

CHRISTIAN-ALBRECHTS-UNIVERSITÄT ZU KIEL

Bachelorarbeit

# Datenvisualisierung in grafischen Modellen

KIELER Visualization of Data

John Julian Carstens

30. September 2010



Institut für Informatik  
Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden

betreut durch:  
Dipl.-Inf. Christian Motika



## Eidesstattliche Erklärung

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

Kiel,

---



## **Zusammenfassung**

Der Ansatz grafischer Modellierung ist die Vereinfachung des Designs durch Einführung einer zusätzlichen Abstraktionsebene, die zwischen Realität und Implementierung vermittelt. Zudem existieren mittlerweile viele Hilfsmittel, die das grafische Modellieren komfortabler und handhabbarer machen, wie beispielsweise automatisches Layout.

Möchte man ein grafisches Modell simulieren oder untersuchen, so muss geklärt werden, auf welche Weise das Verhalten des Modells dargestellt werden kann. Diese Arbeit beschäftigt sich mit der Visualisierung anfallender Daten in diesem Kontext. Es werden hierbei statische und dynamische Visualisierungsmethoden vorgestellt, wie beispielsweise die Animation eines Datenflusses oder das Anzeigen einer Wertehistorie. Zusätzlich wird eine Implementierung einiger der vorgestellten Methoden im Zuge des Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)-Projekts vorgestellt.



# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
1.1	Aufgabenstellung . . . . .	2
1.2	Struktur des folgenden Dokuments . . . . .	2
<b>2</b>	<b>Grundlegende Betrachtungen</b>	<b>5</b>
2.1	Verwandte Arbeiten . . . . .	5
2.2	Betrachtung existierender Werkzeuge . . . . .	6
2.2.1	Ptolemy II . . . . .	6
2.2.2	Matlab/Simulink . . . . .	7
2.2.3	Safety Critical Application Development Environment . . . . .	9
<b>3</b>	<b>Verwendete Technologien</b>	<b>13</b>
3.1	Eclipse . . . . .	13
3.1.1	Eclipse Plug-in Mechanismus . . . . .	13
3.1.2	Graphical Editing Framework / Graphical Modeling Framework	14
3.2	Kiel Integrated Environment for Layout Eclipse Rich Client . . . . .	15
3.2.1	KIELER Execution Manager . . . . .	15
3.2.2	KIELER Infrastructure for Meta Layout . . . . .	15
3.2.3	KIELER Actor Oriented Modeling . . . . .	16
3.3	Multi-View Editor Kit . . . . .	17
3.3.1	Animations-Paket . . . . .	17
3.4	Ptolemy II Plotter . . . . .	17
3.5	Datenformate . . . . .	17
3.5.1	JavaScript Object Notation . . . . .	17
3.5.2	Comma Separated Values . . . . .	18
<b>4</b>	<b>Datenvisualisierung</b>	<b>19</b>
4.1	Begriffsklärung . . . . .	19
4.2	Flüchtige Visualisierung . . . . .	20
4.2.1	Anzeigemöglichkeiten für Marken . . . . .	22
4.2.2	Animierte Marken . . . . .	23
4.2.3	Nutzung vorhandener Elemente . . . . .	24
4.3	Persistente Visualisierung . . . . .	25
4.3.1	Einbetten von Diagrammen . . . . .	25
4.3.2	Einbetten von Grafiken . . . . .	27

<b>5</b>	<b>Implementierung</b>	<b>29</b>
5.1	Überblick . . . . .	30
5.2	Adressierung von Modellelementen . . . . .	31
5.3	Datenverwaltung . . . . .	33
5.3.1	Laufzeit-Einstellungen . . . . .	34
5.4	Datenhaltung . . . . .	35
5.5	Datenquellen . . . . .	36
5.6	Flüchtige Visualisierung mit Marken . . . . .	38
5.6.1	Animation mit dem Multi-View Editor Kit . . . . .	39
5.7	Persistente Visualisierungsobjekte . . . . .	41
5.7.1	Graphen- und Diagrammknoten . . . . .	44
<b>6</b>	<b>Mögliche Erweiterungen von KIELER Visualization of Data</b>	<b>47</b>
<b>7</b>	<b>Fazit</b>	<b>49</b>
<b>A</b>	<b>Arbeiten mit KIELER Visualization of Data</b>	<b>51</b>
A.1	Voraussetzungen . . . . .	51
A.2	Aktivierung im Execution Manager . . . . .	51
A.3	Einstellungen zur Laufzeit . . . . .	52
A.4	Hinzufügen von eingebetten Elementen . . . . .	53
<b>B</b>	<b>Literaturverzeichnis</b>	<b>55</b>



# Abbildungsverzeichnis

2.1	Datenvisualisierung in einem Aktor . . . . .	7
2.2	Datenvisualisierung in einem neuen Fenster . . . . .	8
2.3	Datenvisualisierung im Modell . . . . .	8
2.4	Im neuen Fenster geöffneter Graph . . . . .	9
2.5	Brille als Zeichen für eine Scope . . . . .	9
2.6	Visualisierungsmöglichkeiten in SCADE . . . . .	10
3.1	Plug-in Architektur von Eclipse . . . . .	14
3.2	Die MVC Struktur von GEF . . . . .	15
3.3	Die Infrastruktur von KIEM . . . . .	16
3.4	Ein mit KAOM geöffnetes Ptolemy II Modell . . . . .	16
4.1	Darstellung eines Datums mithilfe einer Marke . . . . .	21
4.2	Behandlung von langen Visualisierungen . . . . .	21
4.3	Verschiedene Möglichkeiten der Platzierung . . . . .	22
4.4	Animierte Marke . . . . .	23
4.5	Animierte Marke ohne textuelle Visualisierung . . . . .	24
4.6	Hervorhebung durch Färbung . . . . .	24
4.7	In das Modelldiagramm eingebundener Ptolemy II Plotter . . . . .	25
4.8	Struktur einer Grundlage für Visualisierungskomponenten . . . . .	26
4.9	In das Modelldiagramm eingebundene Grafik . . . . .	27
5.1	Die MVC Struktur des KIELER-Projekts . . . . .	29
5.2	Ablauf einer Visualisierung . . . . .	30
5.3	Pakete des KViD-Plug-ins . . . . .	31
5.4	Modell und Baumdarstellung . . . . .	32
5.5	Beispieladressen für das Modellelement „Ramp“ . . . . .	33
5.6	Interface eines Datenquellen-Beobachters . . . . .	33
5.7	Ansicht der Benutzeroberfläche für Einstellungen . . . . .	35
5.8	Methoden und Attribute eines DataObjects . . . . .	36
5.9	Modell und generierte Daten des Simulators . . . . .	37
5.10	Interface für eine Grafik mit Datenverwaltung . . . . .	38
5.11	Die verschiedenen grafischen Ebenen von GEF . . . . .	39
5.12	Vorgang zur Erstellung eines EditParts oder einer View . . . . .	43
5.13	Der Request-Mechanismus von GEF . . . . .	44

*Abbildungsverzeichnis*

A.1	Ansicht der globalen Einstellungen . . . . .	52
A.2	Hinzufügen eines Diagrammelements . . . . .	54

# Listings

3.1	Beispiel einer CSV-Datei mit Zahlenwerten . . . . .	18
5.1	Beispiel einer richtig formatierten CSV-Datei . . . . .	37
5.2	Erstellen des Animierungskommandos . . . . .	40
5.3	Extension Point Definition für einen ElementType . . . . .	41
5.4	Extension Point Definition für den View- und EditPartProvider . . .	43
5.5	Extension Point Definition für die Fabrik . . . . .	44
5.6	Implementierung der EditPart-Fabrik . . . . .	45
5.7	Extension Point Definition für den ElementType . . . . .	45
5.8	Extension Point Definition für den Menüeintrag . . . . .	46

*Listings*

# Abkürzungsverzeichnis

<b>API</b>	Application Programming Interface
<b>CSV</b>	Comma Separated Values
<b>EMF</b>	Eclipse Modeling Framework
<b>GEF</b>	Graphical Editing Framework
<b>GMF</b>	Graphical Modeling Framework
<b>GUI</b>	Graphical User Interface
<b>IBM</b>	International Business Machines
<b>JSON</b>	JavaScript Object Notation
<b>KAOM</b>	KIELER Actor Oriented Modeling
<b>KARMA</b>	KIELER Advanced Rendering for Model Appearance
<b>KEV</b>	KIELER Environment Visualization
<b>KIELER</b>	Kiel Integrated Environment for Layout Eclipse Rich Client
<b>KIEM</b>	KIELER Execution Manager
<b>KIML</b>	KIELER Infrastructure for Meta Layout
<b>KViD</b>	KIELER Visualization of Data
<b>MVC</b>	Model-View-Controller
<b>SCADE</b>	Safety Critical Application Development Environment
<b>SVG</b>	Scalable Vector Graphics
<b>UML</b>	Unified Modeling Language
<b>URI</b>	Uniform Resource Identifier



# 1. Einführung

Um eine gute und reibungslose Interaktion zwischen Mensch und Maschine zu erreichen, ist die Wahl der richtigen Methode zur Visualisierung der für den Menschen relevanten Daten von besonderer Bedeutung. Ohne eine solche grafische Schnittstelle bleiben die Zusammenhänge im Inneren des Computers in der Regel verborgen. Eine sinnvolle Nutzung der Daten wird somit unmöglich. Es reicht auch nicht aus, eine beliebige und unüberlegte Methode der Visualisierung zu wählen. Das einfache Anzeigen der Bit-Darstellung eines Textes ist nutzlos, wenn man diesen Text sinnvoll lesen oder bearbeiten möchte. Die auf dem Computer gespeicherten Daten müssen in eine vom Menschen les- und verstehbare Form gebracht werden.

Betrachtet man die Modellierung komplexer Systeme, so kann es passieren, dass auch die Darstellung durch an sich sinnvollen Text nicht mehr ausreicht, um ein Modell schnell und richtig zu verstehen. Es werden daher Modellierungswerkzeuge gewählt, die komplexe Zusammenhänge schematisch-grafisch darstellen.

Es gibt Werkzeuge, die es ermöglichen, den modellierten Zusammenhang zu simulieren, um sein potentiell Verhalten in einem realen System zu erfahren. Auch hier können Daten anfallen, die sinnvoll visualisiert werden müssen.

Ebenso ist es vorstellbar, dass zu einem grafischen Modell eine Menge an gemessenen oder simulierten Daten, etwa in einer Datenbank, vorliegt. Man kann viel Mühe sparen, wenn diese Daten automatisch grafisch mit dem Modell in Verbindung gebracht werden.

Mit diesen Arten der Visualisierung befasst sich die folgende Arbeit. Je nach Anwendungsfall sollen relevante Daten auf vielfältige Weise direkt im Diagramm dargestellt werden. Dies kann das einfache grafische Zuordnen einer Zahl zu einem Modellelement sein, aber auch eine komplexe Animation, die den Datenfluss mehrerer Leitungen durch Bewegung der relevanten Daten repräsentiert.

Der Name der zu dieser Arbeit zugehörigen Implementierung lautet KIELER Visualization of Data (KViD). Sie wird in das Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)-Projekt integriert, welches eine grafische Entwicklungsumgebung für modellbasiertes Design ist.

## 1. Einführung

### 1.1. Aufgabenstellung

Die Aufgabe der Datenvisualisierung muss in Teilbereiche aufgeteilt werden, um eine vernünftige wissenschaftliche und umfassende Bearbeitung zu ermöglichen. Es lassen sich vier Teilaufgaben erschließen, deren Lösung zu einer Lösung des gesamten Problems führen:

1. Das Finden verschiedener Möglichkeiten, vielfältige Daten eingängig und leicht verständlich darzustellen.
2. Die für eine Darstellung von Daten erforderliche Bekanntgabe der Daten an eine Visualisierungsumgebung. Zu diesem Zweck müssen eine oder mehrere Anbindungen an eine Datenquelle oder Simulationsumgebung implementiert werden.
3. Die Implementierung und Einbettung der Darstellungsmöglichkeiten in eine grafische Entwicklungsumgebung, in der die zu den Daten gehörenden Modellen vorliegen.
4. Das Verbinden der Daten aus der Datenquelle oder Simulationsumgebung mit den grafischen Darstellungsmöglichkeiten.

Zu der umfassenden Problemlösung gehört also nicht nur eine theoretische Betrachtung des Themas, sondern auch eine Implementierung, die die Machbarkeit bestätigt und die theoretische Überlegungen durch eine praktische Anwendung veranschaulicht.

### 1.2. Struktur des folgenden Dokuments

In Abschnitt 2.1 werden Grundlagen zum Thema Datenvisualisierung diskutiert und verwandte Arbeiten betrachtet. Dabei werden zunächst allgemeine Arbeiten zum Thema betrachtet, bevor unmittelbar verwandte Arbeiten angeführt werden. Abschnitt 2.2 beschäftigt sich mit Werkzeugen zur grafischen Modellierung von komplexen Zusammenhängen, hauptsächlich im Bereich der Elektrotechnik und der Modellierung eingebetteter Echtzeitsysteme. Auch hier liegt das Augenmerk auf der grafischen Darstellung relevanter Daten. Auf die verschiedenen Technologien, die zur Realisierung der erarbeiteten Konzepte benutzt werden sollen, wird in Kapitel 3 eingegangen.

Im Kapitel 4 werden die schließlich zu implementierenden Ansätze besprochen. Es folgt dann eine Beschreibung der fertigen Implementierung in Kapitel 5. Es werden dort die Umsetzung, die Ergebnisse und die Funktionsweise vorgestellt.

Darauf folgend werden im Kapitel 6 Möglichkeiten vorgestellt, wie man das fertige Werkzeug noch weiter verbessern kann. Dies können Ideen sein, die es nicht mehr in das fertige Werkzeug geschafft haben, aber auch neue Ideen oder Verbesserungen, die erst im Laufe der Arbeit entdeckt wurden.



## 1.2. Struktur des folgenden Dokuments

Abschließend wird in Kapitel 7 ein Fazit gezogen und das im Laufe der Arbeit Erreichte zusammengefasst. In Appendix A wird erklärt, auf welche Weise man das fertige Werkzeug benutzen kann. Es werden die Anbindung an eine Datenquelle und die entsprechenden Voraussetzungen beschrieben. Auch die Benutzung der verschiedenen Einstellmöglichkeiten über eine grafische Oberfläche wird hier detailliert dargestellt.

## *1. Einführung*

## 2. Grundlegende Betrachtungen

Das folgende Kapitel soll einen Überblick darüber geben, welche Arbeiten zum Thema Datenvisualisierung bereits existieren. Zunächst werden wissenschaftliche Arbeiten angeführt, welche sich konkret oder im weiteren Sinne mit dem Thema Visualisierung beschäftigen. Dabei wird eine Abgrenzung geschaffen und dann über Grundlagenarbeit zu konkreteren Arbeiten, die im Zusammenhang mit KViD stehen können, hingeführt.

Im zweiten Abschnitt des Kapitels werden Visualisierungen in bereits existierenden Werkzeugen betrachtet. Es werden Vor- und Nachteile analysiert und Eigenschaften herausgestellt, die im Zusammenhang mit KViD nutzbar sind.

### 2.1. Verwandte Arbeiten

Das Thema der Datenvisualisierung ist ein weites Feld, über das bereits viele wissenschaftliche Arbeiten verfasst wurden. Insbesondere das Thema *Data Mining*, wie es beispielsweise von Han et. al. [13] beschrieben wird, wird oft im Zusammenhang mit der Visualisierung genannt. Schränkt man sich jedoch auf die Datenvisualisierung in grafischen Modellen ein, so erhält man eine überschaubare Auswahl.

Ein grundlegendes Plädoyer für die Nutzung von Visualisierung in der Entwicklung stammt von Scott Dyer [5]. Er kritisiert, dass trotz des großen Nutzens von Visualisierung bei der Lösung komplexer Probleme zum damaligen Zeitpunkt keine Software existierte, die ausreichende Methoden der Visualisierung anbietet. Dyer beschreibt ein System, das beispielsweise wissenschaftliche Daten wie die Temperaturen eines Sees visualisieren kann.

Ricardo Orosco und Roberto Moriyón [21] präsentieren ein generisches, objektorientiertes Framework zur Erstellung beliebiger Visualisierungssysteme. Sie erstellen keine konkrete Visualisierungsmethode für einen einzelnen Anwendungsfall. Stattdessen wird eine Grundlage geschaffen, auf der möglichst einfach neue Visualisierungen für verschiedene Anwendungsfälle implementiert werden können. Ein solcher Ansatz wird auch von dieser Arbeit verfolgt.

Grundlagenforschung zum Thema Animation in grafischen Verhaltensmodellen leisten Magee et. al. [16]. Das Verhalten von Modellen soll durch die Animation verdeutlicht werden und es wird untersucht, wie diese Art der Visualisierung bei der Analyse eines Modells helfen kann.

Ein Visualisierungssystem für Datenflussmodelle und grafische Programmflussmodelle präsentieren Shizuki et. al. [23]. Es wird der Datenfluss eines Programms visualisiert und die Möglichkeit geboten, Daten im Fluss nach Auswahl zu untersu-

## 2. Grundlegende Betrachtungen

chen. Mit der Visualisierung von Algorithmen beschäftigt sich auch Jarosław Francik [8]. Er führt höhere Abstraktionsebenen ein, um die Semantik eines Programms zu verdeutlichen. Auch werden Animationen von ihm genutzt.

Modica et. al. [18] erstellen ein Framework für die Animation von Elementen in einem grafischen Editor und stellen ein Beispiel anhand von Petri-Netzen [2] vor.

Auch im Zusammenhang mit dem KIELER-Projekt entstanden einige Arbeiten, die sich mit dem Thema Visualisierung direkt oder indirekt befassen. Das von Stefan Knauer [14] bearbeitete KIELER Environment Visualization (KEV) nutzt Scalable Vector Graphics (SVG) zur Visualisierung einer von einem Modell beschriebenen Umwelt. Nils Beckel [3] befasst sich mit View Management. Dort werden beispielsweise Teile eines Modells, die zu einem bestimmten Zeitpunkt besondere Aufmerksamkeit verdienen, vergrößert oder gefärbt. Es wird hierbei das *focus + context* Prinzip verwendet, das von Card et. al. [4] beschrieben wird. Zwischen View Management und Visualisierung existieren Überschneidungen, da Visualisierung eine Sicht auf ein Modell bietet und somit als Komponente des View Managements angesehen werden kann. View Management betrachtet jedoch zusätzlich noch andere Fragen der Sicht auf ein Modell. Auch Hauke Fuhrmann und Reinhard von Hanxleden [10] befassen sich mit den Problemen, die das Arbeiten mit komplexen Modellen aufgibt. Sie stellen Lösungen vor, wie das bereits erwähnte View Management und automatisches Layout. Mit der Ausführung von Modellen beschäftigt sich Christian Motika [20], unter anderem resultierend in dem KIELER Execution Manager (KIEM).

