        QApplication::restoreOverrideCursor();
        QMessageBox::critical(0,
                              tr("Error while loading..."),
                              tr("An error occured while opening file. Error-Code: %1")
                              .arg(file.getZipError()));
        return false;
    }

    // read graphical data
    if( file.getActualFileName() == "graphical.xml" )
    {
        diagramParser.readFile(&file);
        ...
        file.close();
    }
    ...
```

6.4.2 Exportieren

Zum Exportieren der logischen Informationen wird mittels des beschriebenen *StreamWriters* (siehe § 3.4.1) eine XML-Datei erstellt.

```
QFile exportFile(fileName);
exportFile.open(QIODevice::WriteOnly);
QXmlStreamWriter writer(&exportFile);
writer << _dataModel;
exportFile.close();
```

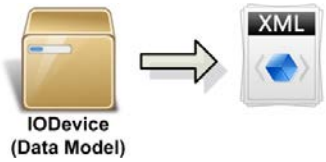


Abbildung 43: Exportieren der logischen Informationen

6.4.3 In Zwischenablage kopieren

Um die angezeigte Zustandsmaschine als Bild in die Zwischenablage zu kopieren, wird zuerst die Zustandsmaschine als Bild gerendert. Das gerenderte Bild wird anschliessend in die Zwischenablage gelegt.

Mittels `QApplication::clipboard()` kann auf die Zwischenablage des Systems zugegriffen und mit `setImage()` kann das Bild in die Zwischenablage gelegt werden.

```
void MainWindow::onCopyToClipboard()
{
    Diagram* currentScene = dynamic_cast<Diagram*>(_view->scene());
    // if a diagram exists, render scene to image and set to clipboard
    if( currentScene )
    {
        currentScene->clearSelection();
        currentScene->shrinkSceneRectToContent();
        currentScene->showGrid(false);

        qreal imageWidth = currentScene->sceneRect().size().width();
        qreal imageHeight = currentScene->sceneRect().size().height();
        QImage image(imageWidth + 10, imageHeight + 10, QImage::Format_ARGB32);
        image.fill(QColor(Qt::white).rgba());

        QPainter painter(&image);
        painter.setRenderHint(QPainter::Antialiasing);
        currentScene->render(&painter,
                             QRectF(5, 5, imageWidth, imageHeight),
                             currentScene->sceneRect());

        currentScene->showGrid(true);
        currentScene->restoreSceneRect();
        painter.end();

        QApplication::clipboard()->setImage(image);
    }
}
```

Bemerkung: Damit das richtige Rechteck der Szene zur Renderung gewählt wird, wird vor dem Rendern `shrinkSceneRectToContent()` aufgerufen. Nach dem Rendern muss das Rechteck mit `restoreSceneRect()` wiederhergestellt werden. (siehe S. 33)

6.4.4 Drucken

Das Drucken funktioniert ähnlich wie das Kopieren in die Zwischenablage. Es wird ein *QPrinter* erstellt und die angezeigte Szene in diesen Printer gerendert.

```
void MainWindow::print()
{
    QPrinter printer;
    printer.setOrientation(QPrinter::Landscape);
    printer.setPageMargins(20, 20, 20, 20, QPrinter::Millimeter);

    Diagram* currentScene = dynamic_cast<Diagram*>(_view->scene());
    if( currentScene )
    {
        QPrintDialog printDialog(&printer, this);
        if( printDialog.exec() )
        {
            QPainter painter(&printer);

            currentScene->clearSelection();
            currentScene->shrinkSceneRectToContent(painter.window().size());
            currentScene->showGrid(false);
```

```
            /* WORKAROUND */
            painter.fillRect(painter.window(), QColor(Qt::transparent));
            /* WORKAROUND */

            painter.setRenderHint(QPainter::Antialiasing);
            currentScene->render(&painter);

            currentScene->showGrid(true);
            currentScene->restoreSceneRect();