Neben diesen wissenschaftlichen Arbeiten bietet es sich an, auch einen kurzen Überblick über einige bereits existierende Werkzeuge zu bekommen. Es ist dann möglich, Vorteile dieser Werkzeuge zu übernehmen und Nachteile eventuell zu vermeiden.

## 2.2. Betrachtung existierender Werkzeuge

Es existieren zahlreiche Werkzeuge zum Arbeiten mit grafischen Modellen. Es ist also an dieser Stelle eine Auswahl nötig. Werkzeuge, die Teilprojekten des KIELER-Projekts verwandt sind, sind das Safety Critical Application Development Environment (SCADE) und Ptolemy II. Auch Matlab/Simulink beschäftigt sich unter anderem mit der Entwicklung eingebetteter Echtzeitsysteme und wird deshalb ebenfalls hier betrachtet. Da der komplette Funktionsumfang der Werkzeuge nicht unbedingt in Zusammenhang mit dieser Arbeit steht, wird im folgenden nur jeweils die Visualisierungskomponente der Werkzeuge betrachtet.

### 2.2.1. Ptolemy II

Ptolemy II ist ein von Lee et. al. [6] an der UC Berkeley entwickeltes Werkzeug für Aktor-orientiertes Design [15]. Aktoren sind nebenläufige Softwarekomponenten und kommunizieren über verbundene *Ports*. Die Semantik eines Modells wird in Ptolemy II von einer *Director* genannten Komponente bestimmt.

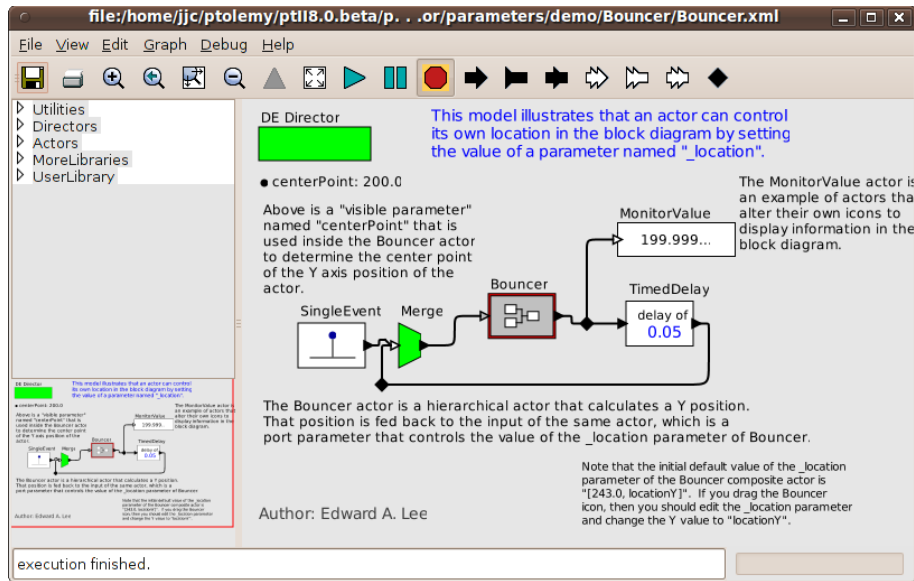


Abbildung 2.1.: Datenvisualisierung in einem Actor

In Ptolemy II existieren Aktoren, die *Sinks* genannt werden. Diese zeichnen sich dadurch aus, dass sie nur eingehende Ports haben. Es gibt Sink-Aktoren, die für die Darstellung eines Datums genutzt werden können. Die Visualisierung von Daten kann somit in Ptolemy II direkt im Modell stattfinden. Ein Beispiel für diese Art der Darstellung ist in Abbildung 2.1 gezeigt.

Eine weitere Art der Datendarstellung in Ptolemy II sind *Scopes* und *Displays*. Dies sind ebenfalls Sink-Aktoren, sie visualisieren aber nicht direkt im Modell. Stattdessen wird für jeden dieser Aktoren ein neues Fenster geöffnet, in dem, wie in Abbildung 2.2 gezeigt, beispielsweise ein Funktionsgraph oder ein Balkendiagramm aufgrund der angefallenen Daten gezeichnet wird.

Die Darstellung von Daten in einem Diagramm mithilfe von Sinks hat den Vorteil, dass die Visualisierung im Modellierungszusammenhang stattfindet. Es kann jedoch ein Nachteil sein, dass für die Visualisierung extra ein neues Element in das Modell eingefügt werden muss. Möglicherweise wäre es auch eine Option, die erstellten Scopes nicht in einem neuen Fenster sondern direkt im Modell, etwa im Sink-Aktor, anzuzeigen. Eine Möglichkeit der Visualisierung, die das Modell nicht verändert, bieten Textdatenfelder. Diese können neben dem Modell arrangiert werden und zeigen die textuelle Darstellung eines Datums.

Ähnliche Visualisierungskonzepte wie Ptolemy II verfolgt Matlab/Simulink.

### 2.2.2. Matlab/Simulink

Simulink [17] ist eine Erweiterung der Entwicklungsumgebung Matlab, die von der Firma Mathworks entwickelt wird.

Simulink dient dazu, dynamische Systeme zu modellieren, zu simulieren und zu

## 2. Grundlegende Betrachtungen

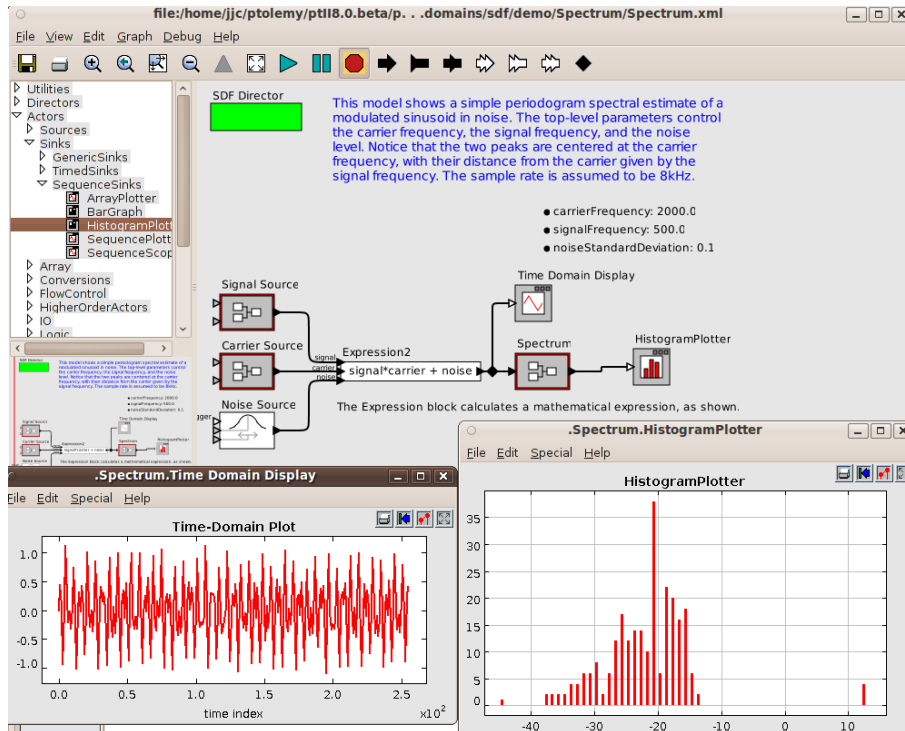


Abbildung 2.2.: Datenvisualisierung in einem neuen Fenster

analysieren. Hier werden die Analysemöglichkeiten, welche im Zusammenhang mit der Visualisierung stehen, betrachtet. Alle in diesem Abschnitt referenzierten Grafiken stammen aus der Simulink Anleitung [17].

Ähnlich wie auch in Ptolemy II kann die Visualisierung in Matlab/Simulink direkt im Modell stattfinden. Dieser Mechanismus wird beispielhaft in Abbildung 2.3 gezeigt. Eine weitere Gemeinsamkeit mit Ptolemy II ist die Nutzung von Sinks, die

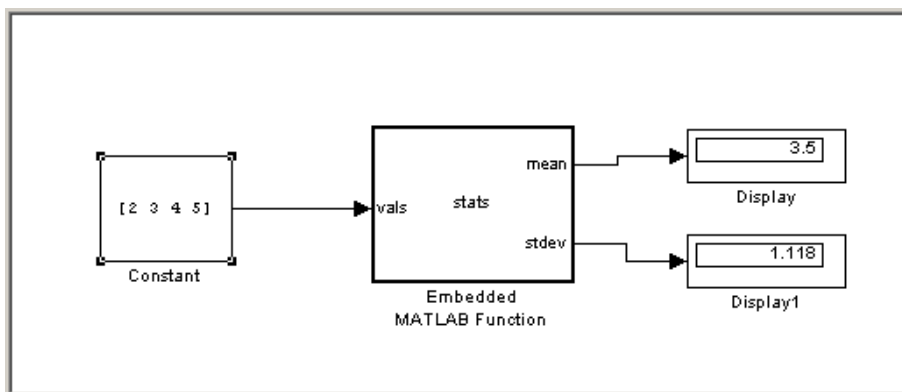


Abbildung 2.3.: Datenvisualisierung im Modell

Scopes in neuen Fenstern öffnen. Die in neuem Fenster geöffneten Funktionsgraphen sind in Abbildung 2.4 dargestellt. Anders als in Ptolemy II können Scopes

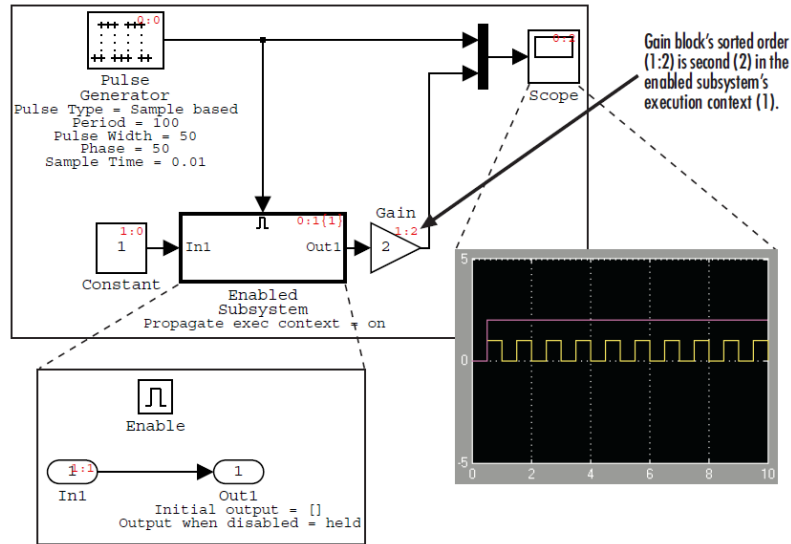


Abbildung 2.4.: Im neuen Fenster geöffneter Graph

aber auch auf eine andere Weise erzeugt werden. Man markiert zu diesem Zweck eine Leitung im Modell als beobachtet, welches in dem Werkzeug durch eine kleine Brille dargestellt wird. Dies wird in Abbildung 2.5 gezeigt. Aufgrund der ähnlichen

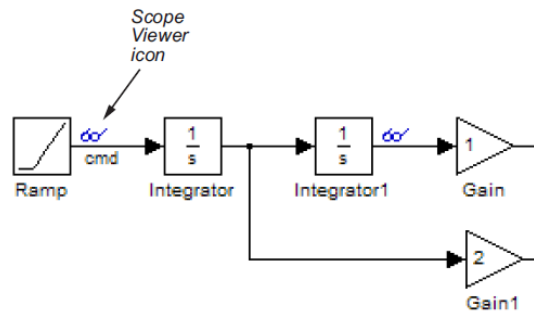


Abbildung 2.5.: Brille als Zeichen für eine Scope

Mechanismen ist auch das Fazit ähnlich wie im Bezug auf Ptolemy II. Der Brillenmechanismus macht es hingegen möglich, Daten zu visualisieren ohne am Modell selbst etwas zu ändern. Dies vereinfacht das Arbeiten für den Benutzer, der nur ein Datum erfahren will, ohne das bearbeitete Modell zu verändern.

### 2.2.3. Safety Critical Application Development Environment

SCADE [7] ist eine Entwicklungsumgebung zur Entwicklung sicherheitskritischer Systeme. Sie basiert auf der Sprache Lustre [12] und kann produktionsreifen C oder Ada Programmcode erzeugen. Es ist mit SCADE möglich, Blockdiagramme zur Modellie-

## 2. Grundlegende Betrachtungen

rung von Datenflusssystemen zu erstellen, oder mit sicheren Zustandsmaschinen zu arbeiten. Auch hier beschränkt sich die Betrachtung auf Möglichkeiten der Visualisierung in SCADE.

Eine Übersicht über die Visualisierungsmöglichkeiten von SCADE bietet Abbildung 2.6. Es existiert dort eine Datentabelle, in der Daten ihren entsprechenden

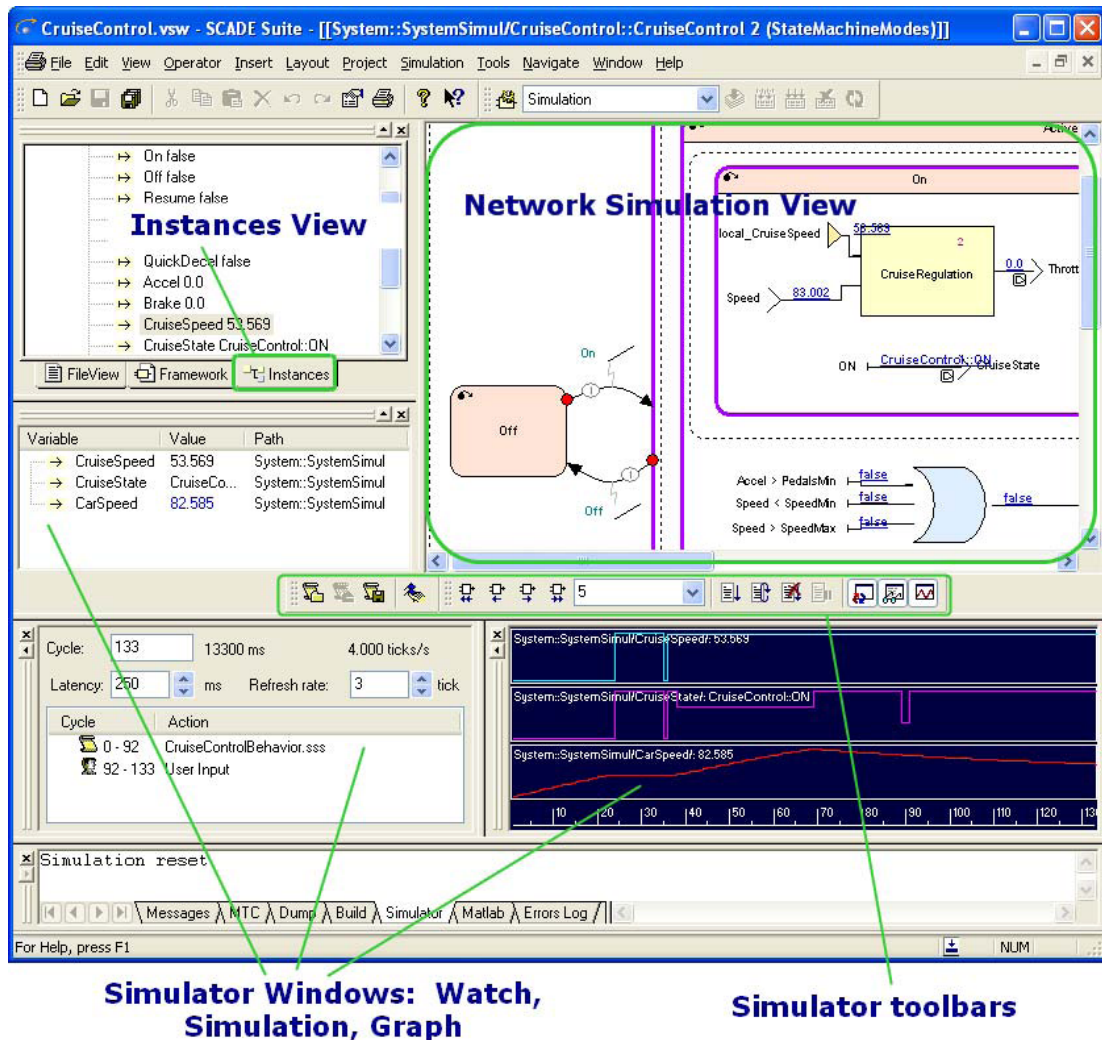


Abbildung 2.6.: Visualisierungsmöglichkeiten nach der SCADE-Anleitung [7]

Modellelementen zugeordnet werden. Funktionsgraphen werden in SCADE nicht in einem neuen Fenster geöffnet, sondern fügen sich in die Entwicklungsumgebung ein. Auch werden Daten direkt im Modell angezeigt, diese erfordern jedoch keine eigenen Modellelemente sondern werden an den Leitungen dargestellt. Diese Vorgehensweise besitzt den Vorteil, dass für die Visualisierung das Modell nicht geändert werden muss. Problematisch kann es jedoch werden, wenn ein Datum eine sehr große Visualisierung besitzt. Dann könnte sie andere Modellelemente kreuzen und die Eindeutigkeit aufheben.



## 2.2. Betrachtung existierender Werkzeuge

Abschließend lässt sich sagen, dass die Arten der Visualisierung in allen vorgestellten Werkzeugen ähnlich sind. Die Unterschiede lassen sich im Arrangement der Visualisierung feststellen. Als einziges Werkzeug folgt SCADE konsequent dem Ansatz, das Modell für die Visualisierung nicht zu verändern.

## 2. *Grundlegende Betrachtungen*

## 3. Verwendete Technologien

In diesem Kapitel werden Technologien vorgestellt, die bei der Entwicklung von KViD zum Einsatz kommen, oder von KViD zur Datenvisualisierung genutzt werden. Hierbei soll jedoch nicht auf technische Details eingegangen werden, sondern vielmehr ein kurzer Überblick über die jeweilige Technologie und die bereitgestellten Funktionen gegeben werden.

### 3.1. Eclipse

Bei Eclipse<sup>1</sup> handelt es sich um eine vielfältige, in Java implementierte Entwicklungsumgebung. Ursprünglich von der amerikanischen Firma IBM entwickelt, ist Eclipse nun ein quelloffenes Projekt, an dem zahlreiche Entwickler aus aller Welt mitarbeiten.

Neben umfassender Unterstützung bei der Java Entwicklung, etwa durch verschiedene Werkzeuge wie *Content-Assist*, einen *Java-Debugger* oder zahlreicher Möglichkeiten zur Code-Navigation, erlaubt Eclipse durch sein Plug-in Konzept nahezu unbegrenzte Erweiterungsmöglichkeiten.

#### 3.1.1. Eclipse Plug-in Mechanismus

Die große Vielfalt von Eclipse ist auf den sogenannten Plug-in Mechanismus<sup>2</sup> zurückzuführen. Über diesen Mechanismus können Entwickler eigene Projekte und Editoren zu ihrer Entwicklungsumgebung hinzufügen oder diese anderen Benutzern zur Verfügung stellen. Einen Überblick über die Struktur des Mechanismus zeigt Abbildung 3.1.

Relevant hierbei sind die Modellierungswerkzeuge Eclipse Modeling Framework (EMF), Graphical Editing Framework (GEF) und Graphical Modeling Framework (GMF). Diese Werkzeuge erlauben es, beliebige grafische Modellierungswerkzeuge mit relativ geringem Aufwand zu erstellen. Für KViD sind diese relevant, weil KViD auf Editoren, die von diesen Werkzeugen erzeugt wurden, seine Visualisierung vornimmt.

---

<sup>1</sup><http://www.eclipse.org>

<sup>2</sup><http://help.eclipse.org/galileo/nav/2>

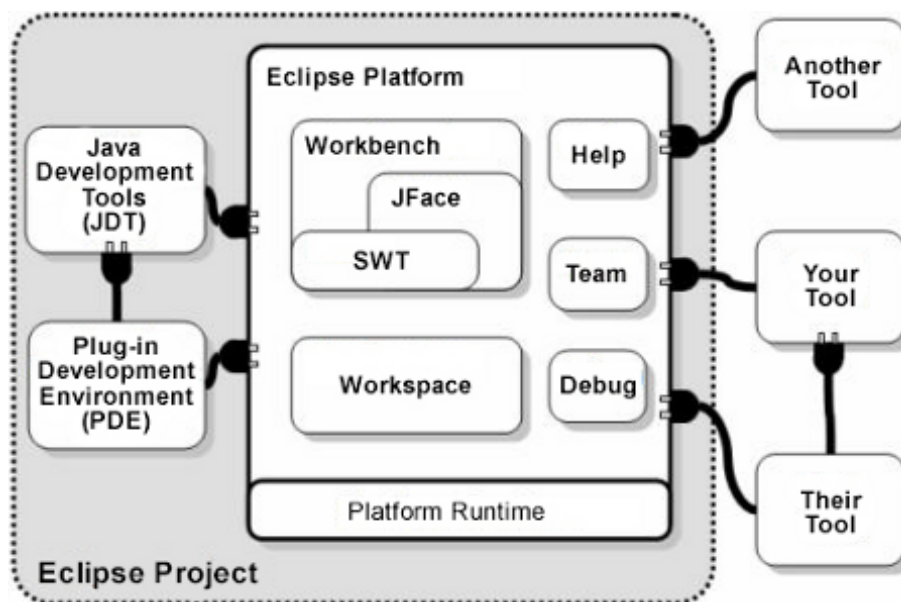


Abbildung 3.1.: Plug-in Architektur wie in der Eclipse-Anleitung<sup>3</sup>dargestellt

#### 3.1.2. Graphical Editing Framework / Graphical Modeling Framework

Das GEF<sup>4</sup> dient dazu, einen grafischen Editor auf Grundlage eines Anwendungsmodells zu erstellen. Es werden Werkzeuge zum Anzeigen von Grafiken oder für das Layout bereitgestellt und der Programmierer wird durch die Unterstützung zahlreicher Standardoperationen entlastet. GEF ist im Model-View-Controller (MVC) [11] Entwurfsmuster implementiert. Eine Aufteilung in *Model*, *View* und *Controller* ist in Abbildung 3.2 dargestellt. Das Diagramm selbst ist das Model, *Figures* bilden die View und *EditParts* sind die Controller-Komponente. GEF ist anwendungsneutral und kann zur Erstellung nahezu aller denkbaren Anwendungen genutzt werden.

GMF<sup>5</sup> wird genutzt, um Editoren mithilfe von EMF<sup>6</sup> und GEF zu erstellen. Der Ansatz ist hierbei, einen Großteil der Grundfunktionalität eines Editors zu generieren, anstatt ihn von Hand zu programmieren. Auch wird eine Trennung von semantischem Modell und grafischem Modell vollzogen. Das semantische Modell enthält nur Informationen über die Bedeutung des Modells, das grafische Modell beschreibt dagegen Aussehen und Ausrichtung der Elemente.

<sup>3</sup><http://help.eclipse.org/galileo/nav/2>

<sup>4</sup><http://www.eclipse.org/gef>

<sup>5</sup><http://www.eclipse.org/gmf>

<sup>6</sup><http://www.eclipse.org/modelling/emf>

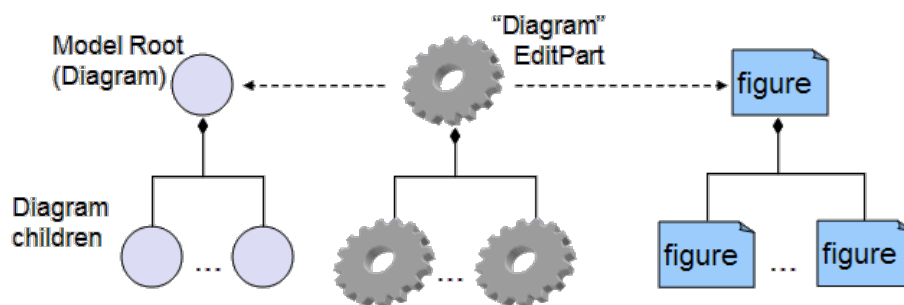


Abbildung 3.2.: Die MVC Struktur von GEF wie in dessen Einführung<sup>7</sup> beschrieben

## 3.2. Kiel Integrated Environment for Layout Eclipse Rich Client

KIELER<sup>8</sup> [10] ist ein Forschungsprojekt, welches sich mit der grafischen Modellierung komplexer Modelle befasst. Es spielt hierbei das Konzept des automatischen Layouts eine große Rolle, welches dem Benutzer die Handhabung großer Modelle deutlich vereinfachen soll.

Des Weiteren wird das Konzept von verschiedenen Sichten auf ein Modell betrachtet, und die Modellierung wird durch strukturbasiertes Editieren und die Möglichkeit, ein Modell zu simulieren, vereinfacht.

### 3.2.1. KIELER Execution Manager

KIEM<sup>9</sup> [20] ist eine Ausführungsinfrastruktur, die konzeptionell jede Art von Ausführung unterstützt. Es bietet unter anderem eine Schnittstelle, um mithilfe des KIELER Werkzeugs domänenspezifische, grafische Modelle, wie etwa GMF-Modelle, simulieren und ausführen zu können. KIEM selber sorgt nicht für die Ausführung oder Visualisierung, sondern verwaltet und synchronisiert Komponenten, die diese Aufgaben übernehmen. Einen Überblick über die Struktur von KIEM bietet Abbildung 3.3.

### 3.2.2. KIELER Infrastructure for Meta Layout

Die Komponente KIELER Infrastructure for Meta Layout (KIML)<sup>10</sup> [22] beschäftigt sich mit dem automatischen Layout von grafischen Modellen. Hierbei wird das Layout eines Diagramms auf abstrakter Ebene spezifiziert, während die konkreten Layout Informationen von Layout-Algorithmen ermittelt werden.

<sup>7</sup><http://www.eclipse.org/gef>

<sup>8</sup><http://www.informatik.uni-kiel.de/rtsys/kieler/>

<sup>9</sup><http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KIEM>

<sup>10</sup><http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KIML>

### 3. Verwendete Technologien

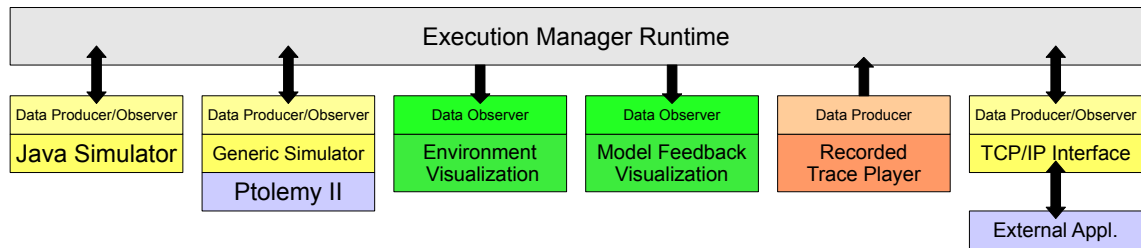


Abbildung 3.3.: Die Infrastruktur von KIEM nach Christian Motika [20]

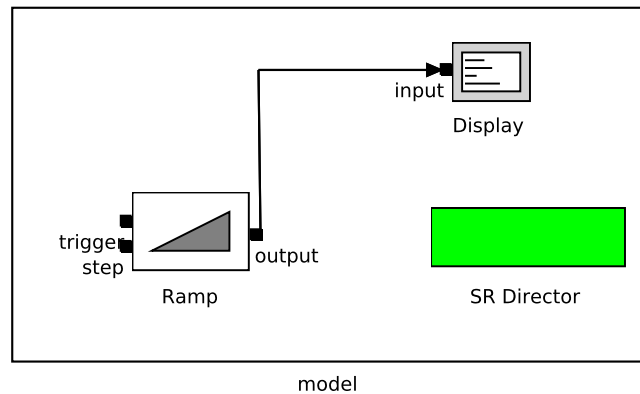


Abbildung 3.4.: Ein mit KAOM geöffnetes Ptolemy II Modell

KIML bietet eine umfassende Schnittstelle, um alle möglichen Arten von Layout-Algorithmen, die auf Graphen arbeiten, anzubinden. Diese lassen sich dann, in der Regel ohne weitere Anpassungen, auf jedes beliebige (GMF-)Modell anwenden.

#### 3.2.3. KIELER Actor Oriented Modeling

KIELER Actor Oriented Modeling (KAOM)<sup>11</sup> ist ein Plug-in, um Aktor-orientierte Modelle, wie sie beispielsweise in Abschnitt 2.2.1 beschrieben werden, zu betrachten und zu bearbeiten. Weiterhin können so für diese Modelle die Vorteile von KIELER genutzt werden, z.B. die Ausführung mit KIEM und automatisches Layout durch KIML. Durch das KIELER Advanced Rendering for Model Appearance (KARMA)-Plug-in<sup>12</sup> ist es möglich, die Modelldiagramme wie im ursprünglichen Editor aussehen zu lassen. Ein mit KAOM geöffnetes Ptolemy II Modell, welches auch als Beispiel in dieser Arbeit verwendet wird, zeigt Abbildung 3.4. Dort gibt es den Aktor *Ramp*, der kontinuierlich einen Zahlenwert inkrementiert und den Aktor *Display*, der den Wert anzeigt. Der *SRDirector* bestimmt die Semantik des Modells.

<sup>11</sup><http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KAOM>

<sup>12</sup><http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/ModelPresentation>

## 3.3. Multi-View Editor Kit

Das Multi-View Editor Kit<sup>13</sup> [18] ist ein umfangreiches Framework, welches die Arbeit mit GEF-basierten Editoren vereinfachen soll und viele verschiedene Funktionen bietet. Unter anderem werden verschachtelte Modelle, verschiedene Sichten auf ein Modell sowie die Animation von Modellverhalten bei der Simulation unterstützt.

Das Kit soll weiterhin dazu dienen, komplexe GEF-Mechanismen zu kapseln und dem Benutzer eine einfachere Schnittstelle zu bieten. Um die Verwendung des Multi-View Editor Kits im Zusammenhang mit KViD zu ermöglichen, mussten kleine Anpassungen vorgenommen werden.

### 3.3.1. Animations-Paket

Besonders interessant ist das Animations-Paket des Multi-View Editor Kits. Dieses vereinfacht nicht nur die Benutzung des GEF-eigenen Animationsmechanismus, es erweitert diesen stark, so dass weitere Effekte bei der Animation möglich werden. Es ist nun beispielsweise möglich, ein grafisches Element während der Animation seine Größe ändern zu lassen, oder statt eines geraden Weges zwischen Start- und Zielpunkt eine Sinuskurve oder eine Ellipse zu beschreiben.

## 3.4. Ptolemy II Plotter

Der Ptolemy II<sup>14</sup> Plotter ist Teil des in Abschnitt 2.2.1 beschriebenen Ptolemy II Werkzeugs. Er bietet umfangreiche Möglichkeiten zum Zeichnen von Funktionsgraphen und Diagrammen und besitzt ein einfaches aber sehr gutes Application Programming Interface (API). Insbesondere die Funktion zum Exportieren der Graphen und Diagramme als Bild ist für eine Weiterverarbeitung nützlich.

## 3.5. Datenformate

Neben den genannten Werkzeugen werden im Zusammenhang von KViD einige Datenformate zur Kommunikation oder zur persistenten Datenhaltung genutzt. Diese werden nun kurz vorgestellt.

### 3.5.1. JavaScript Object Notation

Die JavaScript Object Notation (JSON)<sup>15</sup> ist eine Teilmenge der Programmiersprache *JavaScript* und ein Format, um Daten mit geringem Aufwand austauschen zu können. Ein Vorteil von JSON ist, dass seine Darstellung sowohl menschen- als auch maschinenlesbar ist.

<sup>13</sup><http://tfs.cs.tu-berlin.de/roneditor/>

<sup>14</sup><http://ptolemy.berkeley.edu/ptolemyII/>

<sup>15</sup><http://www.json.org>

### 3.5.2. Comma Separated Values

Eine Comma Separated Values (CSV)-Datei ist eine Textdatei, die eine bestimmte Struktur hat, um Daten persistent zu halten oder zu übertragen. Insbesondere für Tabellen oder Schlüssel-Wert Zuordnungen ist das CSV-Format geeignet. Es existiert jedoch kein verbindlicher Standard zur Formatierung. Es muss daher bei einer Nutzung eine Festlegung auf eine bestimmte Formatierung stattfinden. Listing 3.1 zeigt eine beispielhafte CSV-Datei. Man kann sich den Inhalt als Tabelle mit drei Spalten und vier Zeilen vorstellen, in denen jeweils eine Zahl steht.

Listing 3.1: Beispiel einer CSV-Datei mit Zahlenwerten

```
1 0;1;2
2 2;2;4
3 4;3;6
4 6;4;8
```



## 4. Datenvisualisierung

Im folgenden Kapitel werden Ansätze und Ideen diskutiert, mit deren Hilfe eine Datenvisualisierung im Sinne der Aufgabenstellung ermöglicht werden soll. Es werden dabei unterschiedliche Methoden verwendet, statische Anzeigen, dynamische Anzeigen und auch Weiterverarbeitung, bei der eine grafische Aufbereitung stattfindet.

Einleitend werden Begriffe eingeführt und geklärt, mit denen in diesem Kapitel gearbeitet wird. Danach folgen Beschreibungen und Beispiele für Ansätze, die in die zwei Kategorien flüchtige und persistente Visualisierung unterteilt sind.

### 4.1. Begriffsklärung

**Definition.** *Liegen Informationen in ihrer rohen Maschinendarstellung vor, als Datei auf der Festplatte oder als Ausgabe einer Simulation, so bezeichnet man sie als **Daten**.*

Daten sind also die Grundform jener Informationen, die nun weiterverarbeitet werden müssen. Ein Beispiel für Daten ist die Binärdarstellung einer Zahl. So entspricht etwa **101010** der Zahl **42**. Um von Programmen gelesen und bearbeitet werden zu können, bereitet man die Daten lesbar auf. Dies führt zu einem weiteren Begriff.

**Definition.** *Die programmatisch nutzbare Form von Daten, die in einem Computerprogramm vorliegen, nennt man **Repräsentation** der Daten.*

In einigen Programmiersprachen werden zu der reinen Repräsentation auch Zusatzinformationen vorgehalten, insbesondere der Datentyp. So würde in Java zu dem Datum **101010** als Repräsentation die Zahl **42** und als Datentyp **Integer** vorliegen. Um nun nicht nur von einem Programm, sondern auch von einem Menschen bearbeitet werden zu können, muss eine dritte Ebene eingeführt werden.

**Definition.** *Die grafische, menschenlesbare Darstellung der Repräsentation eines Datums ist die **Visualisierung**.*

Die Visualisierung in unserem Beispiel wären die Ziffern **4** und **2**, die etwa auf der Konsole ausgegeben werden. Zum Zwecke der Vereinfachung wird in der Sprache die Zwischenschicht der Repräsentation oft weggelassen, man spricht von der Visualisierung der Daten. Mit dieser Visualisierung arbeitet der Benutzer eines Computers in der Regel, beispielsweise wenn er einen Text schreibt oder verändert. Eine ähnliche Dreiteilung, wie sie hier vorgestellt wird, verwenden auch Orosco und Moriyón [21].

Im Zuge dieser Arbeit wird bei der Visualisierung zwischen zwei Arten unterschieden, die in ihrer Handhabung grundlegend verschieden sind.

## 4. Datenvisualisierung

**Definition.** *Findet die Visualisierung eines Datums nur für eine kurze Zeit statt, oder werden die visualisierten Elemente nach Ende des Programms verworfen, so liegt eine **flüchtige** Visualisierung vor.*

Diese Art der Visualisierung wird eingesetzt, um den Benutzer über Vorgänge und Vorfälle zu informieren, die zur Laufzeit auftreten. Man wählt sie oft dann, wenn Informationen immer für ein ganzes Programm oder für alle anderen visualisierten Daten angezeigt werden sollen und in der Regel keine speziellen Einstellungen benötigen. Ein einfaches Beispiel ist die Rechtschreibprüfung eines Textprogramms. Zur Laufzeit werden falsch geschriebene Wörter angestrichen, diese Visualisierungsinformationen werden aber nicht gespeichert, sondern stets neu generiert und für das ganze Dokument angezeigt. Dem gegenüber stehen Visualisierungen, die länger Bestand haben.