            painter.end();
        }
    }
    else
    {
        QMessageBox::warning( 0,
                              tr("Printing Error"),
                              tr("No state chart shown to print...") );
    }
}
```

Bemerkung: Die aktuelle Version 4.7.4 der Qt-Library beinhaltet einen Fehler beim Rendern in einen Printer. Daher musste zuerst ein transparentes Rechteck im Printer gezeichnet werden.

7 Tests

Am Ende der Realisierung wurde die Funktionalität des State Chart Editors überprüft. Alle durchgeführten Tests lieferten ein positives Resultat.

7.1 Die Bedienung

Um die Bedienung des Editors zu testen, wurden sämtliche möglichen Aktionen durchgeführt.

Es wurde mit dem Erstellen eines neuen Projekts begonnen. Dann wurden dem Projekt Diagramme (Zustandsmaschinen) hinzugefügt und wieder gelöscht. Auf den erstellten Diagrammen wurden mit den unterschiedlichen Elementen Zustandsmaschinen gezeichnet. So konnte das Hinzufügen von Elementen und Transitionen kontrolliert werden. Den Zuständen wurden mehrere Regionen hinzugefügt und entfernt. Dabei wurde überprüft, dass Regionen, nur wenn sie leer sind, gelöscht werden können.

Zudem wurde das Verschieben von Elementen und Transitionen getestet, um zu verifizieren, dass die Elemente und Transition immer dem richtigen *Parent Item* zugeordnet werden.

Am Ende wurde noch das Löschen von Elementen und Transition überprüft. Es wurde darauf geachtet, dass sämtliche Informationen korrekt entfernt werden.

7.2 Property-Editor

Beim Selektieren von Zustandsmaschinen, Elementen und Transitionen wurde kontrolliert, dass jeweils der richtige Property-Editor angezeigt wird.

Falls Änderungen im Property-Editor vorgenommen wurden, wurde nachgeprüft, ob die Änderungen übernommen worden sind.

Ebenfalls die Funktionalität des NameCheckers wurde verifiziert. Es wurde überprüft, dass ein Name nur einmal im Projekt existieren darf.

7.3 Speichern / Laden

Zum Testen des Speicher- und Ladevorgangs wurde eine Zustandsmaschine erstellt und abgespeichert. Der Inhalt der abgespeicherten Datei wurde mit der gezeichneten Zustandsmaschine verglichen. Anschliessend wurde die gespeicherte Zustandsmaschine wieder geladen und es wurde kontrolliert, ob sie korrekt und vollständig geladen wurde.

7.4 Exportieren

Um den Export der logischen Informationen zu testen, wurde eine Zustandsmaschine exportiert. Darauf wurden die Informationen der exportierten XML-Datei mit den Informationen der gezeichneten Zustandsmaschine verglichen.

7.5 In Zwischenablage kopieren / Drucken

Das Kopieren in die Zwischenablage und das Drucken wurde geprüft, indem eine Zustandsmaschine kopiert/gedruckt wurde und das Resultat mit der am Bildschirm angezeigten Zustandsmaschine verglichen wurde.

8 Schlussfolgerung

8.1 Resultat

Die Diplomarbeit konnte mit Erfolg abgeschlossen werden. Sämtliche Ziele wurden erreicht.

Es wurde ein funktionsfähiges modellbasiertes Entwicklungstool realisiert. Das Tool erlaubt es Zustandswechseldiagramme zu zeichnen. Selbst komplexere Zustandsmaschinen, welche Zustände mit mehreren Regions enthalten, lassen sich erstellen. Das Verhalten der Zustände (Action On Entry, …) und der Transitionen (Trigger, …) kann direkt mit Hilfe eines Property-Editors implementiert werden.

Es besteht die Möglichkeit die Diagramme abzuspeichern und bestehende Diagramme zu öffnen. Das Entwicklungstool besitzt eine Exportfunktion, mit welcher sich die logischen Informationen einer Zustandsmaschine zur weiteren Verwendung exportieren lassen.

Zusätzlich bietet das Tool die Funktionalität des „Kopierens in die Zwischenablage“ an. Die Kopie enthält das Zustandswechseldiagramm als Bild und kann in einer anderen Anwendung (Word und ähnliche) beispielsweise zur Dokumentation verwendet werden. Zudem können die Zustandsmaschinen ausgedruckt werden.

8.2 Realisierung

Die Realisierung des Entwicklungstools (State Chart Editor) erfolgte in verschiedenen Etappen.

Datenmodelle

Zu Beginn wurde eine Analyse zur Ablage der Zustandsmaschinendaten durchgeführt. Da die Daten im Tool hierarchisch angezeigt werden, eignete sich das Model-View-Konzept von Qt am besten.

Die Modelle (logisch und graphische Daten) wurden auf diesem Konzept aufgebaut und sind somit kompatibel mit den Qt-Views, um in diesen angezeigt werden zu können. Um die Modelle realisieren zu können, wurde zuerst die Datenstruktur der Modelle definiert.

In einem nächsten Schritt wurde ein Interface implementiert, welches es erlaubt, auf die Datenstruktur zuzugreifen, damit diese in Views angezeigt bzw. bearbeitet werden kann.

Um die Modelle abspeichern zu können, wurde ein Stream-Writer von Qt genutzt und zum Laden wurde ein DOM-Parser erstellt.

Graphische Elemente

In einer nächsten Etappe wurde die graphische Benutzeroberfläche des State Chart Editors entwickelt.

Um die Benutzeroberfläche möglichst einfach zu erstellen, boten sich die Qt-Libraries an. Mit Hilfe dieser Libraries wurde das Hauptfenster mit den verschiedenen Bereichen erstellt.

Zur Anzeige der hierarchisch gegliederten Daten konnten direkt Qt-Views verwendet werden, da die Modelle auf dem Model-View-Konzept von Qt aufgebaut sind. Ebenfalls die Menüleiste und Toolbar liess sich einfach implementieren.

Die grössten Schwierigkeiten bereitete das Zeichnen der Zustandsmaschinen. Es musste zuerst nach einer geeigneten Lösung gesucht werden. Wie beim Erstellen der Benutzeroberfläche, konnte der passende Ansatz in den Bibliotheken von Qt gefunden werden. Mit Hilfe des „Graphics Framework“ konnte das Zeichnen schliesslich erfolgreich realisiert werden.

Implementierung

Mit Hilfe der Entwicklung der Datenmodelle und der graphischen Elemente konnte schliesslich der State Chart Editor implementiert werden. Nach der Realisierungsphase wurde der State Chart Editor auf seine Funktionalität überprüft.

Als Resultat kann ein funktionsfähiges modellbasiertes Entwicklungstool geliefert werden.

8.3 Vorschläge zur Weiterentwicklung

Da es sich beim realisierten Entwicklungstool um eine Erstversion handelt, gibt es einige Punkte, die in einer zukünftigen Version optimiert oder hinzugefügt werden könnten.

Optimierung des Ankerpunkts
Der Algorithmus, der zur Berechnung der Ankerpunkte beim Zeichnen von Transitionen implementiert wurde, ist sehr einfach aufgebaut und könnte in einer nächsten Version ausgearbeitet werden.

Zoom
Werden komplexe und grosse Zustandsmaschinen gezeichnet, reicht die Arbeitsfläche nicht mehr aus, um diese anzuzeigen. Daher ist es sinnvoll eine Zoom-Funktion hinzuzufügen.

Copy-Paste
Beim Zeichnen von Zustandswechseldiagrammen kann es vorkommen, dass bestimmte Elemente mehrfach verwendet werden. Eine Copy-Paste-Funktion würde somit das Zeichnen solcher Zustandsmaschinen vereinfachen.

Löschen der Elemente via Projekt-Ansicht
In der aktuellen Version ist das Löschen von Elementen nur auf der Arbeitsfläche möglich. Handelt es sich um eine grössere Zustandsmaschine, muss das zu löschende Element im Diagramm gesucht werden. In der Projekt-Ansicht könnten die Elemente schneller gefunden werden, jedoch wird das Löschen hier nicht unterstützt.

Einbinden eines Code-Generators
Der State Chart Editor dient vorläufig nur zum Zeichnen der Zustandswechsel-diagramme. Die logischen Informationen müssen zur Weiterverarbeitung exportiert werden. In einer nächsten Version könnte beispielsweise der Code-Generator direkt eingebunden werden.

Drucken des implementierten Programmcodes
Das Drucken des implementieren Programmcodes könnte erwünscht sein. Momentan wird jedoch nur das Drucken der Zustandsmaschinen angeboten.

9 Unterschrift

Sion, 11. Juli 2011

Ralph Martig

10 Verzeichnisse

10.1 Abbildungsverzeichnis

10.2 Diagrammverzeichnis

11 Referenzen

[1] Jürgen Wolf,
Qt 4.6 GUI-Entwicklung mit C++,
2. Auflage, Galileo Press, 2010

[2] OMG Unified Modeling Language (OMG UML),
Superstructure, V2.1.2

[3] Qt Reference Documentation,
Qt Assistant, Version 4.7.4,
Nokia Corporation, 2010

[4] Jasmin Blanchette, Mark Summerfield,
C++ GUI Programming with Qt 4,
Prentice Hall Professional, 2006

[5] *Exmaple of re-sizing a QGraphicsItem using Mouse events*,
http://davidwdrell.net/graphicslesson3.htm,
18. Mai 2011

[6] *Drawing Gird on QGraphicsScene*,
http://stackoverflow.com/questions/2779142/artifacts-when-trying-to-draw-background-grid-without-anti-aliasing-in-a-qgraphic,
19. Mai 2011

[7] *QuaZIP - Qt/C++ wrapper for ZIP/UNZIP package,*
http://quazip.sourceforge.net/,
10. Juni 2011

[8] *A Massively Spiffy Yet Delicately Unobtrusive Compression Library*
http://zlib.net/,
10. Juni 2011

12 Anhang

12.1 Speicherformate

Die logischen und graphischen Informationen der Elemente werden im XML-Format gespeichert. Dieser Abschnitt beschreibt den Aufbau der beiden Dateien.

12.1.1 Logische Elemente

Die logischen Informationen werden in einer XML-Datei mit dem Root-Tag `<ProjectData>` abgespeichert.

Für die Zustandsmaschinen wird der Tag `<StateMachine>` mit den Attributen `type`, `name` und `uuid` erstellt.

Sämtliche Elemente, die in einer Zustandsmaschine bzw. Region enthalten sein können, sind wie folgt aufgebaut:

Zustände:

```
<State type="..." name="..." uuid="{########-####-####-####-############}">
    <ActionOnEntry>...</ActionOnEntry>
    <ActionInState>...</ActionInState>
    <ActionOnExit>...</ActionOnExit>
    <Regions>
        ...
    </Regions>
</State>
```

Regionen:

```
<Region type="..." name="..." uuid="{########-####-####-####-############}"/>
```

Pseudo-Zustände:

```
<PseudoState type="..." name="..." uuid="{########-####-####-####-############}"/>
```

Transitionen:

```
<Transition type="..." name="..." uuid="{########-####-####-####-############}">
    <Trigger>...</Trigger>
    <Guard>...</Guard>
    <Action>...</Action>
</Transition>
```

Beispiel:

Folgend wird ein Beispiel der logischen Informationen einer kleinen Zustandsmaschine gezeigt:

```
<?xml version="1.0" encoding="UTF-8"?>
<ProjectData>
    <StateMachine type="StateMachine" name="Clock"
                  uuid="{61af3933-5b91-47c5-b302-b2351f80700d}">
        <PseudoState type="InitialState" name="START"
                     uuid="{98057502-299c-44ff-8fa0-5af5b6021f00}"/>
        <State type="State" name="INIT" uuid="{82ecfb72-b44c-487f-81b7-e615b6ef7585}">
            <ActionOnEntry>seconds = 0;
                           minutes = 0;
                           hours   = 0;
            </ActionOnEntry>
            <ActionInState></ActionInState>
            <ActionOnExit></ActionOnExit>
```

```
            <Regions>
                <Region type="Region" name="Region_1"
                        uuid="{fb77dbbe-62c3-426e-98a4-0a4e643e4600}"/>
            </Regions>
        </State>
        <State type="State" name="WAIT" uuid="{1c4d3b43-7eb4-48ad-b5d6-b60958b7014c}">
            <ActionOnEntry></ActionOnEntry>
            <ActionInState></ActionInState>
            <ActionOnExit></ActionOnExit>
            <Regions>
                <Region type="Region" name="Region_2"
                        uuid="{6eb821da-0d9a-4f3a-a0f7-a788620d7d99}"/>
            </Regions>
        </State>
        <State type="State" name="TICK" uuid="{f396f156-32d1-45a2-a2f4-9249d74457e5}">
            <ActionOnEntry>seconds++;
                           if( seconds == 60 )
                           {
                               seconds = 0;
                               minutes++;
                           }
                           if( minutes == 60 )
                           {
                               minutes = 0;
                               hours++;
                           }
                           if( hours == 24 )
                           {
                               hours = 0;
                           }
            </ActionOnEntry>
            <ActionInState></ActionInState>
            <ActionOnExit></ActionOnExit>
            <Regions>
                <Region type="Region" name="Region_3"
                        uuid="{05efb9b0-ac6c-43ca-b327-baffd793ccf8}"/>
            </Regions>
        </State>
        <Transition type="Transition" name="fromStart"
                    uuid="{79bb9cd2-700a-4829-a225-6cd248f2244c}">
            <Trigger></Trigger>
            <Guard></Guard>
            <Action></Action>
        </Transition>
        <Transition type="Transition" name="fromInit"
                    uuid="{9f699367-d952-4bdf-ac8e-370131f94c04}">
            <Trigger></Trigger>
            <Guard></Guard>
            <Action></Action>
        </Transition>
        <Transition type="Transition" name="fromWait"
                    uuid="{0b2ecfa6-55c9-4379-87e7-2261c81826d2}">
            <Trigger>tm(1000)</Trigger>
            <Guard></Guard>
            <Action></Action>
        </Transition>
        <Transition type="Transition" name="fromTick"
                    uuid="{667719a3-a6c9-476c-8fda-f0055a6ccf0e}">
            <Trigger></Trigger>
            <Guard></Guard>
            <Action></Action>
        </Transition>
    </StateMachine>
</ProjectData>
```

12.1.2 Graphische Elemente

Die graphischen Informationen werden in einer XML-Datei mit dem Root-Tag `<DiagramData>` abgespeichert.

Jedes Diagramm (Zustandsmaschine) wird in einem Tag `<Diagram>` mit dem Attribut `uuid` abgelegt.

Die graphischen Elemente werden im Tag `<DiagramItem>` gespeichert. Das Tag besitzt das Attribut type, das den Elementtyp angibt, die Attribute `width`, `height`, `x` und `y`, welche die Dimension und Position des Elements angeben, sowie das Attribut `uuid` mit der UUID des Elements.

```
<DiagramItem type="..." width="..." height="..." x="..." y="..."
             uuid="{########-####-####-####-############}"/>
```

Zustände enthalten zudem den Tag `<Region>`, welcher ein Attribut `uuid` und `width` enthält.

```
<Region uuid="{########-####-####-####-############}" width="..."/>
```

Die Transition werden im Tag `<ConnectionItem>` gespeichert. Dieser Tag besitzt ebenfalls die Attribute `type` und `uuid`. Zuden enthält dieses Element Angaben zum Source- und Target-Element, sowie Angaben zur Position des Transition-Texts.

```
<ConnectionItem type="Transition" uuid="{########-####-####-####-############}">
    <Source itemUuid="{########-####-####-####-############}" x="..." y="..."/>
    <Target itemUuid="{########-####-####-####-############}" x="..." y="..."/>
    <TextBoxVectorFromCenter x="..." y="..."/>
</ConnectionItem>
```

Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<DiagramData>
    <Diagram uuid="{61af3933-5b91-47c5-b302-b2351f80700d}">
        <ConnectionItem type="Transition"
                        uuid="{667719a3-a6c9-476c-8fda-f0055a6ccf0e}">
            <Source itemUuid="{f396f156-32d1-45a2-a2f4-9249d74457e5}"
                    x="-140" y="-60"/>
            <Target itemUuid="{1c4d3b43-7eb4-48ad-b5d6-b60958b7014c}"
                    x="-140" y="-120"/>
            <TextBoxVectorFromCenter x="0" y="0"/>
        </ConnectionItem>
        <ConnectionItem type="Transition"
                        uuid="{0b2ecfa6-55c9-4379-87e7-2261c81826d2}">
            <Source itemUuid="{1c4d3b43-7eb4-48ad-b5d6-b60958b7014c}"
                    x="-80" y="-120"/>
            <Target itemUuid="{f396f156-32d1-45a2-a2f4-9249d74457e5}"
                    x="-80" y="-60"/>
            <TextBoxVectorFromCenter x="31" y="7"/>
        </ConnectionItem>
        <ConnectionItem type="Transition"
                        uuid="{9f699367-d952-4bdf-ac8e-370131f94c04}">
            <Source itemUuid="{82ecfb72-b44c-487f-81b7-e615b6ef7585}"
                    x="-80" y="-240"/>
            <Target itemUuid="{1c4d3b43-7eb4-48ad-b5d6-b60958b7014c}"
                    x="-80" y="-180"/>
            <TextBoxVectorFromCenter x="0" y="0"/>
        </ConnectionItem>
        <ConnectionItem type="Transition"
                        uuid="{79bb9cd2-700a-4829-a225-6cd248f2244c}">
            <Source itemUuid="{98057502-299c-44ff-8fa0-5af5b6021f00}"
                    x="-78.7670075952098" y="-352.602045571259"/>
            <Target itemUuid="{82ecfb72-b44c-487f-81b7-e615b6ef7585}"
                    x="-79" y="-300"/>
            <TextBoxVectorFromCenter x="0" y="0"/>
        </ConnectionItem>
```

```
        <DiagramItem type="State" width="180" height="60"
                     x="-200" y="-60" uuid="{f396f156-32d1-45a2-a2f4-9249d74457e5}">
            <Region uuid="{05efb9b0-ac6c-43ca-b327-baffd793ccf8}" width="160"/>
        </DiagramItem>
        <DiagramItem type="State" width="180" height="60"
                     x="-200" y="-180" uuid="{1c4d3b43-7eb4-48ad-b5d6-b60958b7014c}">
            <Region uuid="{6eb821da-0d9a-4f3a-a0f7-a788620d7d99}" width="160"/>
        </DiagramItem>
        <DiagramItem type="State" width="120" height="60"
                     x="-140" y="-300" uuid="{82ecfb72-b44c-487f-81b7-e615b6ef7585}">
            <Region uuid="{fb77dbbe-62c3-426e-98a4-0a4e643e4600}" width="100"/>
        </DiagramItem>
        <DiagramItem type="InitialState" width="15" height="15"
                     x="-80" y="-360" uuid="{98057502-299c-44ff-8fa0-5af5b6021f00}"/>
    </Diagram>
</DiagramData>
```

12.2 Programm-Code

Der gesamte Source-Code, sowie die generierte Doxygen-Dokumentation befinden sich auf der beigelegten CD.