**Definition.** *Werden Elemente, die nur für die Visualisierung vorhandener Daten genutzt werden, auch nach Ende des Programms noch vorgehalten, so spricht man von **persistenter** Visualisierung.*

Ein einfaches Beispiel hierfür ist ein Diagramm in einer Tabellenkalkulation. Man wählt mehrere Spalten aus, deren Daten dann in einem eingebetteten Diagramm angezeigt werden. Dieses Diagramm ist auch nach Neustart des Programms noch vorhanden.

### 4.2. Flüchtige Visualisierung

Eine sehr einfache Möglichkeit der Datendarstellung ist Text. Die meisten anfallenden Daten haben eine textuelle Visualisierung, die man auch für die Anzeige wählen kann. Dies gilt insbesondere für den Anwendungsfall der Datenflussmodelle, in denen oft Zahlen als Daten anfallen.

Im Zusammenhang mit der grafischen Modellierung reicht die einfache textuelle Darstellung oft nicht aus. In größeren oder komplexeren Modellen entsteht schnell eine Fülle an Daten, deren Visualisierung als reiner Text einen Benutzer überfordern kann, insbesondere dann, wenn der Zusammenhang zwischen Visualisierung und Modell nicht besteht oder unklar ist. Für ein bequemes und leicht verständliches Arbeiten ist es notwendig, die Visualisierung der Daten grafisch in den korrekten Zusammenhang zu bringen. Dies kann in der konkreten Anwendung bedeuten, dass eine Darstellung in der Nähe der Datenquelle erfolgt oder dass der Text einer bestimmten Leitung zugeordnet wird. Des Weiteren ist es nicht unwichtig, die Daten in einer ansprechenden Weise anzuzeigen. So soll, statt dem Ausgeben von einfachem Text, die Visualisierung mithilfe eines grafischen Objekts stattfinden, so dass sie sich in den Zusammenhang der Modellierung einfügt.

Um dies zu erreichen, wird die textuelle Visualisierung des Datums durch eine grafische Figur umschlossen. Eine solche Figur, mit deren Hilfe der Zusammenhang der Modellierung erhalten werden soll, wird im Folgenden als **Marke** bezeichnet.



Abbildung 4.1.: Darstellung eines Datums mithilfe einer Marke

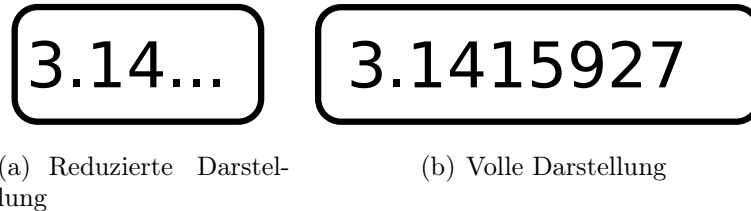


Abbildung 4.2.: Behandlung von langen Visualisierungen

Es wird zu diesem Zweck ein Datum, bzw. dessen Repräsentation, aus einer Datenquelle geladen. Dann wird die textuelle Visualisierung erzeugt, aber zusätzlich ein Rahmen um diese gezogen, um die Visualisierung besser in eine grafische Modellierungsumgebung einzufügen. Eine solche Marke wird in Abbildung 4.1 gezeigt. Auch andere Werkzeuge binden Visualisierung in den Modellierungszusammenhang ein, wie bereits in Abschnitt 2.2.1 und Abschnitt 2.2.2 dargelegt wurde. Dort sind die visualisierenden Elemente aber Teil des Modells und haben semantische Bedeutung. Ziel der flüchtigen Visualisierung soll es sein, die Visualisierung ohne Veränderung am Modell zu ermöglichen.

Ein weiteres Problem tritt in bereits existierenden Werkzeugen auf, wenn die textuelle Darstellung sehr lang ist, sei es aufgrund ihrer Komplexität, der Menge oder einfach der bloßen Größe einer Zahl. Zeigt man hier, wie beispielsweise in Abschnitt 2.2.3 beschrieben, die volle textuelle Visualisierung eines Datums in der Nähe einer Leitung, so kann die Visualisierung sich mit anderen Elementen überschneiden und eine eindeutige Zuordnung erschweren. Um dies zu vermeiden muss entschieden werden, wie eine vereinfachte Darstellung in einer Marke stattfinden kann und auf welche Weise Zugriff auf die vollständige Darstellung erfolgen kann.

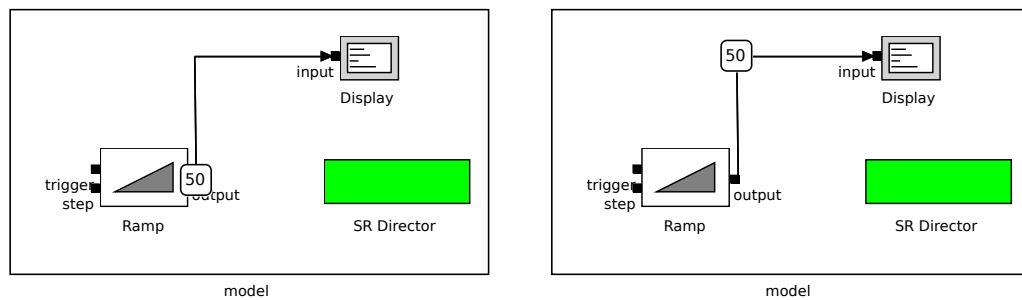
Einen guten Kompromiss erhält man, wenn man einen Teil der Daten oder ein Zeichen, das für die Art der Daten steht, in der Marke anzeigt und die Möglichkeit bietet, sich die gesamten Daten auf Anfrage anzeigen zu lassen. Ein Beispiel für beide Teildarstellungen zeigt Abbildung 4.2.

Wie einführend erwähnt, ist es sehr wichtig, in einem grafischen Modell die Daten in einen konkreten Zusammenhang mit dem Modell zu bringen. Eine Marke soll sich nicht nur in den Modellierungszusammenhang einfügen, sie muss auch an der richtigen Stelle platziert werden. Für die Wahl der Platzierung gibt es mehrere Möglichkeiten.

### 4.2.1. Anzeigemöglichkeiten für Marken

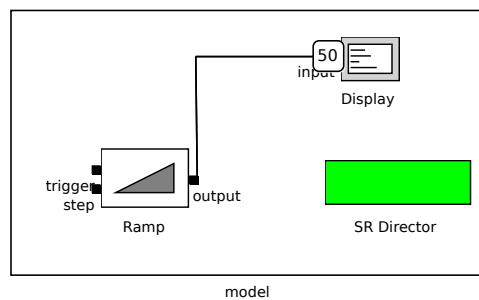
Die einzelnen Optionen zur Darstellung sind in Abbildung 4.3 zu sehen. Das gezeigte Beispielmmodell ist in Abschnitt 3.2.3 erklärt. Die Platzierung am erzeugenden Knoten hilft dabei festzustellen, wer welche Daten generiert. Um die Daten zu erfahren, die zu einem bestimmten Ergebnis führen, platziert man die Marken am Zielknoten. Für eine allgemeine Übersicht lassen sich die Daten auch auf der Verbindungslinie darstellen. Diese Möglichkeiten der Darstellung sollten für jedes Datum individuell wählbar sein. Es ergeben sich so einige Kombinationsmöglichkeiten, die bei der Analyse hilfreich sein können.

- Um einen allgemeinen Überblick über den aktuellen Zustand zu bekommen, platziert man alle Marken auf den Verbindungslinien.
- Um eine Berechnung nachzuvollziehen, platziert man die anliegenden Werte am Zielknoten und das Ergebnis am erzeugenden Knoten. Man hat so alle relevanten Werte beieinander.
- Um festzustellen, ob ein Knoten möglicherweise falsche Werte liefert, platziert man eine Marke am erzeugenden Knoten und überwacht die Werte.



(a) Am erzeugenden Knoten

(b) Auf der Verbindungslinie



(c) Am Zielknoten

Abbildung 4.3.: Verschiedene Möglichkeiten der Platzierung

Zusätzlich zu dieser statischen Darstellung ist es auch denkbar, die Visualisierung durch dynamische Elemente wie die Animation anzureichern.

### 4.2.2. Animierte Marken

Wenn weitere Informationen aus den Daten bzw. deren Repräsentation gewonnen werden können, eröffnen sich auch Möglichkeiten, die Visualisierung umfang- und informationsreicher zu gestalten. Lassen sich etwa Informationen darüber gewinnen, welche Verbindungslinie des Modells durch das Auftreten des Datums genutzt wird, so kann man die Marke des Datums in einer Animation dieser Linie folgen lassen. Diese Animation kann insbesondere die Ausführung eines Modells dynamischer wirken lassen. Auch wird durch die Animation die Aufmerksamkeit des Benutzers auf Teile des Modells gelenkt, in denen Veränderungen auftreten und Daten anfallen. Dies kann bei der Analyse und Fehlersuche hilfreich sein.

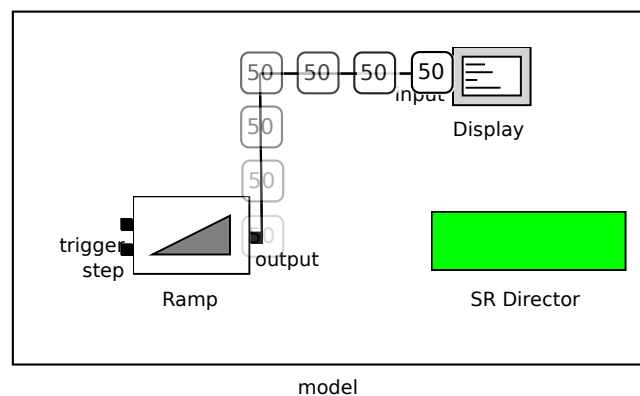


Abbildung 4.4.: Animierte Marke

Ein möglicher Anwendungsfall hierfür findet sich in Datenflussmodellen. Dort fließen oft Daten über die Verbindungslinien. Es bietet sich also an, nicht nur das Datum sondern auch dessen Fluss zu visualisieren. Ein Beispiel hierfür wird in Abbildung 4.4 gezeigt.

Eine solche animierte Visualisierung lässt sich auch in Modellen nutzen, in denen über die Verbindungslinien keine Daten fließen. Hierzu muss aus den Daten bzw. deren Repräsentation lediglich die Information über das Nutzen der Verbindungslinien gewonnen werden.

Das in Abbildung 4.5 dargestellte Beispiel stellt das Nehmen einer Transition in SyncCharts [1] dar. Die Abbildung ist mithilfe des SyncCharts-Editors des KIELER Werkzeugs entstanden. Die animierte Marke verdeutlicht, welche Transition genommen wird und welches der Nachfolgezustand ist. Allgemein lässt sich so der Fluss in Modelltypen darstellen, in denen Zustandsübergänge mithilfe von Verbindungslinien stattfinden. Ein ähnlicher Mechanismus zum Hervorheben von relevanten Modell-elementen wird auch von Nils Beckel [3] beschrieben. Dieser nutzt jedoch Färbung vorhandener Elemente zur Hervorhebung anstelle von Animation.

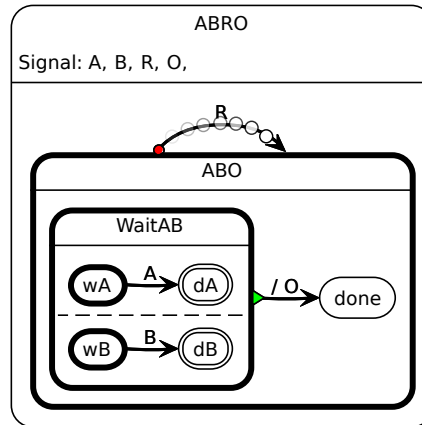


Abbildung 4.5.: Animierte Marke ohne textuelle Visualisierung

### 4.2.3. Nutzung vorhandener Elemente

Der Ansatz der Färbung bereits vorhandener Elemente lässt sich auch im Zusammenhang der Datenvisualisierung nutzen. Neben der Visualisierung der Informationen über das Nutzen von Verbindungslinien lassen sich auch anfallende oder relevante Daten durch Färbung hervorheben, wenn diese bereits eine Darstellung im Modell haben. Dies wird am Beispiel von SyncCharts in Abbildung 4.6 dargestellt. Da in

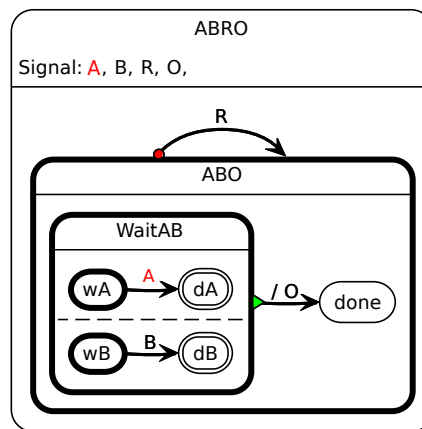


Abbildung 4.6.: Hervorhebung durch Färbung

Modellen mit bedingten Transitionen oft nur zwischen erfüllter und nicht erfüllter Bedingung unterschieden wird, kann eine erfüllte Bedingung durch rote Färbung der ansonsten schwarz dargestellten Bedingung verdeutlicht werden. Eine nicht erfüllte Bedingung wird dann schwarz belassen. Bei komplexeren Bedingungen, wie etwa Teilbedingungen die durch logische Operatoren verbunden werden, ist es auch denkbar, die erfüllten Teile einer Bedingung zu färben.

## 4.3. Persistente Visualisierung

Im Gegensatz zu den Konzepten der flüchtigen Visualisierung steht die persistente Visualisierung. Die Visualisierungselemente dieser Kategorie werden in das Modelldiagramm eingefügt und haben auch nach Programmende bestand. Wichtig ist hierbei aber, dass diese Elemente weiterhin keinerlei Einfluss auf die Semantik des Modells haben.

Im Folgenden werden Elemente betrachtet, deren Einbettung in das Modelldiagramm das Arbeiten erleichtern kann. Zunächst wird auf die Einbettung von Diagrammen eingegangen, danach folgen Überlegungen, welche Art von Grafiken nutzbar sind, um anfallende Daten zu visualisieren.

### 4.3.1. Einbetten von Diagrammen

Neben der einfachen Visualisierung eines Datums als Text oder in einer Marke gekapseltem Text gibt es zahlreiche weitere, auch grafisch umfangreichere Methoden der Darstellung. Denkt man allein an die in Massenmedien oft zum Einsatz kommenden Diagrammtypen wie Torten- oder Balkendiagramme, so sind schnell zahlreiche Darstellungsmöglichkeiten gefunden.

Oft ist es üblich, die aus Daten generierten Diagramme in einer zusätzlichen Ansicht, etwa einem neuen Fenster, anzuzeigen. In Abschnitt 2.2 wurde bereits festgestellt, dass viele existierende Werkzeuge diesen Weg gehen.

Eine andere Möglichkeit wäre es, auch die Diagramme in die Modellierung direkt einzubinden. Hierzu bietet es sich an, die erstellten Diagramme in das Modelldiagramm einzubetten, anstatt sie in einem externen Fenster anzuzeigen. Um den Bezug zwischen eingebettetem Diagramm und Modellelement klar zu machen, sollte eine Verbindungslinie zwischen dem betreffenden Element und dem Diagramm gezogen werden. Hierbei ist jedoch darauf zu achten, dass sich diese Verbindungslinie von allen anderen Verbindungslinien, die in dem Modell eine semantische Bedeutung haben, unterscheidet. Ein Beispiel für ein in ein Modelldiagramm eingebundenes

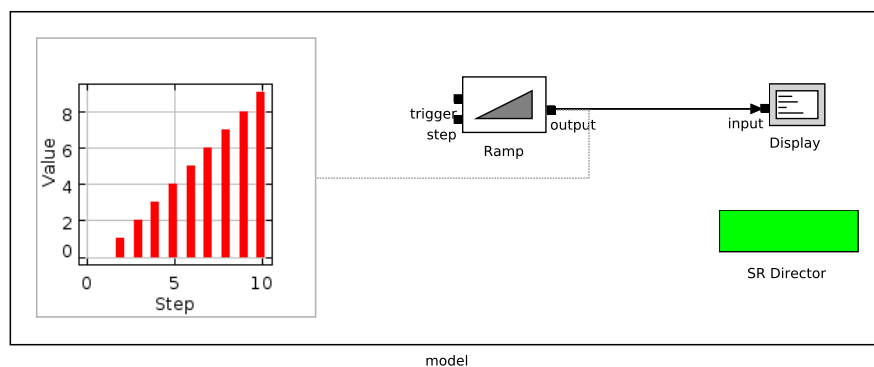


Abbildung 4.7.: In das Modelldiagramm eingebundener Ptolemy II Plotter

#### 4. Datenvisualisierung

Balkendiagramm zeigt Abbildung 4.7. Das gezeigte Balkendiagramm wurde mithilfe des Diagramm-Zeichentools des Ptolemy II Werkzeugs, welches in Abschnitt 2.2.1 vorgestellt wurde, erstellt.

Es soll nicht Ziel sein, alle denkbaren Möglichkeiten von verschiedenen Diagrammen anzubieten. Vielmehr soll eine generische Grundlage für das Einbetten von Diagrammen in grafische Modellierungswerkzeuge geschaffen werden. In Abbildung 4.8 wird dargestellt, wie diese Grundlage mit den anderen Bestandteilen zusammenhängt. Der Datenverteiler erhält Daten von einer Schnittstelle oder einer Ausführungsumgebung. Es werden dann die Repräsentationen und Metainformationen wie Platzierungen gewonnen. Dies ist die Grundlage, bei der sich neue Visualisierungskomponenten registrieren. Diese Komponenten erhalten dann zur weiteren Verwendung die gesammelten Informationen. So kann man dann auch von anderen Funktionen von KIELER profitieren, etwa das automatische Layout für eine vereinfachte Handhabung, insbesondere bei großen Modellen. Hat man eine solche Grundlage geschaffen, ist das Erweitern um neue Diagrammtypen unproblematisch.

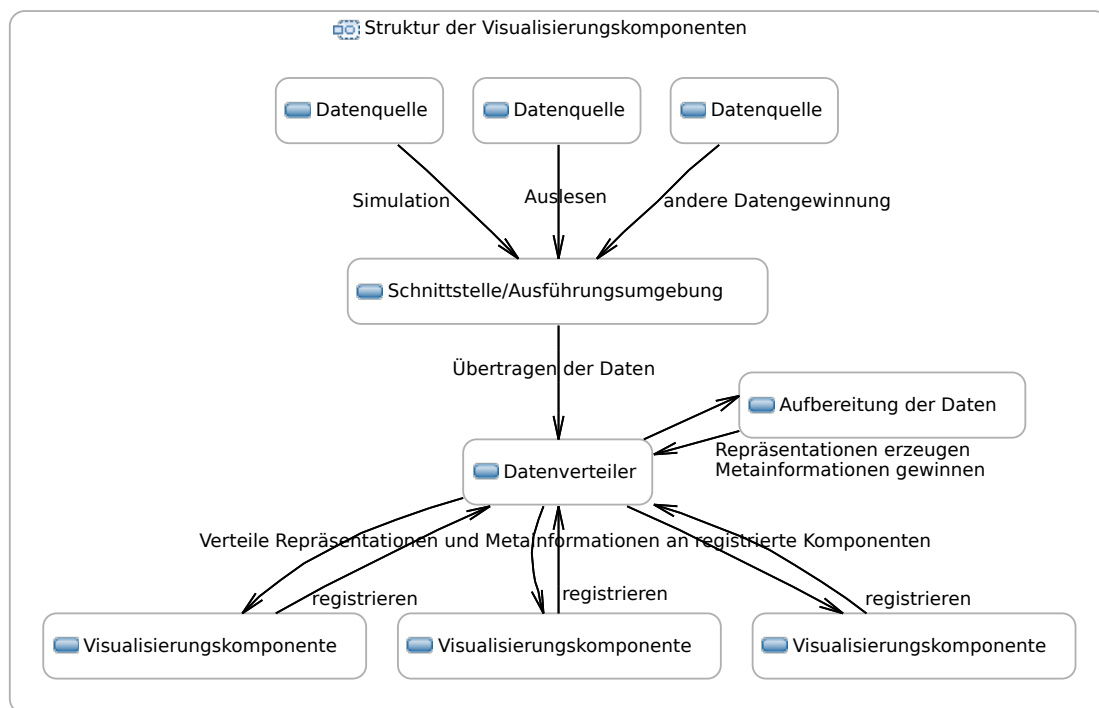


Abbildung 4.8.: Struktur einer Grundlage für Visualisierungskomponenten

Zur Veranschaulichung dieser Funktion soll das Werkzeug zur Erstellungen von Diagrammen und Funktionsgraphen aus Ptolemy II benutzt werden, um Diagramme für die Einbettung zu erstellen.

Eine einmal erstellte Möglichkeit zur Einbettung von Visualisierungen beschränkt sich nicht nur auf Diagramme. Auch allgemeine Grafiken lassen sich dann zum Zwecke der Visualisierung nutzen.



### 4.3.2. Einbetten von Grafiken

Grafiken allgemeiner Natur lassen sich, oft abhängig vom konkreten Modell, zur Visualisierung von Modellverhalten nutzen. Ein Projekt, das ebenfalls Teil des KIELER Werkzeugs ist, ist KEV. Dieses von Stephan Knauer [14] bearbeitete Projekt nutzt animierte Grafiken, um die Daten, die im Zuge einer Modellausführung anfallen, in einen realitätsnahen Zusammenhang zu bringen. Die entsprechenden Grafiken werden auch hier in einem externen Fenster angezeigt. Hier ist es erneut möglich, diese Grafiken direkter in den Modellierungszusammenhang einzubinden, wenn man die Grafiken in das Modelldiagramm einbettet. So ist es möglich, anfallende Daten verständlicher zu visualisieren, indem man ein alltägliches Objekt als animierte Grafik in das Modelldiagramm einbettet. Abbildung 4.9 zeigt beispielsweise ein eingebettetes Thermometer, welches zur Visualisierung der anfallenden Daten genutzt wird.

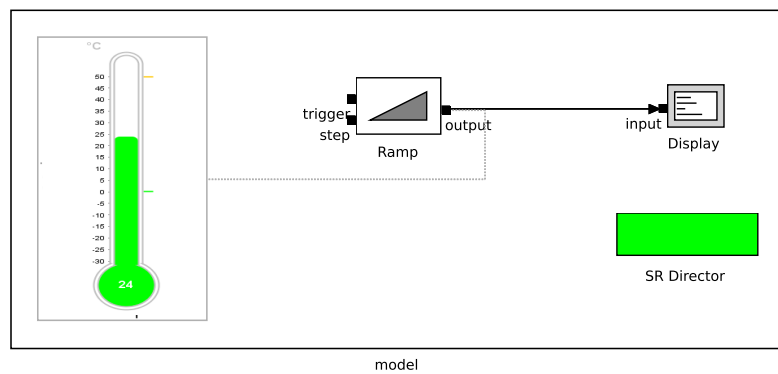


Abbildung 4.9.: In das Modelldiagramm eingebundene Grafik

#### 4. Datenvisualisierung

## 5. Implementierung

KViD ist als Teil des KIELER-Projekts implementiert und dort als Eclipse Plug-in, wie in Abschnitt 3.1.1 beschrieben, integriert. Das KIELER-Projekt ist nach dem MVC Entwurfsmuster strukturiert, wie es unter anderen von Gamma et. al. [11] erwähnt wird. Eine Übersicht über die Komponenten und deren Einordnung in KIELER zeigt Abbildung 5.1. Wie aus der Abbildung hervorgeht, ist KViD als Teil der View-

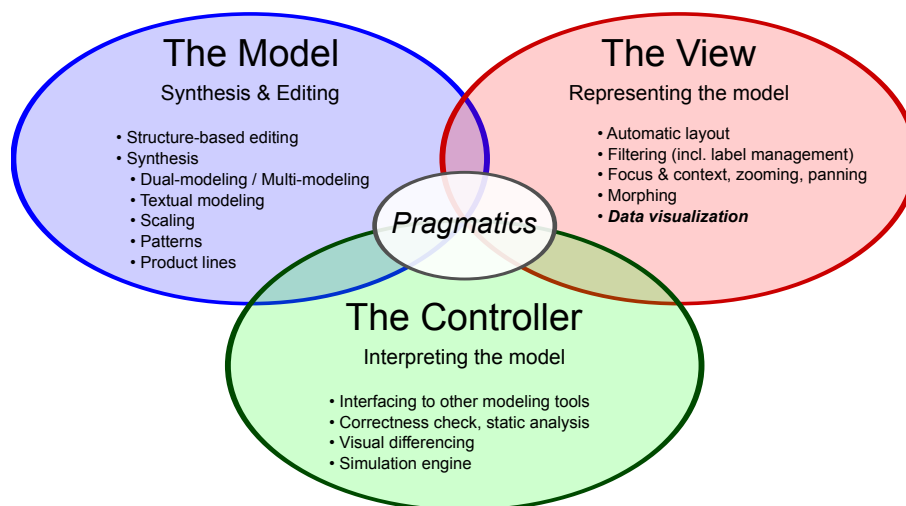


Abbildung 5.1.: Die MVC Struktur des KIELER-Projekts nach Fuhrmann und von Hanxleden [9]

Komponente anzusehen. Trotz dieser Einstufung verwendet KViD intern ebenfalls eine MVC Struktur für die Erfüllung seiner Aufgaben. Somit sind Datenhaltung und Visualisierungskomponenten so modularisiert, dass sie bei Bedarf ausgetauscht werden können. Im folgenden Kapitel wird zunächst die allgemeine Struktur der Implementierung sowie die Zuteilung der Pakete in die Bereiche *Model*, *View* und *Controller* dargelegt. Es folgt eine Beschreibung der verwendeten Schemata zur Adressierung von Modellelementen. Dann werden die einzelnen Pakete besprochen und ihre Aufgaben innerhalb der Implementierung erklärt. An ausgewählten Stellen wird hierbei ins Detail gegangen, um bestimmte Mechanismen und Vorgehensweisen zu erläutern.

## 5.1. Überblick

Die Implementierung folgt der Idee, eine generische Grundlage für die Visualisierung zu schaffen. Hierzu wird eine Struktur gewählt, wie sie auch von Orosco und Moriyón [21] verwendet wird. Die Repräsentationen der zu visualisierenden Daten werden in das Programm geladen. Es wird dabei versucht, möglichst viele Metainformationen zu sammeln, die bei der Visualisierung erweiterte Möglichkeiten eröffnen. Dies wäre beispielsweise der Typ der Daten, etwa ganze Zahlen oder ein Wahrheitswert. Eine zentrale Komponente übernimmt dann die Verteilung der Repräsentationen an grafische Komponenten, die dann die Visualisierung erstellen. Diese zentrale Komponente verknüpft auch die Repräsentationen mit dem Modell-

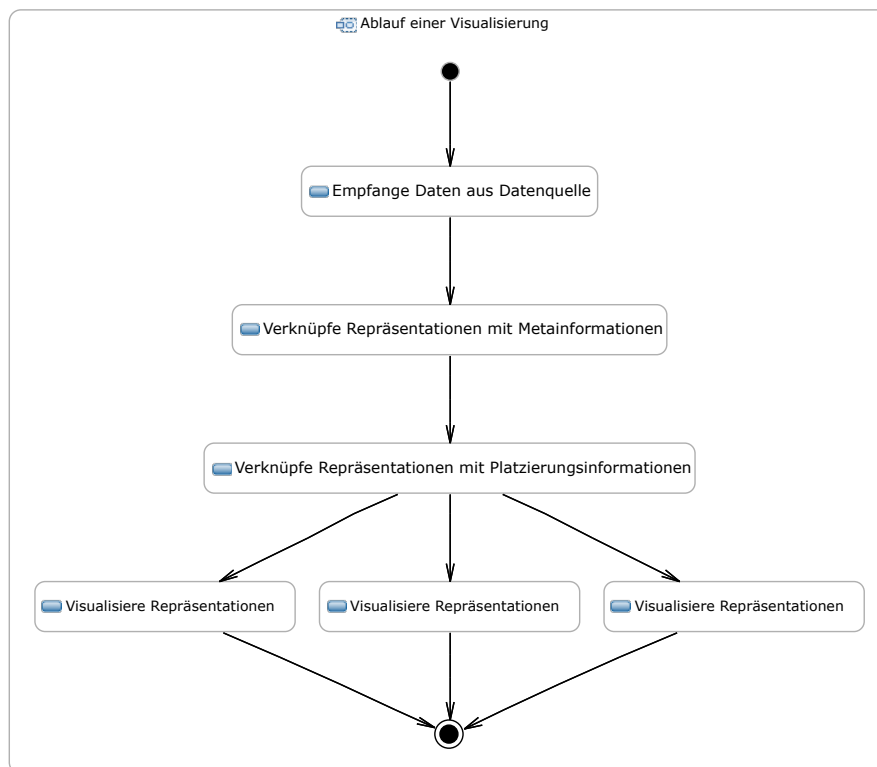


Abbildung 5.2.: Ablauf einer Visualisierung

diagramm, in dem es Informationen über die Platzierung der Modellelemente mit den Repräsentationen verknüpft. Nachdem alle Informationen zusammengetragen wurden, benachrichtigt die zentrale Komponente alle grafischen Komponenten, die dann die eigentliche Visualisierung nach ihren eigenen Spezifikationen im Modelldiagramm vornehmen. Dieser Ablauf ist in Abbildung 5.2 schematisch dargestellt.

Die MVC Struktur von KViD wird auch durch die Einteilung in die Pakete deutlich. Als *Model* sind die Elemente des `data`-Pakets zu verstehen, hier werden die Repräsentationen von Daten für die Visualisierung vorgehalten. Auch die Klassen des

`dataprovider`-Pakets sind dem zuzuordnen, da sie die Schnittstelle zu verschiedenen Datenquellen bereitstellen.

*Controller* sind die Klassen des `datadistributor`-Pakets. Sie sorgen für die Verteilung der Daten an die richtigen Anzeigen und liefern Metadaten, die etwa für eine Animation vonnöten sind.

*View*-Komponenten sind dann in den Paketen `visual` und `visual.complex` sowie in den `kvid.ui` Paketen gruppiert. Die Klassen dieser Pakete sind direkt für das Zeichnen und Animieren der Daten verantwortlich.

Eine Übersicht über die Klassen und Pakete von KViD wird in Abbildung 5.3 gegeben. Wie in Abschnitt 4.2 beschrieben ist es zentral für das Visualisieren direkt im

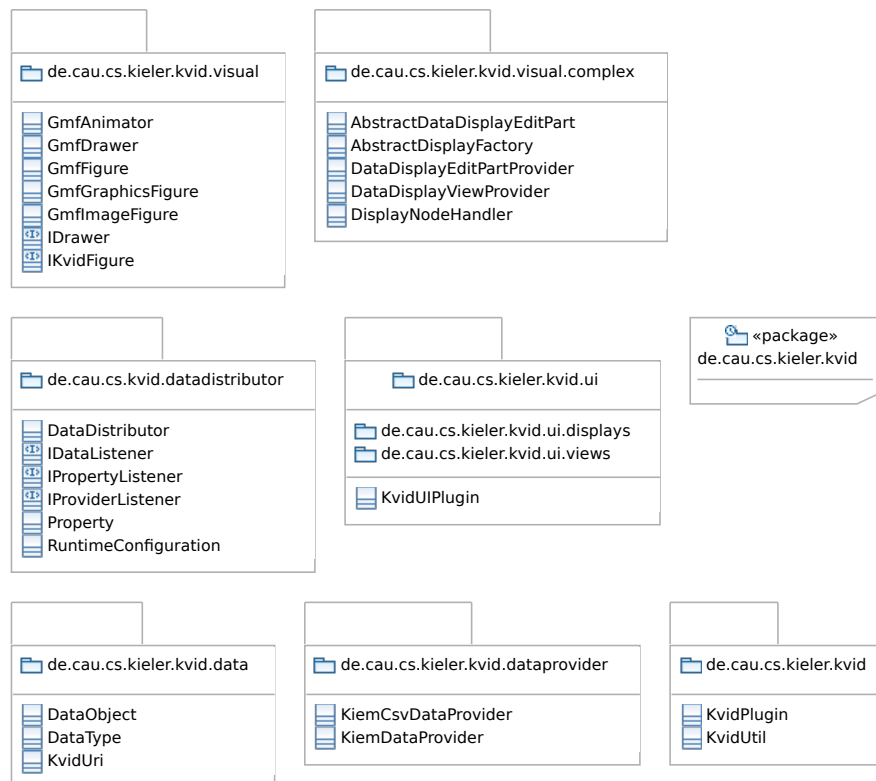


Abbildung 5.3.: Pakete des KViD-Plug-ins

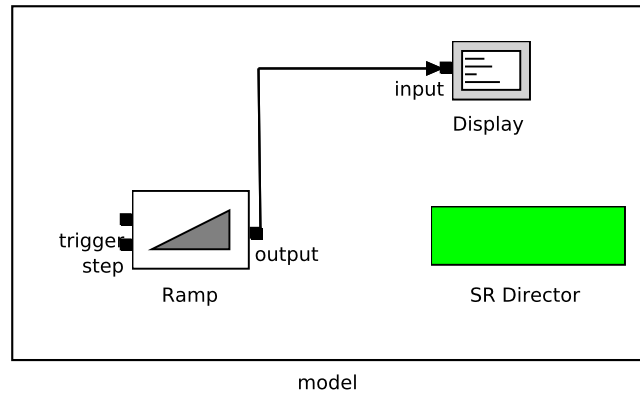
Modelldiagramm, eine Verbindung zwischen Visualisierung und den Modellelementen zu schaffen. Zu diesem Zweck muss der Zusammenhang zwischen Datum und Modell maschinenlesbar vorliegen.

## 5.2. Adressierung von Modellelementen

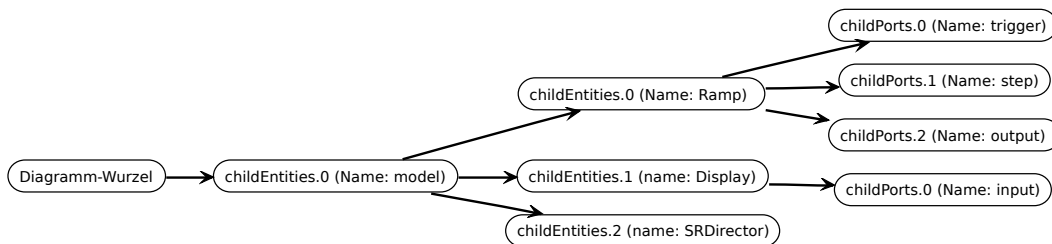
Um eine Visualisierung dem richtigen Modellelement zuzuordnen, muss das Datum mit einer Referenz auf das Element verknüpft werden. Diese Referenz, auch Uniform Resource Identifier (URI) genannt, wird von der Datenquelle zur Verfügung gestellt.

## 5. Implementierung

Sie kann in zwei verschiedenen Adressformaten vorliegen, welche sich an der Struktur des Modells orientieren. Die Hierarchie der von KViD unterstützten Modelle lässt sich stets durch eine Baumstruktur darstellen. In Abbildung 5.4(a) ist ein grafisches Modell dargestellt, Abbildung 5.4(b) zeigt den dazugehörigen Baum. Die zwei



(a) Ein einfaches grafisches Modell



(b) Die Hierarchie als Baumstruktur

Abbildung 5.4.: Modell und Baumdarstellung

Adressformate, die von KViD benutzt werden, unterscheiden sich in ihrem Orientierungspunkt. Die sogenannte *Fragment URI* wird bereits in EMF<sup>1</sup> genutzt und bezieht sich nur auf die Baumstruktur des Modells. Die *Namen-basierte URI* bezieht sich auf die Namen der adressierten Elemente und ist leichter für einen Menschen lesbar. Sie besitzt jedoch den Nachteil, dass alle Elemente einer Hierarchieebene einen eindeutigen Namen besitzen müssen, damit man sie nutzen kann.

**Fragment URI:** Eine Fragment URI hat das Format  $\langle \text{Wurzel}, \text{ leer wenn es nur eine Wurzel gibt} \rangle \langle \text{Kindelement der Wurzel} \rangle \langle \text{Kindelement des vorherigen Elements} \rangle$  etc.

**Namen-basierte URI:** Eine Namen-basierte URI hat das Format  $\langle \text{Wurzelnamen}, \text{ leer wenn es nur eine Wurzel gibt} \rangle \langle \text{Name eines Kindelements der Wurzel} \rangle \langle \text{Name eines Kindelements des vorherigen Elements} \rangle$  etc. Abbildung 5.5 zeigt Beispieladressen für das Modellelement „Ramp“ aus dem in Abbildung 5.4 gezeigten Modell.