12.3 Benutzerhandbuch

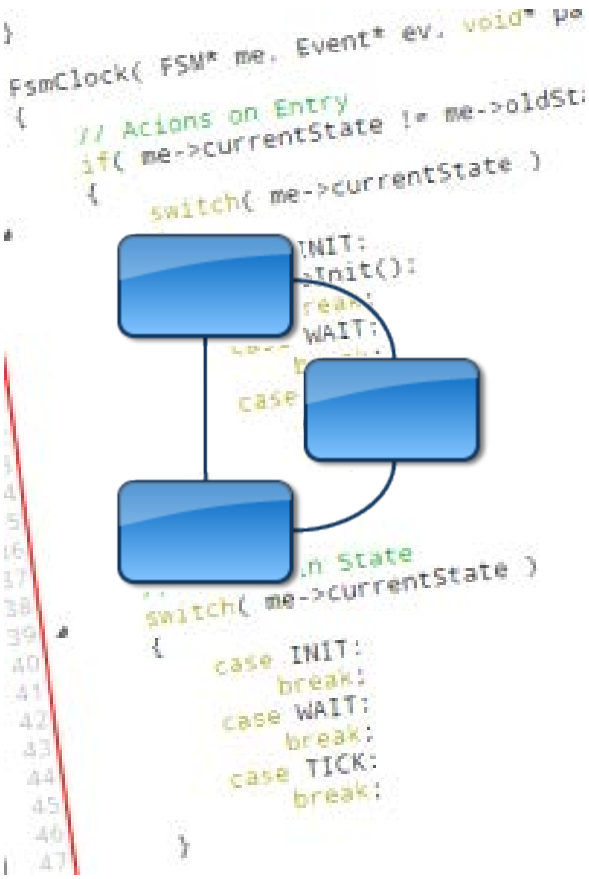
Folgend ist das Benutzerhandbuch zum State Chart Editor zu finden.

State Chart Editor

Model based Software Development Tool

University of Applied Sciences – Western Switzerland

Systems Engineering



Developer: **Ralph Martig**

Inhaltsverzeichnis

1 Beschreibung

Der State Chart Editor ist ein modellbasiertes Entwicklungstool, dass es erlaubt, Zustandswechseldiagramme zu erstellen.

Nebst dem Zeichnen der Zustandsmaschinen, kann zudem Programmcode in Zuständen oder auf Transitionen implementiert werden. Die Diagramme können gedruckt oder zur weiteren Verwendung in die Zwischenablage kopiert werden.

Das Modell wird in eine Archiv-Datei gespeichert, welche die graphischen und logischen Informationen in der Form von XML enthält. Die logischen Informationen können zudem exportiert werden und in einem nächsten Schritt von einem Code-Generator weiterverarbeitet werden.

2 Der State Chart Editor

Die Folgende Abbildung zeigt das Hauptfenster des Entwicklungstool.

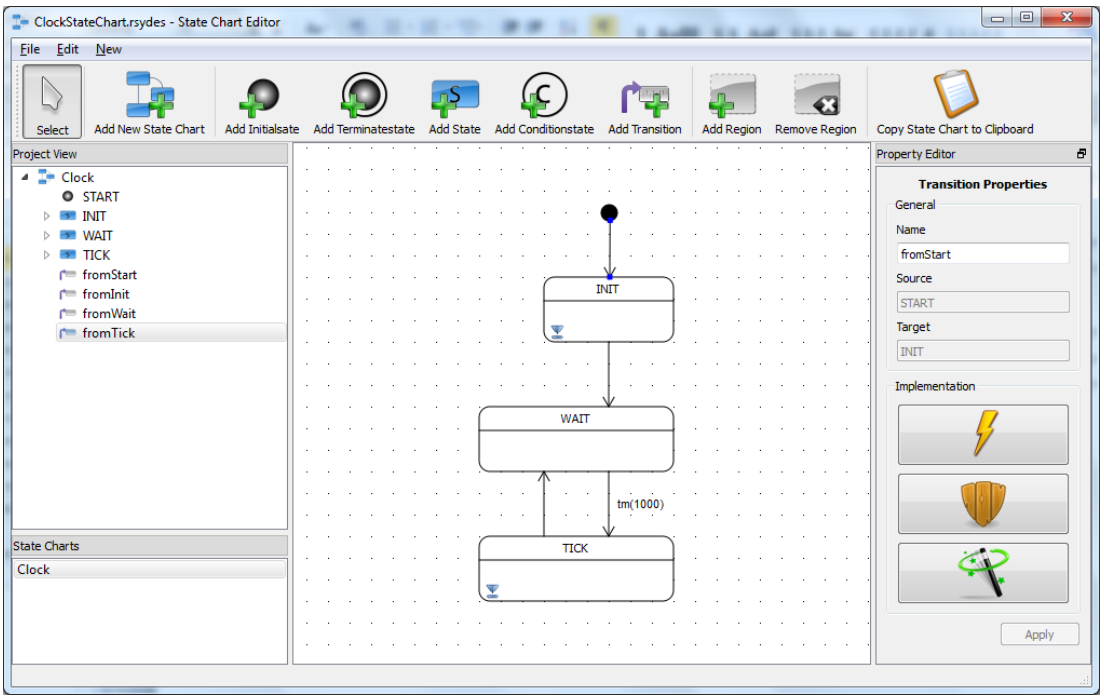


Abbildung 1: Das Hauptfenster

Das Hauptfenster ist in drei Bereiche aufgeteilt. Ganz links befindet sich eine Projekt- und State-Chart-Ansicht, die einen Überblick zu den Zustandsmaschinen verschaffen. In der Mitte befindet sich die Arbeitsfläche, auf welcher die Zustandsmaschinen gezeichnet werden können. Ganz rechts wird der Property-Editor angezeigt, welcher die Eigenschaften des ausgewählten Elements anzeigt.

Eine Menüleiste, sowie eine Toolbar bieten verschiedene Funktionalitäten. Am unteren Fensterrand zeigt eine Statusleiste Informationen (beispielsweise während des Ladevorgangs) an.

2.1 Die Ansichten

Die Projekt- sowie die State-Chart-Ansicht verschaffen dem Benutzer einen Überblick zu den Zustandsmaschinen.

2.1.1 Die Projekt-Ansicht

Die Projekt-Ansicht listet alle Elemente, die zum Zeichnen der Zustandsmaschine verwendet wurden, hierarchisch auf.