<sup>1</sup><http://www.eclipse.org/modelling/emf>

EMF Fragment URI: //childEntities.0/childEntities.0
Namen-basierte URI: .model.Ramp

Abbildung 5.5.: Beispieladressen für das Modellelement „Ramp“

Bei der Namen-basierten URI stellen Verbindungsstellen, sogenannte *Ports*, einen Sonderfall dar. Um diese von Knoten unterscheiden zu können, wird der Name des Ports durch einen Doppelpunkt getrennt an die Adresse ihres zugehörigen Modellelements angefügt. Die Fragment URI behandelt Ports hingegen wie alle anderen Elemente.

Mithilfe einer Methode der `KvidUtil` Klasse lässt sich eine Fragment URI auch in eine Namen-basierte URI übersetzen, sofern alle Namen einer Hierarchieebene eindeutig sind. Eine Übersetzung von einer Namen-basierten URI in eine Fragment URI ist möglich, aber für `KViD` nicht nötig und deshalb nicht implementiert.

Zur besseren programmatischen Handhabung der URIs kann aus einer textuellen Adresse ein `KvidUri` Objekt erstellt werden. Dieses kapselt die URI und bietet Methoden, um direkt auf einzelne Bestandteile wie den Port zugreifen zu können.

Um nun die adressierten Daten zu Visualisieren, muss eine zentrale Stelle die Daten aus der Quelle empfangen, und sie den richtigen Empfängern zuordnen.

## 5.3. Datenverwaltung

Die Datenverwaltung, und hier insbesondere der `DataDistributor`, ist das Herzstück von `KViD`. Der `DataDistributor` ist nach dem *Singleton*-Entwurfsmuster implementiert, wie es von Gamma et. al. [11] eingeführt wird, und ist nach einem Drei-Stufen-System aufgebaut. Dieses resultiert aus dem implementierten Inter-

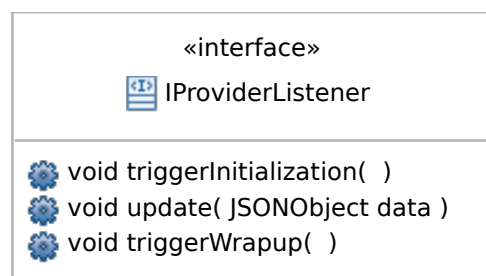


Abbildung 5.6.: Interface eines Datenquellen-Beobachters

face, dem `IProviderListener`, welches in Abbildung 5.6 gezeigt wird. Das Interface wird als Teil des *Observer*-Entwurfsmuster eingesetzt, welches ebenfalls von Gamma et. al. [11] beschrieben wird. Der `DataDistributor` ist der Beobachter, der eine Datenquelle beobachtet. Beginnt nun eine Quelle damit, Daten zu liefern, so wird

## 5. Implementierung

die Methode `triggerInitialization()` aufgerufen. Dies aktiviert die `initialize()`-Methode des `DataDistributors`. Hier werden nötige Informationen geladen, etwa die Platzierungsinformationen zu einem Modell.

Die Methode `update()` wird jedes Mal aufgerufen, wenn die Datenquelle einen neuen Satz Daten liefert. Diese werden in ein `JSONObject`, wie es in Abschnitt 3.5.1 beschrieben wird, verpackt, sortiert nach einem Schlüssel-Wert-Schema, bei dem die Adresse eines Modellelements den Schlüssel bildet. Der zugehörige Wert ist dann das entsprechende Datum bzw. dessen Repräsentation. In der `update()`-Methode werden dann die Daten an die visualisierenden Objekte verteilt, und auch die entsprechenden Schritte zur Visualisierung, wie etwa das Zeichnen der grafischen Objekte, wird angestoßen. Auch werden in diesem Schritt Beobachter, die das `IDataListener`-Interface implementieren, benachrichtigt, dass sich die Daten geändert haben. Das `IDataListener`-Interface ist dem `IProviderListener`-Interface sehr ähnlich, lediglich der Schritt der Initialisierung fehlt hier, da sich diese Beobachter bereits vor einer Visualisierung beim Laden des Programms selbst initialisieren. Auf welche Art auf Änderungen reagiert wird obliegt dann dem jeweiligen Beobachter.

In der letzten der drei Phasen, ausgelöst durch ein `triggerWrapup()` der beobachteten Datenquelle, werden schließlich vorgehaltene Daten gelöscht. Auch Verweise auf ein bestimmtes Modelldiagramm und zwischengespeicherte Informationen werden zurückgesetzt. Der `DataDistributor` ist dann bereit, einen neuen Visualisierungsvorgang zu beginnen.

Einige Hilfsfunktionen werden im Hauptpaket von KViD von der Klasse `KvidUtil` zur Verfügung gestellt.

Für die Animation von besonderer Bedeutung ist dort die Methode `getPathsByElement()`. Diese Methode liefert zu einem Knoten des Modells alle Pfade, die ein dort generiertes Datum bei der Animation nehmen kann. Die Bestimmung dieser Pfade findet anhand des Layouts statt, welches von KIML zur Verfügung gestellt wird.

Die Methode `getActiveEditor()` vereinfacht es, auf den aktuell aktiven Editor zuzugreifen. Dies nutzen insbesondere Klassen der `visual`-Pakete, da der aktuelle Editor jener ist, in dem das zur Zeit visualisierte Modell vorliegt. Weiterhin finden sich dort Möglichkeiten, zwischen verschiedenen Bildformaten oder Adressformaten, wie auch in Abschnitt 5.2 erklärt, zu übersetzen.

Ein weiterer Teil des Kerns von KViD ist die Verwaltung der Laufzeit-Einstellungen.

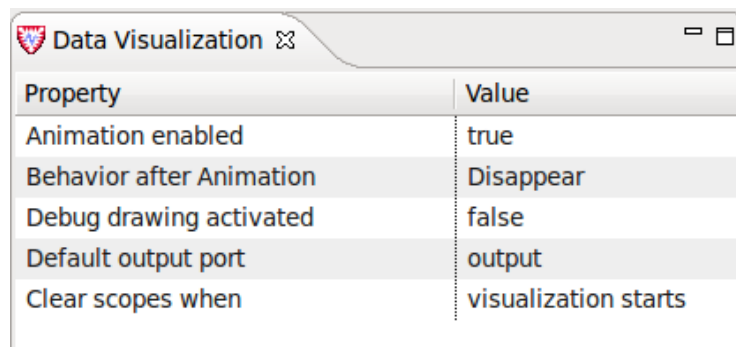
### 5.3.1. Laufzeit-Einstellungen

Um das Verhalten der Visualisierung zur Laufzeit zu ändern, wird über die Klasse `RuntimeConfiguration` eine Schnittstelle zur Veränderung von Einstellungen angeboten. Diese Einstellungen werden jeweils als ein eigenes `Property`-Objekt instanziiert und von der `RuntimeConfiguration` vorgehalten und verwaltet. Teilnehmer, die über die Veränderungen von Einstellungen informiert werden möchten, müssen das `IPropertyListener`-Interface implementieren, und sich als Beobachter bei der `RuntimeConfiguration` registrieren.



Die Einstellungen selber sind in zwei Arten klassifizierbar, *statische* sowie *dynamische* Einstellungen. Statische Einstellungen sind jene, die in jeder Situation verfügbar sind und zu Programmstart stets gleich initialisiert werden, beispielsweise die Option zur Aktivierung oder Deaktivierung der in Abschnitt 4.2.2 beschriebenen Animation. Dynamische Einstellungen hingegen werden zur Laufzeit generiert. So wird z.B. für jede dargestellte Marke ein **Property**-Objekt instantiiert, mit dessen Hilfe man das Verhalten der Marke beeinflussen kann. Man kann dann die Animation einer einzelnen Marke gezielt abschalten, oder die Marke statisch an einer der in Abschnitt 4.2.1 gezeigten Stellen anzeigen lassen.

Diese Einstellungen lassen sich sowohl programmatisch als auch über eine grafische Benutzeroberfläche, die in Abbildung 5.7 dargestellt ist, verändern.



Property	Value
Animation enabled	true
Behavior after Animation	Disappear
Debug drawing activated	false
Default output port	output
Clear scopes when	visualization starts

Abbildung 5.7.: Ansicht der Benutzeroberfläche für Einstellungen

Neben diesen Einstellungen werden auch die Repräsentationen der zu visualisierenden Daten von KViD in einer eigenen Struktur zwischengespeichert.

## 5.4. Datenhaltung

Die Haltung der zu visualisierenden Daten bzw. deren Repräsentation dient KViD zur direkten Verknüpfung derselben mit möglichen Metainformationen. Für eine Animation wie in Abschnitt 4.2.2 beschrieben sind Informationen über den Verlauf des Pfades vonnöten. Für eine Weiterverarbeitung, etwa in ein Diagramm wie in Abschnitt 4.3.1 gezeigt, ist Wissen über den Typ der Daten essentiell.

Angelehnt an die Namensgebung von Orosco und Moriyón [21] wird zu diesem Zweck ein **DataObject** implementiert. Eine Übersicht über die Methoden und Attribute eines **DataObject**s zeigt Abbildung 5.8. Es werden die URI des Datums als **KvidUri** Objekt, die Daten selbst sowie deren Typ, Informationen zum Verlauf der Werte des Datums und die möglichen Pfade bei einer Animation gespeichert.

Es ist nun zu klären, auf welchem Weg die Daten bekannt werden, die durch die **DataObject**s repräsentiert werden.

## 5. Implementierung

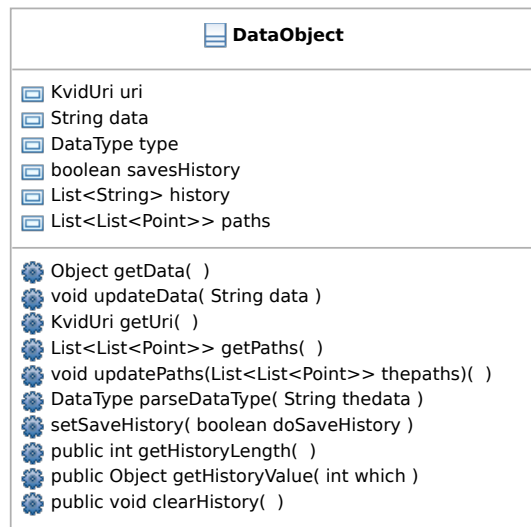


Abbildung 5.8.: Methoden und Attribute eines **DataObjects**

## 5.5. Datenquellen

Daten können aus unterschiedlichen Quellen stammen, denkbar sind beispielsweise Simulationen eines Modells oder Dateien von der Festplatte. Um eine einheitliche Schnittstelle zu haben, wurde für die Verwaltung von Datenquellen der in Abschnitt 3.2.1 beschriebene Execution Manager gewählt.

Um über KIEM Daten empfangen zu können, muss der **KiemDataProvider** als **DataComponent** implementiert werden und sich bei KIEM als Beobachter registrieren. Für das Weitergeben der erhaltenen Daten verwaltet der **KiemDataProvider** eine Liste von Beobachtern, die das **IProviderListener**-Interface implementieren. Wie in Abschnitt 5.3 beschrieben, ist der **DataDistributor** eine solche implementierende Klasse. Auch in KIEM erfolgt die Kommunikation über **JSONObjects**, so dass hier keine Umwandlung nötig ist. Die Daten bzw. deren Repräsentationen können direkt an die registrierten Beobachter weitergegeben werden.

Zu den beschriebenen Beobachtern müssen KIEM-Komponenten existieren, die Daten produzieren oder aus einer externen Quelle laden. Es existieren zwei Datenquellen, die die Daten in einem für KViD lesbaren Format liefern. Dies ist zum einen der **KAOMPtolemySimulator**, der mithilfe der Simulationmöglichkeiten des in Abschnitt 2.2.1 beschriebenen Ptolemy II Werkzeugs ein Modell simulieren kann, welches auch vom KAOM Editor anzeigbar ist. In Abbildung 5.9 wird ein Modell gezeigt, welches durch den Ptolemy II Simulator simuliert wird und dadurch Daten generiert. In der links befindlichen *Data Table*, einer weiteren **DataComponent** von KIEM, werden die aktuellen Daten als Text visualisiert. Die Visualisierung erfolgt nach einem Schlüssel-Wert Schema, wie es auch schon bei den **JSONObjects** der Fall war. Der Schlüssel ist auch in diesem Fall eine Adresse nach dem in Abschnitt 5.2 beschriebenen Schema. An der Data Table kann man gut einsehen, welche Reprä-

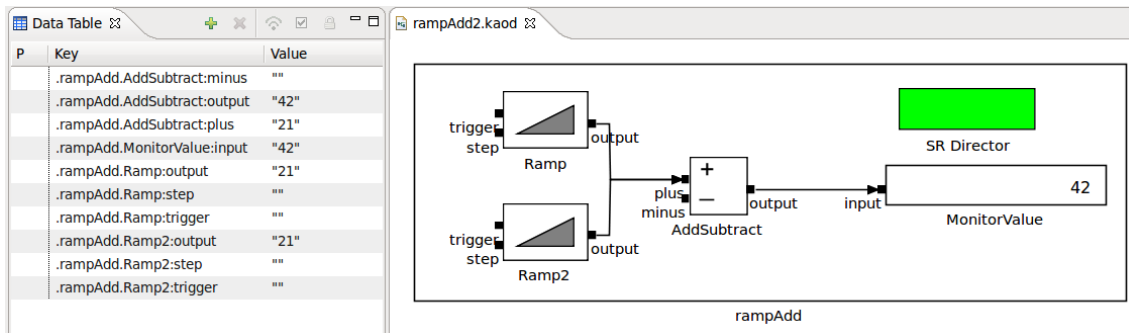


Abbildung 5.9.: Modell und generierte Daten des Simulators

sentationen auch KViD zur Verfügung stehen, da hier alle Daten in einer ähnlichen Formatierung visualisiert werden.

Als weitere mögliche Datenquelle wurden Dateien auf der Festplatte genannt. Um dies zu verdeutlichen, wird hier ein Schema vorgestellt, nach dem Daten an KViD mithilfe des in Abschnitt 3.5.2 beschriebenen CSV-Formats kommuniziert werden können.

Um von KViD verarbeitet werden zu können, wird eine `DataComponent`, ähnlich wie der Simulator, erstellt, der die gelesenen Daten als `JSONObject` im Execution Manager zur Verfügung stellt. Dieser `KiemCsvDataProvider` benötigt nun CSV-Dateien, die ihm die Zuordnung von Adresse und Datum erlauben. In der ersten Zeile befinden

Listing 5.1: Beispiel einer richtig formatierten CSV-Datei

```

1 .elm1.innerElement1:output;.elm1.innerElement2:output;.elm1.innerElement3:output
2 0;1;2
3 2;2;4
4 4;3;6
5 6;4;8

```

sich, durch Semikolons getrennt, die Adressen der referenzierten Modellelemente, im Beispiel in Listing 5.1 im Namen-basierten Adressformat. Voraussetzung für eine erfolgreiche Visualisierung ist hierbei, dass die Adressierung zu einem vorliegenden grafischen Modell passt. Nach einem Zeilenumbruch folgen dann beliebig viele Zeilen, in denen, wieder durch Semikolons getrennt, den Elementen aus der ersten Zeile Daten zugeordnet werden. Sollte man eine Weiterverarbeitung der Daten durch KViD, etwa in ein Balkendiagramm, wünschen, so müssen diese Daten alle dem selben Datentyp, z.B. Integer entspringen. Da dieses Format ein Klartextformat ist, lassen sich so große Datenmengen behandeln und generieren. Dem `KiemCsvDataProvider` übergibt man nun den Pfad zu einer solchen CSV-Datei. Die Verwaltung und Weiterleitung der gelesenen Daten wird dann wieder von KIEM übernommen.

Nach dem beschriebenen Schema lassen sich auch weitere Datenquellen hinzufü-

## 5. Implementierung

gen. Voraussetzung ist, dass diese als **DataComponent** von KIEM eine Adresse und ein Datum nach dem geschilderten Schlüssel-Wert Schema als **JSONObjects** zu Verfügung stellt.

Nachdem Daten bzw. deren Repräsentationen zusammengetragen wurden, folgen nun die Bestandteile von KViD, die für die tatsächliche Visualisierung im Modell-diagramm sorgen. Den Anfang macht hier die flüchtige Visualisierung mithilfe von Marken.

### 5.6. Flüchtige Visualisierung mit Marken

Grundlage für die Erstellung von Marken, welche die nicht-persistente Komponente von KViD darstellen, ist eine Zeichenklasse, die das **IDrawer**-Interface implementiert. Dieses Interface wird eingeführt, um die Zeichenklasse bei Bedarf wechseln zu können. Das Interface verlangt zwei Methoden: eine zum Zeichnen der zu visualisierenden Daten und ein weitere, um eine Zeichnung zurückzusetzen. Diese Zeichenklasse implementiert zusätzlich das **IDataListener**-Interface, und registriert sich als Beobachter beim **DataDistributor**. KViD enthält eine Implementierung für das GMF, denkbar sind hier jedoch Implementierungen für jede beliebige grafische Oberfläche.

Jedes grafische Objekt in einem GMF/GEF-Editor ist eine sogenannte **Figure** und erfüllt als solche das **IFigure**-Interface. Da grafische Objekte wie die Marken in die-

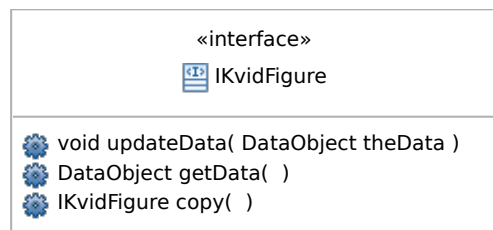


Abbildung 5.10.: Interface für eine Grafik mit Datenverwaltung

sem Fall Daten anzeigen sollen, wird das Interface **IKvidFigure** eingeführt, welches zusätzlich Datenverwaltung fordert. Eine Übersicht über das Interface zeigt Abbildung 5.10. Es wird mit **copy()** auch eine Methode gefordert, die die **IKvidFigure** dupliziert. Dies ist notwendig, wenn eine Visualisierung beispielsweise mehreren Verbindungslinien zugeordnet werden soll.

Die einfachste implementierende Klasse ist die **GmfFigure**. Die **GmfFigure** ist die Klasse, die eine einfache Marke, wie in Abschnitt 4.2 beschrieben, repräsentiert. Die Marken sind als nicht-persistente Darstellung vorgesehen, das heißt, nach einem Neustart oder Beendigung der Visualisierung werden sie nicht weiter dargestellt. Der **GmfDrawer** zeichnet sie daher nur auf die grafische Oberfläche und erstellt kein zugehöriges, persistentes Objekt. In der in Abbildung 5.11 dargestellten Struktur der grafischen Ebenen von GEF werden die Marken auf einem *Printable Layer* gezeichnet.

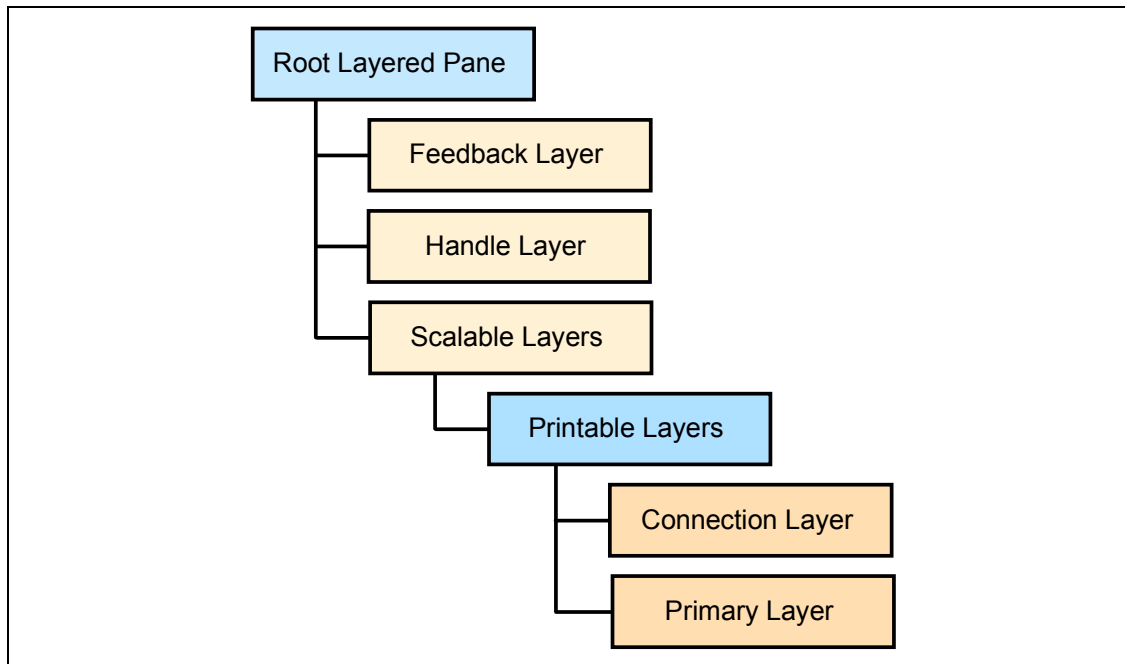


Abbildung 5.11.: Die verschiedenen grafischen Ebenen von GEF nach Moore et. al. [19]

Durch die Adresse, die jedem Datum zugeordnet wurde, kann nun eine Position im Modelldiagramm ermittelt werden, an welcher die Daten visualisiert werden können. Für die Bestimmung dieser Positionen wird der *Layout-Graph* des KIML-Plug-ins genutzt, welches in Abschnitt 3.2.2 beschrieben wird. Dieser Layout-Graph enthält die Informationen, die nötig sind, um Position und Verlauf des Modellelements und seiner Verbindungslinien festzustellen. Die Marke wird dann zunächst an den Anfang der Verbindungslinie gezeichnet.

Da die Positionsfindung aus Leistungsgründen nur zu Anfang einer Visualisierung stattfindet, muss nach Veränderung des Diagramms eine erneute Positionierung stattfinden. Zu diesem Zweck beobachtet der `DataDistributor` die Datei des Modelldiagramms und merkt an, wenn der Layout-Graph neu ausgelesen werden muss. Dies ist insbesondere für die Animation von Marken von besonderer Bedeutung.

### 5.6.1. Animation mit dem Multi-View Editor Kit

Um eine Animation mithilfe des in Abschnitt 3.3 eingeführten Multi-View Editor Kits in KIELER zu ermöglichen, musste es marginal angepasst werden. Es war hierzu notwendig, die von KIELER genutzten Editortypen dem Multi-View Editor Kit bekannt zu machen.

Für eine konkrete Animation während der Visualisierung benötigt man nun weitere Informationen aus dem Layout-Graphen. Der `GmfAnimator`, der vom `GmfDrawer` am Ende des Zeichnens aufgerufen wird, benötigt für das Animieren der Marken

## 5. Implementierung

einen Pfad von Punkten im Editor, welchen die Marke nach und nach ablaufen wird. Somit werden jedem `DataObject` diese aus dem Layout-Graphen gewonnenen Pfade hinzugefügt. Diese Pfade werden dann vom `GmfAnimator` genutzt. Eine verein-

Listing 5.2: Erstellen des Animierungskommandos

```
1 AnimatingCommand anima = new AnimatingCommand();
2 while (!allPathsExeeded) {
3     for(IKvidFigure figure : allFigures) {
4         if (pathCounter == 0) {
5             anima.initializeAnimatedElement(figure,
6                 diagram.getViewer());
7         }
8         if (pathCounter < path.size()) {
9             anima.specifyStep(figure, path.get(pathCounter));
10            if ((pathCounter + 1) < path.size()) {
11                allPathsExeeded = false;
12            }
13        } else {
14            insertDummyStep();
15        }
16    }
17    if (!allPathsExeeded) {
18        anima.nextStep();
19    }
20    pathCounter++;
21 }
```

fachte Darstellung der Erzeugung eines Animierungskommandos ist in Listing 5.2 gezeigt. Dort werden, solange nicht alle Punkte aus allen Pfaden in das Kommando aufgenommen wurden, die Figures durchlaufen und der jeweils nächste Punkt dem Kommando hinzugefügt. Wenn noch nicht alle Punkte abgearbeitet wurden, aber ein Pfad einer Marke bereits zu Ende ist, dann muss für die folgenden Schritte der letzte Punkt wiederholt eingefügt werden, da das Multi-View Editor Kit für jeden Schritt und jede Marke einen Punkt erwartet. Das Animierungskommando wird dann an die Klasse, die das Modelldiagramm verwaltet, weitergegeben, welches Kommandos in einem *Stack* vorhält und sie ausführt. Es werden schließlich alle Marken gleichzeitig auf den ausgehenden Verbindungslinien von den Knoten animiert, denen zuvor Daten zugeordnet wurden.

Einen anderen Ansatz als flüchtige Marken in Nutzen und Funktion bilden, wie in Kapitel 4 dargelegt, die persistenten Visualisierungsobjekte.

## 5.7. Persistente Visualisierungsobjekte

Persistente Visualisierungen unterscheiden sich, wie der Name nahelegt, darin, dass diese vom Benutzer mit Modellelementen verknüpft werden und dann persistent gehalten werden. Somit sind sie nach der Visualisierung oder auch nach dem Neustart des Werkzeugs noch vorhanden.

Um dies zu erreichen, ist ein tieferer Einblick in die Mechanismen von EMF, GEF und GMF und des Eclipse Werkzeugs nötig. Es muss ein neuer Knotentyp eingeführt werden, welcher von allen Ebenen der Kontrollstruktur von Eclipse verstanden wird. Um dies zu erreichen, ist es zunächst einmal notwendig, den neuen Knoten als *ElementType* in einem Extension Point zu definieren. Wichtig ist hierbei, dass das neue Element ein sogenannter *INotationType* ist. Dieses von GMF bereitgestellte Interface wird für Diagrammelemente genutzt, die nur zur Sicht auf das Modell, aber nicht zum Modell selbst gehören. Auch muss ein eindeutiger sogenannter *semanticHint* gewählt werden, über den später der neue ElementType referenziert wird. Ein vollständiges Beispiel für einen Extension Point, der ein Element zur Darstellung von Diagrammen definiert, ist in Listing 5.3 aufgeführt. Es wird dort in

Listing 5.3: Extension Point Definition für einen ElementType

```

1 <extension
2     id="de.cau.cs.kieler.kvid.elementTypes"
3     name="KViDElementTypes"
4     point="org.eclipse.gmf.runtime.emf.type.core.elementTypes">
5     <elementTypeFactory
6         factory="org.eclipse.gmf.runtime.diagram.ui.internal.type.
7                 NotationTypeFactory"
8         kind="org.eclipse.gmf.runtime.diagram.ui.util.INotationType"
9         params="semanticHint">
10    </elementTypeFactory>
11    <specializationType
12        icon="icons/datanode.gif"
13        id="de.cau.cs.kieler.kvid.visual.complex.datanode"
14        kind="org.eclipse.gmf.runtime.diagram.ui.util.INotationType"
15        name="%DataNode.Label">
16        <param
17            name="semanticHint"
18            value="DataNode">
19        </param>
20        <specializes
21            id="org.eclipse.gmf.runtime.emf.type.core.null">
22        </specializes>
23    </specializationType>
24 </extension>

```

## 5. Implementierung

Zeile 4 eine `elementTypeFactory` angeben, welche jedoch bei allen Visualisierungselementen gleich und so wie dort beschrieben zu wählen ist. Entscheidend ist der `specializationType`, dessen Definition in Zeile 11 beginnt. Als `kind` muss auch hier `INotationType` gewählt werden, `icon`, `id` und `name` können beliebig gewählt werden, wobei die `id` global eindeutig sein muss. Ebenfalls eindeutig sein muss der in den Zeilen 17 und 18 definierte `semanticHint`. Hierüber wird der neue Typ später referenziert. Die Option `specializes` spielt für diesen Anwendungsfall keine Rolle und kann auf dem Standardwert belassen werden.

Nun muss noch definiert werden, wie die grafische Komponente und der Controller, ein sogenannter `EditPart`, für das neue Element erzeugt werden. Auch dies erfolgt über Extension Points, die den `DataDisplayViewProvider` und den `DataDisplayEditPartProvider` registrieren, so wie in Listing 5.4 dargestellt. Es muss dort für die neuen ElementTypes ein `ViewProvider` und ein `EditPartProvider` registriert werden. Dem `ViewProvider` wird in den Zeilen 9 und 10 zusätzlich der `semanticHint` und die Art der View, in diesem Fall ein Knoten, mitgeteilt. Beim Laden eines grafischen Modells, oder beim Erstellen eines neuen Elements in diesen, werden alle registrierten Provider gefragt, ob sie diesen `ElementType` kennen. Die beiden neu registrierten Provider müssen nun im Falle der `KViD-ElementTypes` dies bejahen und dann eine `View`-Instanz und einen `EditPart` bereitstellen. Dieser Vorgang wird in Abbildung 5.12 schematisch dargestellt. Damit diese Funktionalität dem Benutzer zur Verfügung gestellt wird, muss das Erstellen der neuen Knotentypen in der Benutzeroberfläche ermöglicht werden. Über weitere Extension Points wird ein Eintrag in ein Pop-Up Menü, sowie ein `Command` zur Erzeugung eines Elements des neuen Typs registriert. Dieses `Command` referenziert dann einen Handler, den allgemein gehaltenen `DisplayNodeHandler`, welcher in der `execute()`-Methode die entsprechenden Requests zum Erstellen des neuen Knotens zusammenbaut. Hierzu muss dem `Command` der `semanticHint` des `ElementTypes` als Parameter mitgegeben werden. Diese Request wird dann an den `EditPart` übergeben, welcher, wie in Abbildung 5.13 dargestellt, zu dieser Request ein `Command` erstellt. Dies ist ein Kommando wie es auch in Abschnitt 5.6.1 zum Animieren verwendet wird, und darf nicht mit dem `Command` verwechselt werden, welches zur Bereitstellung eines Menüeintrags verwendet wird. Das erstellte `Command` wird dann wie das Animierungskommando ausgeführt. Als letzter Schritt für das Erstellen eines neuen Knotentyps bleibt das Beschreiben des neuen Knotentyps. Man muss die Klasse `AbstractDisplayFactory` erweitern und dort zu den `semanticHints`, die man definiert hat, Methoden angeben, die die entsprechenden `EditParts` erzeugen. Diese Fabrik muss dann noch über den `dataDisplay` Extension Point mit ihren bekannten `semanticHints` registriert werden. In den erzeugten `EditParts` kann man schließlich als `IFigure` jede beliebige Art der Visualisierung erstellen. Damit der `EditPart` auch über Änderungen und neue Daten informiert wird, muss er sich als `IDataListener` beim `DataDistributor` registrieren.

Zur Verdeutlichung des Ablaufs, welcher für das Hinzufügen eines neuen Knotentyps nun noch notwendig ist, wird im folgenden Abschnitt beschrieben, wie ein in Abschnitt 4.3.1 beschriebener Diagrammknoten als Knotentyp eingeführt wird. Auf die gleiche Art können dann beliebige andere Knotentypen eingefügt werden.



Listing 5.4: Extension Point Definition für den View- und EditPartProvider

```

1 <extension
2   point="org.eclipse.gmf.runtime.diagram.core.viewProviders">
3 <viewProvider
4   class="de.cau.cs.kieler.kvid.visual.complex.DataDisplayViewProvider">
5   <Priority
6     name="Lowest">
7   </Priority>
8   <context
9     semanticHints="DataNode"
10    viewClass="org.eclipse.gmf.runtime.notation.Node">
11   </context>
12 </viewProvider>
13 </extension>
14 <extension
15   point="org.eclipse.gmf.runtime.diagram.ui.editpartProviders">
16 <editpartProvider
17   class="de.cau.cs.kieler.kvid.visual.complex.DataDisplayEditPartProvider">
18   <Priority
19     name="Lowest">
20   </Priority>
21 </editpartProvider>
22 </extension>

```

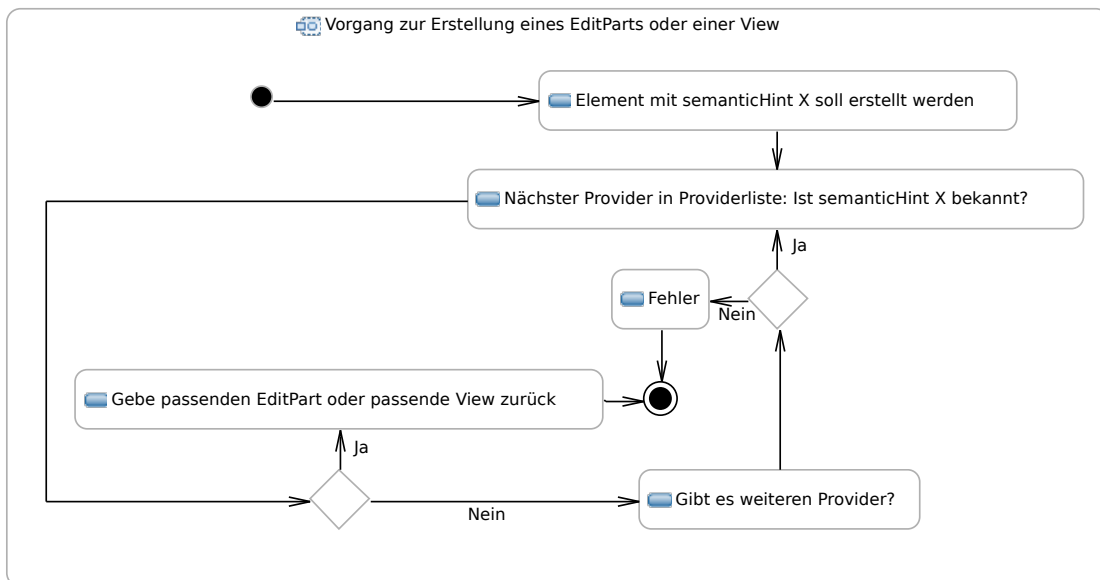


Abbildung 5.12.: Vorgang zur Erstellung eines EditParts oder einer View

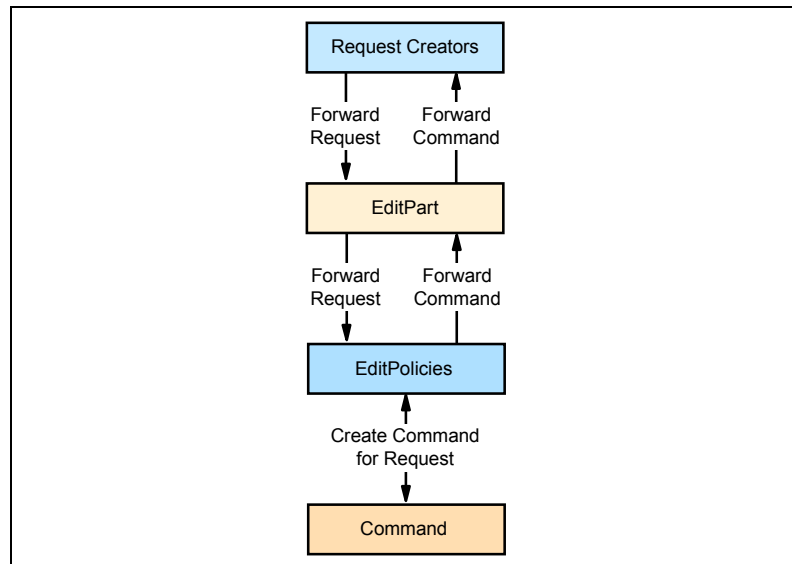


Abbildung 5.13.: Der Request-Mechanismus von GEF nach Moore et. al. [19]

### 5.7.1. Graphen- und Diagrammknoten

Mithilfe dieser Knoten werden Elemente überwacht und ihr Werteverlauf über die Zeit in einem Diagramm oder durch einen Graphen visualisiert. Hierzu verwaltet der **EditPart** dieses Knotens eine Instanz des Graphen- und Diagramm-Zeichenwerkzeuges von Ptolemy II. Dieses empfängt die gelesenen Werte und zeichnet ein passendes Diagramm. Dieses wird dann als Bild exportiert und so im Diagramm dargestellt.

Listing 5.5 zeigt die Registrierung der Fabrik für das Erzeugen der **EditParts**. Es

Listing 5.5: Extension Point Definition für die Fabrik

```

1 <extension
2     point="de.cau.cs.kieler.kvid.dataDisplay">
3     <displayFactory
4         factoryClass="de.cau.cs.kieler.kvid.ui.displays.DisplayFactory">
5         <knownSemanticHints
6             semanticHint="ScopeNode">
7         </knownSemanticHints>
8     </displayFactory>
9 </extension>
    
```

wird dazu in Zeile 4 eine Implementierung der **AbstractDisplayFactory** Klasse angegeben und in der Zeile 6 der **semanticHint**, den die Fabrik verarbeiten kann. Unter **knownSemanticHints** können beliebig viele **semanticHints** angegeben werden, solange diese von der Fabrik verarbeitet werden können.