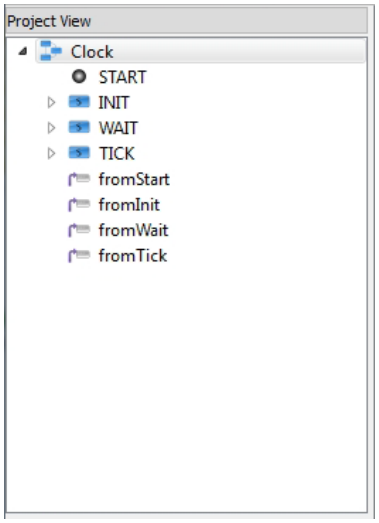


Abbildung 2: Die Projekt-Ansicht

2.1.2 Die State-Chart-Ansicht

Jede Zustandsmaschine wird in der State-Chart-Ansicht aufgelistet. Durch Auswählen einer Zustandsmaschine in dieser Ansicht, kann diese geöffnet und angezeigt werden.

Abbildung 3: Die State-Chart-Ansicht

2.2 Die Arbeitsfläche

Auf der Arbeitsfläche kann der Benutzer die Zustandsmaschinen zeichnen. Es wird jeweils die ausgewählte Zustandsmaschine angezeigt.

2.3 Der Property-Editor

Im Property-Editor werden jeweils die Eigenschaften des ausgewählten Elements angezeigt und können vom Benutzer angepasst werden.

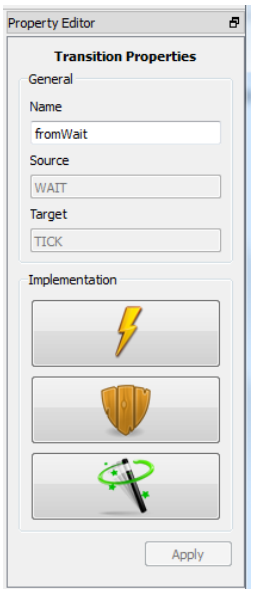


Abbildung 4: Der Property-Editor

3 Neues Projekt starten

Wird der State Chart Editor gestartet, wird automatisch ein neues leeres Projekt erstellt.

Um während der Benutzung des Editors ein neues Projekt zu erstellen, erfolgt dies durch wählen von ***New File*** im Menü ***File***.

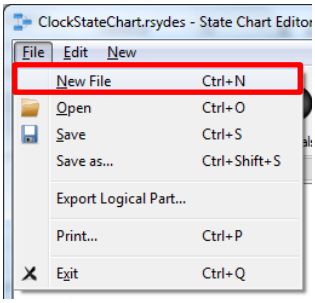


Abbildung 5: New File

Wenn ein neues Projekt erstellt werden will und bereits ein Projekt geöffnet ist, an welchem Änderungen vorgenommen worden sind, wird nachgefragt, ob die Änderungen gespeichert werden sollen.

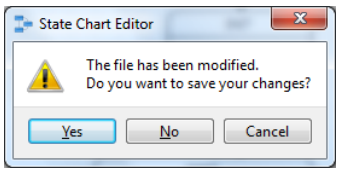


Abbildung 6: Dialog "Änderungen speichern?"

4 Eine Zustandsmaschinen hinzufügen und entfernen

Um dem Projekt ein neues Zustandswechseldiagramm hinzuzufügen, wird im Menü ***New*** oder in der Toolbar ***Add New State Chart*** angeklickt.

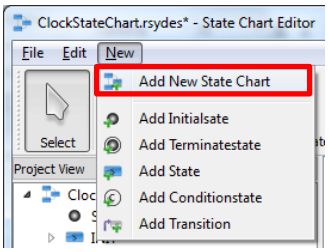


Abbildung 7: «Add New State Chart» im Menü New

Abbildung 8: «Add New State Chart» in der Toolbar

Die hinzugefügte Zustandsmaschine wird auf der Arbeitsfläche angezeigt und den Ansichten (Projekt- / State-Chart-Ansicht) hinzugefügt.

5 Eine Zustandsmaschine anzeigen

Durch Anklicken einer Zustandsmaschine in der State-Chart-Ansicht (Abbildung 3), kann diese auf der Arbeitsfläche angezeigt und bearbeitet werden.

6 Eine Zustandsmaschine löschen

Um eine Zustandsmaschine vom Projekt zu löschen, muss diese in der State-Chart-Ansicht (Abbildung 3) ausgewählt und die ***Delete***-Taste gedrückt werden.

7 Eine Zustandsmaschine zeichnen

Sämtliche Elemente die zum Zeichnen einer Zustandsmaschine benötigt werden, sind im Menü ***New*** zu finden.

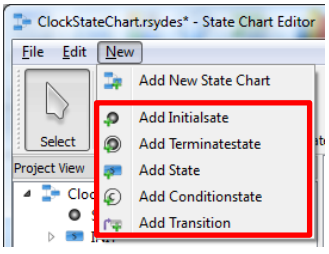


Abbildung 9: Menü New

7.1 Elemente hinzufügen

Um dem Zustandswechseldiagramm ein Element hinzuzufügen, wird im Menü ***New*** oder in der Toolbar das entsprechende Element ausgewählt.

Anschliessend drückt man in der Arbeitsfläche an der gewünschten Position die linke Maustaste. Eine Vorschau des einzufügenden Elements erscheint. Durch bewegen der Maus kann das Element wunschgemäss positioniert werden. Beim Loslassen der Maustaste wird das Element schliesslich platziert und am Raster ausgerichtet.

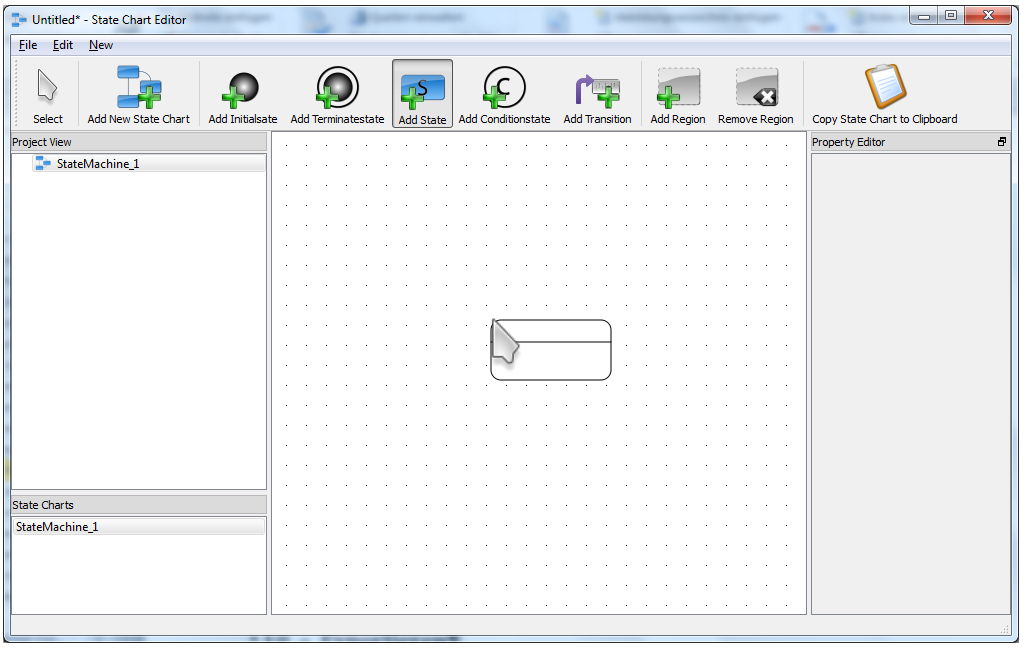


Abbildung 10: Vorschau des Elements beim Einfügen

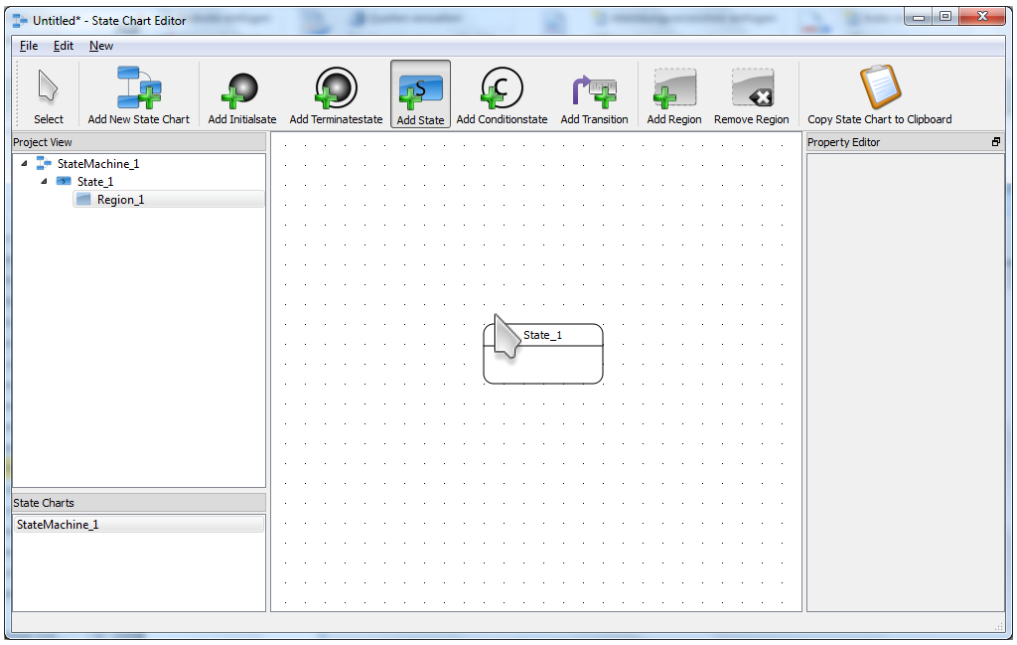


Abbildung 11: Beim Einfügen wird das Element am Raster ausgerichtet

7.2 Elemente markieren

Zum Markieren von Elementen wird in den Selektierungsmodus gewechselt. Der Selektierungsmodus wird durch Anwählen von ***Select*** , im Menü ***Edit*** oder in der Toolbar, aktiviert.

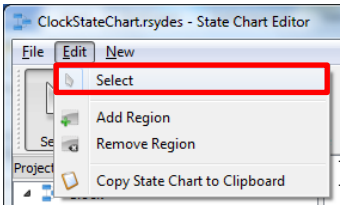


Abbildung 12: «Select» Menü Edit

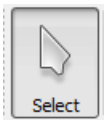


Abbildung 13: «Select» in der Toolbar

Die Elemente können nun durch Anklicken mit der linken Maustaste markiert werden. Eine Mehrfachauswahl ist möglich durch Ziehen eines Rahmens um die Elemente oder durch Gedrückt halten der CTRL-Taste.

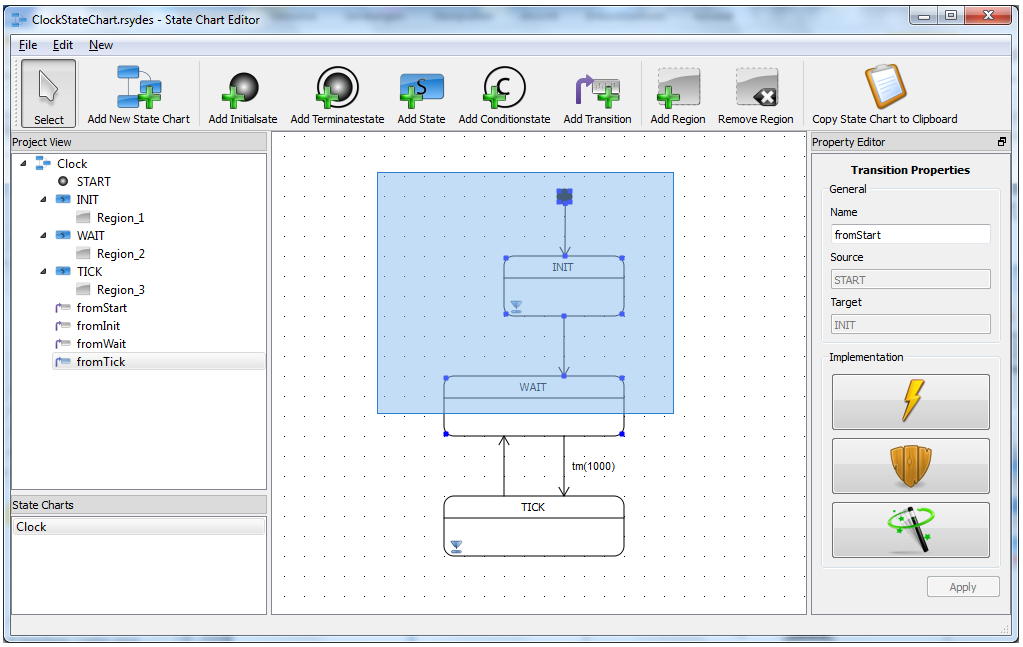


Abbildung 14: Mehrfachauswahl durch Ziehen eines Rahmens

7.3 Elemente entfernen

Um ein Element oder mehrere Elemente zu entfernen, wird dieses bzw. diese markiert. Durch anschliessendes Drücken der ***Delete***-Taste  wird das Elemente / die Elemente gelöscht.

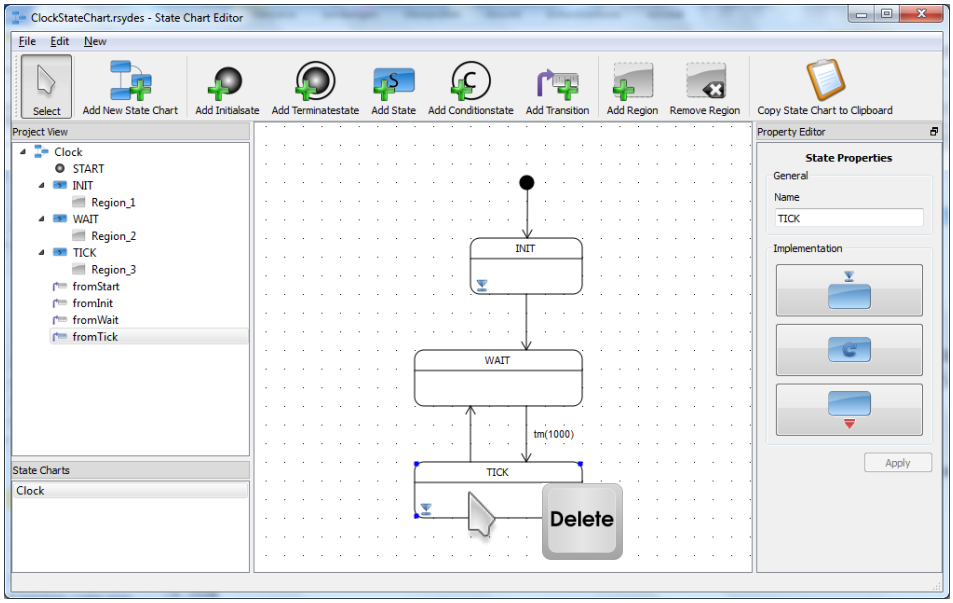


Abbildung 15: Element(e) markieren zum Entfernen

Bemerkung: Falls ein Element weitere Elemente enthält oder ein-/ausgehende Transitionen besitzt, werden diese ebenfalls gelöscht.

7.4 Elemente verschieben

Zum Verschieben von Elementen müssen diese zuerst markiert werden, anschliessend können sie mit gedrückter linker Maustaste verschoben werden. An der gewünschten Stelle wird die Maustaste wieder losgelassen.

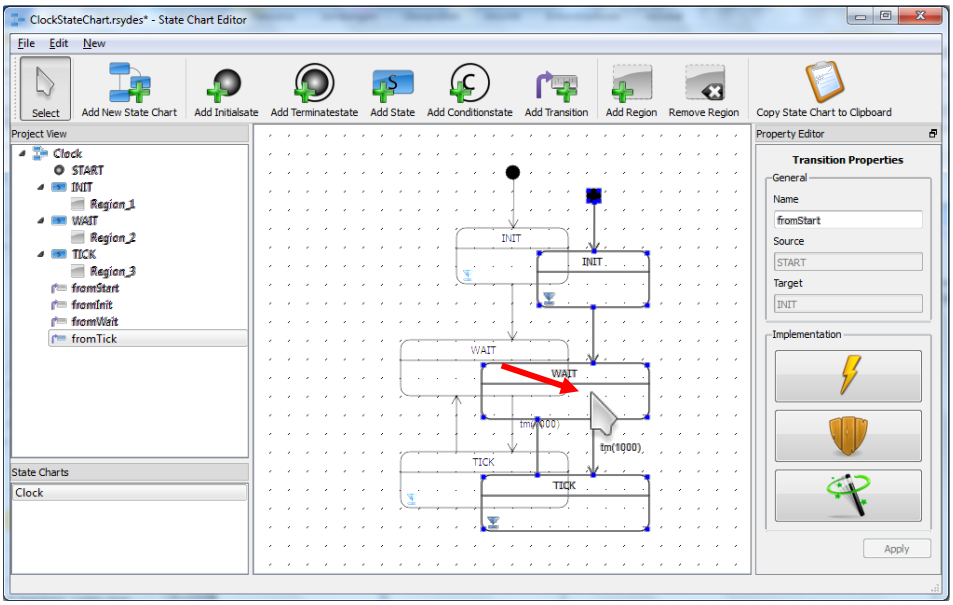


Abbildung 16: Elemente verschieben

7.5 Einem Zustand Regionen hinzufügen / entfernen

Hinzufügen

Standardmässig besitzt ein Zustand eine Region. Um dem Zustand weitere Regionen hinzuzufügen, wird ***Add Region*** , im Menü ***Edit*** oder in der Toolbar, ausgewählt. Danach wird der Zustand, welchem die Region hinzugefügt werden will, mit der linken Maustaste angeklickt.

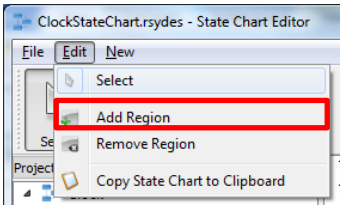


Abbildung 17: «Add Region» Menü Edit

Abbildung 18: «Add Region» in der Toolbar

Im Selektierungsmodus kann durch Verschieben (mit linker Maustaste) der Region-Begrenzungslinie, die Breite der Region geändert werden.

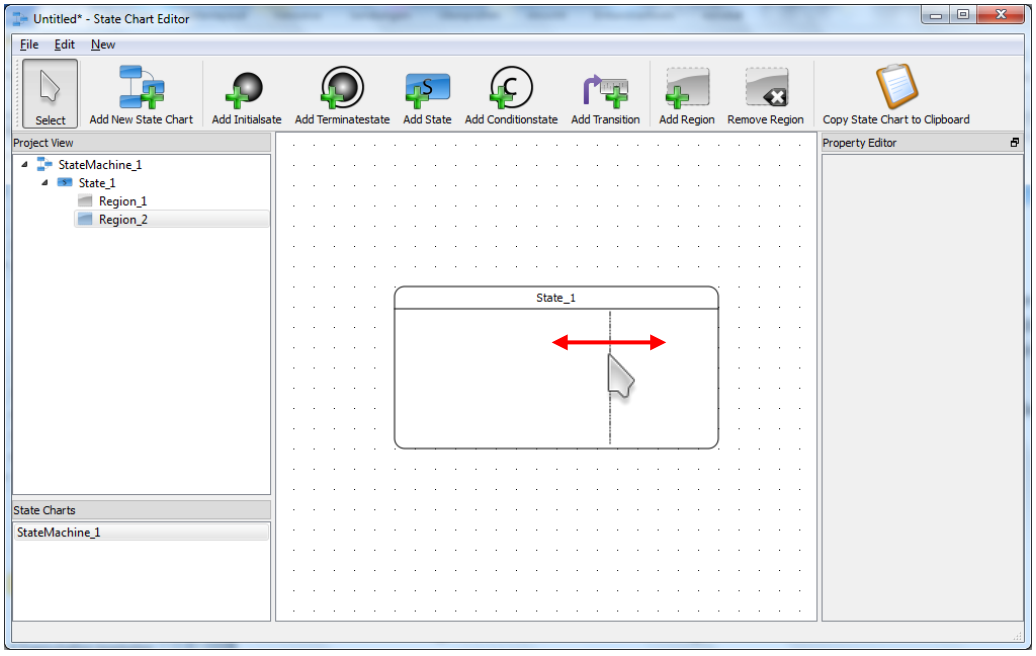


Abbildung 19: Ändern der Breite einer Region

Entfernen

Soll eine hinzugefügte Region wieder entfernt werden, wird ***Remove Region***  , im Menü ***Edit*** oder in der Toolbar, gewählt. Anschliessend wird mit der linken Maustaste in die zu entfernende Region geklickt, die Region wird entfernt.

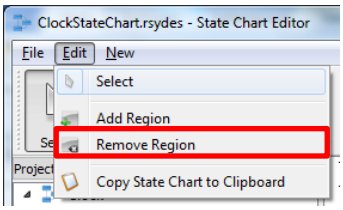


Abbildung 20: «Add Region» Menü Edit

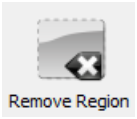


Abbildung 21: «Add Region» in der Toolbar

Bemerkung: Damit eine Region entfernt werden kann, darf sie keine Elemente enthalten. Ein Zustand muss mindestens eine Region besitzen.

7.6 Transitionen erstellen

Zum Erstellen einer Transition wird ***Add Transition***, im Menü ***New*** oder in der Toolbar, ausgewählt.

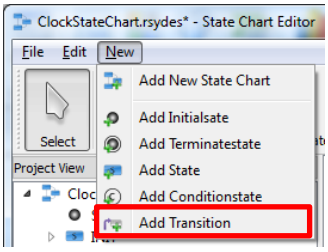


Abbildung 22: «Add Transition» im Menü New

Abbildung 23: «Add Transition» in der Toolbar

Schliesslich kann die Transition gezeichnet werden. Dazu wird am Source-Element, an der gewünschten Stelle, die linke Maustaste gedrückt. Mit gedrückter Maustaste wird die Transition zum Target-Element gezogen. Beim Loslassen der Maustaste wird die Transition erstellt.

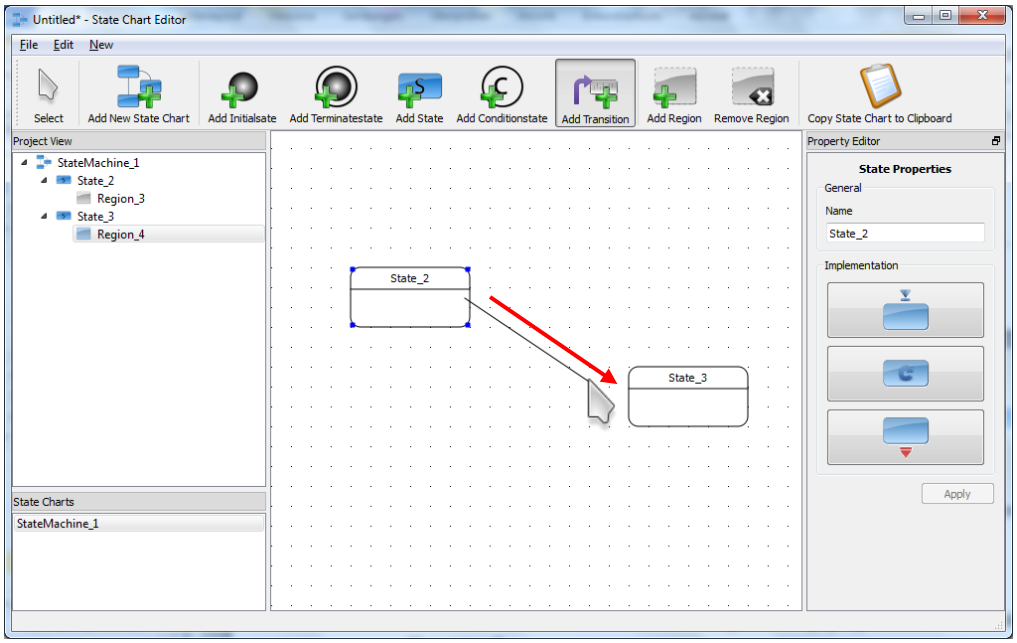


Abbildung 24: Erstellen einer Transition

Wichtig: Beim Klicken und Loslassen muss sich die Maus innerhalb des Elements befinden.

7.7 Transitionen-Text verschieben

Um den auf der Transition angezeigten Text zu verschieben, klickt am diesen mit der linken Maustaste an und verschiebt ihn an die gewünschte Stelle.

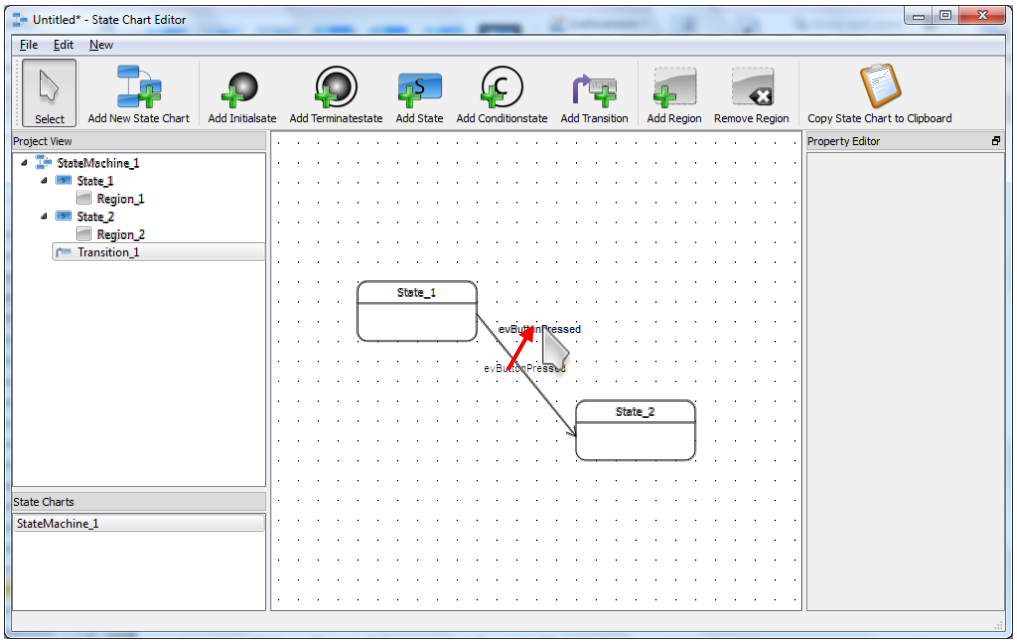


Abbildung 25: Text verschieben

7.8 Transitionen entfernen

Um eine Transition zu entfernen wird gleich vorgegangen, wie beim Entfernen von Elementen. Siehe 7.3 Elemente entfernen (S. 7).

8 Eigenschaften bearbeiten

Damit die Eigenschaften eines Elementes bearbeitet werden können, wird das Element auf der Arbeitsfläche, in der Projekt- oder State-Chart-Ansicht mit der linken Maustaste angeklickt.

Im Property-Editor (Abbildung 4) werden die Eigenschaften des Elements angezeigt und können bearbeitet werden.

8.1 Code implementieren

Bei den Zuständen oder Transitionen ist es möglich, das Verhalten zu implementieren. Zum Implementieren des Verhaltens kann der entsprechende Button im Property-Editor angeklickt werden und ein Code-Editor zur Implementierung wird geöffnet.

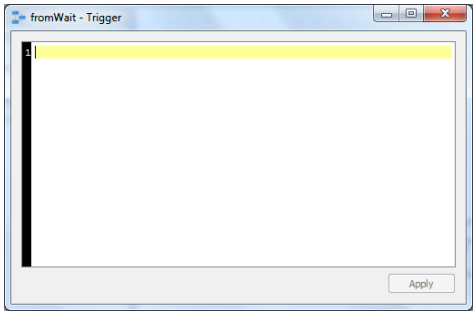


Abbildung 26: Code-Editor

9 Exportieren

Die logischen Informationen der im Projekt enthaltenen Zustandswechseldiagramme können als XML-Datei exportiert werden.

Diese Datei kann in einem nächsten Schritt, beispielsweise von einem Code-Generator, weiterverarbeitet werden.

Zum Exportieren dieser Informationen wählt man im Menü ***File*** den Eintrag ***Export Logical Part***.

Bemerkung: Das Projekt muss gespeichert sein, damit der Export möglich ist.

10 In Zwischenablage kopieren

Durch Anklicken von ***Copy State Chart to Clipboard***, im Menü ***Edit*** oder in der Toolbar, wird die gesamte angezeigte Zustandsmaschine als Bild in die Zwischenablage kopiert.

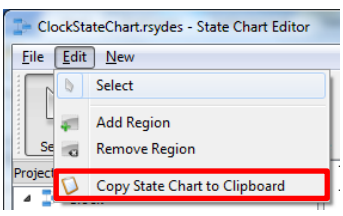


Abbildung 27: «Copy State Chart to Clipboard» Menü Edit

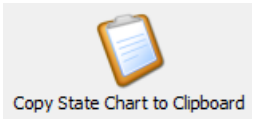


Abbildung 28: «Copy State Chart to Clipboard» in der Toolbar

11 Drucken

Die aktuell angezeigte Zustandsmaschine kann ausgedruckt werden. Dazu wählt man im Menü ***File*** die entsprechende Aktion ***Print***.

Bemerkung: Im Druck-Dialog vorgenommene Änderungen haben keinen Einfluss auf den Druckvorgang. Es wird jeweils die gesamte angezeigte Zustandsmaschine ausgedruckt.

12 Beispiele

Im Verzeichnis des State Chart Editor existiert ein Odner „Examples“, der zwei Beispiel-zustandsmaschinen beinhaltet. Die Beispiele können mit dem Editor geöffnet werden.