In Listing 5.6 ist die Implementierung der Fabrik für einen Diagrammknoten dargestellt. Es wird nach dem **semanticHint** entschieden, welcher **EditPart** erstellt wird.

Listing 5.6: Implementierung der EditPart-Fabrik

```

1 public class DisplayFactory extends AbstractDisplayFactory {
2
3     public DisplayFactory() { }
4
5     public AbstractDataDisplayEditPart create(final View view,
6         final String semanticHint) {
7         if (semanticHint.equals("ScopeNode")) {
8             return new ScopeEditPart(view);
9         }
10        return null;
11    }
12 }

```

Listing 5.7: Extension Point Definition für den ElementType

```

1 <extension
2     id="de.cau.cs.kieler.kvid.elementTypes"
3     name="KViDElementTypes"
4     point="org.eclipse.gmf.runtime.emf.type.core.elementTypes">
5     <elementTypeFactory
6         factory="org.eclipse.gmf.runtime.diagram.ui.internal.type.
7             NotationTypeFactory"
8         kind="org.eclipse.gmf.runtime.diagram.ui.util.INotationType"
9         params="semanticHint">
10    </elementTypeFactory>
11    <specializationType
12        icon="icons/datanode.gif"
13        id="de.cau.cs.kieler.kvid.ui.displays.scopenode"
14        kind="org.eclipse.gmf.runtime.diagram.ui.util.INotationType"
15        name="%ScopeNode.Label">
16        <param
17            name="semanticHint"
18            value="ScopeNode">
19        </param>
20        <specializes
21            id="org.eclipse.gmf.runtime.emf.type.core.null">
22        </specializes>
23    </specializationType>
24 </extension>

```

Wird kein passender semanticHint gefunden, so wird null zurückgegeben. Die Definition des ElementTypes, wie sie in Listing 5.7 gezeigt ist, entspricht im wesentlichen der beispielhaften Definition aus Listing 5.3. Für die Beispielwerte wurden hier die

## 5. Implementierung

richtigen Werte für einen Diagrammknoten eingefügt.

Der letzte notwendige Extension Point wird in Listing 5.8 gezeigt. Dieser ist für das Auftauchen eines Eintrags in ein Menü notwendig. In Zeile 4 wird bestimmt, in welchem Menü der Eintrag erscheinen soll. In Zeile 6 wird das `Command` für das Hinzufügen einer `DisplayNode` über seine `commandId` referenziert. Dieses ist stets so wie hier zu wählen. Die Zeilen 7 bis 11 können nach belieben ausgefüllt werden, da sie die Schnittstelle zum Benutzer sind, sollten sie ihm aber eine gute Orientierung bieten.

Es muss dann noch der richtige Knotentyp referenziert werden. In Zeile 13 wird der entsprechende Parameter benannt, auch dieser ist stets gleich. In Zeile 14 wird dann der Typ referenziert, der *value* hier entspricht für den Diagrammknoten der `id` des `ElementTypes`, also konkret Zeile 13 in Listing 5.7.

Listing 5.8: Extension Point Definition für den Menüeintrag

```
1 <extension
2   point="org.eclipse.ui.menu">
3   <menuContribution
4     locationURI="popup:de.cau.cs.kieler">
5     <command
6       commandId="de.cau.cs.kieler.kvid.addDisplayNodeCommand"
7       disabledIcon="icons/kvid_disabled.png"
8       icon="icons/kvid.png"
9       label="Add Scope Node"
10      mnemonic="S"
11      style="push">
12      <parameter
13        name="de.cau.cs.kieler.kvid.visual.complex.displayID"
14        value="de.cau.cs.kieler.kvid.ui.displays.scopenode">
15      </parameter>
16    </command>
17  </menuContribution>
18 </extension>
```

## 6. Mögliche Erweiterungen von KIELER Visualization of Data

Für einige Teile von KViD existieren bereits theoretische Überlegungen zu Erweiterungen. Das folgende Kapitel bietet eine Übersicht über diese Ideen und Erweiterungen. Zu jedem dieser Vorschläge wird zusätzlich noch ein möglicher Lösungsansatz angegeben, der aber ebenfalls auf theoretischen Überlegungen beruht und noch nicht auf seine Praxistauglichkeit untersucht wurde.

- **Erweiterung:** Zur Zeit ist es mit KViD nur möglich, Daten für einen Editor zur Zeit zu visualisieren. Es ist aber denkbar, eine Visualisierung für jeden von mehreren parallel geöffneten Editoren gleichzeitig durchzuführen. Das Durchführen mehrerer Visualisierungen gleichzeitig wird in der vorliegenden Implementierung durch die Singleton-Struktur der Kern-Klassen verhindert. Denkbar wäre hier eine Verwalterklasse, die die gesamte KViD Architektur für jede neue Visualisierung vorhält. Es muss dann aber jedem Visualisierungselement bekannt sein, welcher Visualisierung es zuzuordnen ist.
- **Erweiterung:** Um eine bessere Unterstützung verschiedener Editoren und Adressierungsformate zu erlauben, kann eine Schnittstelle, die neue Adressierungsformate registriert, eingesetzt werden. Dies könnte über den Extension-Point Mechanismus von Eclipse gelöst werden. Es würde ein Extension-Point angeboten werden, in dem man ein Adressformat und eine Klasse registrieren kann, die dieses Format in ein bereits von KViD bekanntes Format, wie sie in Abschnitt 5.2 beschrieben werden, übersetzt.
- **Erweiterung:** Als eingebettete Elemente im Modelldiagramm, wie in Abschnitt 5.7 beschrieben, bieten sich auch animierte Grafiken an. Eine Grundlage hierfür ist mit KEV bereits vorhanden. Das Einbinden von Grafiken an sich ist bereits mit KViD möglich. Es muss nun eine Verknüpfung zwischen KEV und KViD stattfinden, die es ermöglicht, die von KViD angezeigten Grafiken mit KEV auf Grundlage der vorhandenen Daten zu manipulieren.
- **Erweiterung:** Die in Abschnitt 5.3.1 erklärten Laufzeit-Einstellungen sind nur zur Laufzeit verfügbar und werden bei einem erneuten Start des Programms auf ihre Standardwerte zurückgesetzt. Wünscht ein Benutzer andere Werte, so sollte es ihm ermöglicht werden, diese als Standard festzulegen.

## 6. Mögliche Erweiterungen von KIELER Visualization of Data

Mit den *Eclipse Preference Pages* bietet Eclipse die Möglichkeit, für ein bestimmtes Plug-in Einstellungen anzugeben, die ein Benutzer dann vornehmen kann. Es müsste für KViD eine ebensolche Einstellungsmöglichkeit geschaffen werden.

- **Erweiterung:** Marken zeigen zur Zeit nur die textuelle Visualisierung eines Datums. Für spezielle Datentypen könnten hier auch andere Darstellungen in Markenform möglich sein. Ein einfaches Beispiel wären rote und grüne Marken für Wahrheitswerte.

`DataObjects` können in gewissem Maße schon Datentypen erkennen. Diese müssten bei der Erstellung von Marken dann berücksichtigt werden. Die Hauptarbeit bei dieser Erweiterung liegt in der Findung sinnvoller Marken für die Datentypen.

- **Erweiterung:** Die grafischen Elemente von KViD sind für GMF-Editoren vorgesehen. Es gibt jedoch das Grafik-Framework Graphiti<sup>1</sup>, welches ebenfalls die Darstellung von EMF-basierenden Editoren erlaubt. Es ist denkbar, auch hierfür Visualisierungen zu implementieren.

Am Kern von KViD müssten dazu keine Änderungen erfolgen. Lediglich die Bestandteile der `visual-` und `visual.complex-`Pakete, also die View-Komponente, sowie mögliche für GMF erstellte Erweiterungen, müssten erneut für Graphiti implementiert werden.

---

<sup>1</sup><http://www.eclipse.org/graphiti/>

## 7. Fazit

Im Laufe der Arbeit wurde die Visualisierung von Daten im Zusammenhang mit grafischer Modellierung behandelt. Es wurde zunächst dargelegt, welche Arbeiten bereits zu dem Thema vorhanden sind und wie die Visualisierung von Daten in einigen anderen ausgewählten Werkzeugen zur grafischen Modellierung umgesetzt wurde. Dabei wurde erkannt, dass Visualisierung eine wichtige Rolle bei modellbasiertem Design spielt.

Es existieren einige Ansätze oder auch konkrete Implementierungen in Werkzeugen wie Ptolemy II, die Vielzahl bezieht sich jedoch auf einen konkreten Anwendungsfall. Auf dieser Grundlage wurden Ansätze weiterentwickelt und neue Ideen vorgestellt und evaluiert. Besonderes Augenmerk wurde hierbei auf eine möglichst vielfältige Nutzbarkeit geworfen und es wurde darauf geachtet, eine erweiterbare Grundlage zu erstellen.

Um die Nutzbarkeit dieser Grundlage zu zeigen, wurde dann eine beispielhafte Implementierung im Zusammenhang mit dem KIELER-Projekt vorgestellt und auf Vorgehensweisen und Lösungen eingegangen. Die implementierten Visualisierungskonzepte verdeutlichten die Benutzung der geschaffenen Grundlage und erklärten an ihrem Beispiel, wie eine Erweiterung im Rahmen der geschaffenen Grundlage möglich ist. Abschließend wurde ein Ausblick auf mögliche Erweiterungen und Verbesserungen gegeben. Die Verbesserungen bezogen sich dabei zum Großteil auf eine bessere Nutzbarkeit der Grundlage.

Betrachtet man nun erneut die einzelnen Punkte der Aufgabenstellung, so stellt man fest, dass diese durch die Arbeit und die zugehörige Implementierung erfüllt werden:

- Finden verschiedener Möglichkeiten, vielfältige Daten eingängig und leicht verständlich darzustellen.

Das Kapitel 4 beschreibt Möglichkeiten flüchtiger und persistenter Visualisierung. Verständlichkeit hängt von der Qualifikation des Benutzers ab, weswegen eine zuverlässige Aussage hierüber nicht ohne weiteres möglich ist. Die verschiedenen vorgestellten Möglichkeiten der Visualisierung zielen aber darauf ab, einer möglichst breiten Menge an Benutzern Daten verständlich darzustellen.

- Implementieren einer oder mehrerer Anbindungen an eine Datenquelle oder eine Simulationsumgebung.

Durch die in Abschnitt 5.5 beschriebenen Teile der Implementierung werden sowohl eine Simulationsumgebung als auch eine beispielhafte Datenquelle zur Anbindung bereitgestellt.

## 7. Fazit

- Implementierung und Einbettung dieser Möglichkeiten in eine grafische Entwicklungsumgebung, in der die zu den Daten gehörenden Modellen vorliegen.

Die Implementierung ist Teil des KIELER-Projekts, welches den Fokus auf grafische Modellierung gelegt hat. Eine Anbindung an den in Abschnitt 3.2.3 beschriebenen Editor wurde angefertigt.

- Verbinden der Daten aus der Datenquelle oder Simulationsumgebung mit den grafischen Darstellungsmöglichkeiten.

Auch dies wurde im Rahmen der Implementierung geleistet. Über die Adressierung, die in Abschnitt 5.2 beschrieben wird, können Daten bzw. deren Visualisierung den passenden Modellelementen zugeordnet werden.

Mit der Erfüllung der Aufgabenstellung wurde eine Grundlage für Datenvisualisierung im KIELER-Projekt geschaffen. Die beispielhafte Implementierung für den KAOM-Editor zeigt das Potential des geschaffenen Frameworks für die Nutzung im Zusammenhang mit Datenflussmodellen. Durch die Erweiterbarkeit ist es jedoch nicht auf diese beschränkt.

Es wurde deutlich, dass das Verknüpfen von Daten mit einem grafischen Modell nicht trivial ist. KViD kann dem Benutzer diese Arbeit abnehmen und somit das Verständnis eines grafischen Modells fördern. Es fügt sich damit in die Reihe von Teilprojekten des KIELER-Projekts ein, die eine bessere Handhabung der grafischen Modellierung zum Ziel haben.



# A. Arbeiten mit KIELER

## Visualization of Data

Das folgende Kapitel soll einen Überblick über die Benutzung der Grundfunktionen von KViD geben. Es wird hierbei auf die Funktionen eingegangen, die ein späterer Benutzer zur Verfügung haben soll. Hinweise für Entwickler, die KViD weiterentwickeln möchten, wurden in Kapitel 5 gegeben.

Zunächst werden Voraussetzungen erläutert, die ein Modell erfüllen muss, um eine Visualisierung mit KViD zu erlauben. Dann wird detaillierter auf die Benutzung eingegangen. Es wird beschrieben, wie man KViD im Execution Manager aktiviert. Es werden danach mögliche Datenquellen vorgestellt und dargelegt, welche Einstellungen man zur Laufzeit von KViD vornehmen kann. Schließlich wird das Arbeiten mit eingebetteten Elementen wie Diagrammen im Modelldiagramm erläutert.

### A.1. Voraussetzungen

Es gibt zwei Voraussetzungen, die ein Modell erfüllen muss, damit es mit KViD visualisierbar ist.

- Es muss ein GMF-Editor existieren, der das Modell darstellen kann. Diese werden auch in Abschnitt 3.1.2 beschrieben.
- Die Elemente des Modells müssen adressierbar sein, nach einem Schema wie es in Abschnitt 5.2 beschrieben wird.

Sind diese Voraussetzungen erfüllt, so können die Daten von KViD eingelesen und im Modelldiagramm dargestellt werden. Ein Editor und dessen zugehörige Modelle, der die Voraussetzungen erfüllt, ist der in Abschnitt 3.2.3 beschriebene KAOM Editor des KIELER-Projekts.

Die Aktivierung und Steuerung von KViD erfolgt unter anderem über den Execution Manager. Zu diesem Zweck muss man die gewünschten Komponenten in KIEM aktivieren.

### A.2. Aktivierung im Execution Manager

Für die Visualisierung sind zwei Dinge zwingend erforderlich, zum einen die KViD Komponente selbst und zum anderen eine Datenquelle. Diese Datenquelle muss Daten bzw. Repräsentationen der Daten so liefern, dass KViD sie verarbeiten und den

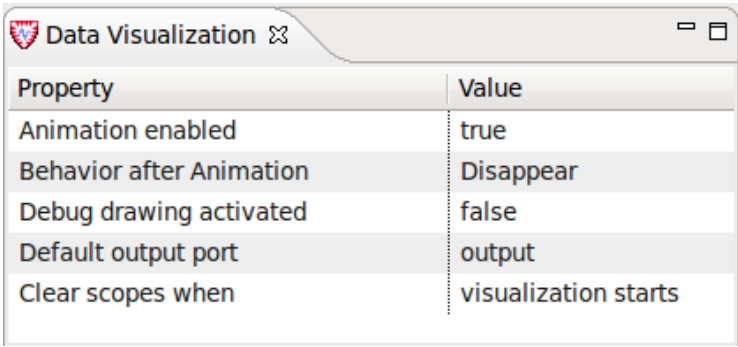
Elementen des Modelldiagramms zuordnen kann. Bei KAOM kann man beispielsweise den *KAOM Ptolemy Simulator* oder eine CSV-Datei über den *CSV Data Provider* als Datenquelle nutzen. Die CSV-Dateien müssen wie in Abschnitt 5.5 beschrieben passend zum Modell formatiert sein. Als Visualisierungskomponente nutzt man *Data Visualization*. Im Execution Manager müssen das Häkchen bei einer der Datenquellen und bei der Visualisierungskomponente gesetzt sein, bei allem anderen sollte es deaktiviert sein. Tauchen die genannten Komponenten nicht in der Tabelle auf, dann muss man diese über einen Rechtsklick und den Befehl *Add Component* hinzufügen.

Hierbei muss beachtet werden, dass die Visualisierungskomponente in der Liste unter der Datenquelle stehen muss. Nutzt man die CSV-Datenquelle, so muss man in dessen Eigenschaften noch den Pfad zu der passenden CSV-Datei angeben. Drückt man nun den *Step* oder *Play* Knopf des Execution Managers, so startet die Visualisierung der gewählten Daten. Die Länge der Animationen richtet sich nach der wählbaren *Step Size* des Execution Managers. Es ist ratsam, diese nicht zu klein zu wählen; alle Werte unter einer Sekunde erschweren das Lesen der visualisierten Daten. Außerdem werden Animationen übersprungen, wenn neue Daten vorliegen, bevor die Animation beendet ist.

Des Weiteren ist es möglich, über die *Data Visualization* Sicht das Verhalten der Visualisierung zu beeinflussen.

### A.3. Einstellungen zur Laufzeit

Beim Einstellen von KViD wird zwischen zwei Typen von Eigenschaften unterschieden. Globale Eigenschaften betreffen das Verhalten der gesamten Visualisierung, lokale Eigenschaften die Visualisierung von Daten, die einem bestimmten Modellelement zugeordnet werden. Die Ansicht der Eigenschaften ist kontextsensitiv: Hat man nichts ausgewählt, so erscheinen wie in Abbildung A.1 die globalen Eigenschaften. Wählt man jedoch ein Modellelement aus, zu dem Daten visualisiert werden, so zeigt die Ansicht Optionen, die dieses Element betreffen.



Property	Value
Animation enabled	true
Behavior after Animation	Disappear
Debug drawing activated	false
Default output port	output
Clear scopes when	visualization starts

Abbildung A.1.: Ansicht der globalen Einstellungen

Im Folgenden werden nun die globalen Optionen aufgelistet und ihre Bedeutung erklärt:

- **Animation enabled:** Diese Option steuert, ob bei der Visualisierung generell Animationen genutzt werden. Der Wert *true* steht für aktivierte Animation, *false* dementsprechend für deaktivierte.
- **Behavior after animation:** Hier kann entschieden werden, wie mit den Marken nach der Animation verfahren werden soll. *Disappear* lässt Marken an ihrem Ziel verschwinden, *stay at last location* lässt die Marke am Zielpunkt sichtbar, bis eine neue Animation beginnt.
- **Debug drawing activated:** Es ist möglich, sich den Pfad einer Animation als rote Linie im Diagramm anzeigen zu lassen. Setzt man diese Option auf *true*, so wird die Linie eingezeichnet, bei *false* nicht.
- **Default output port:** Hat ein Modell Konnektoren, sogenannte Ports, an denen Verbindungslinien platziert werden, müssen visualisierbare Daten diesen zugeordnet werden. Erfolgt dies nicht in der Adressierung, so kann hier der Name des Ports angegeben werden, der dann standardmäßig genutzt wird.
- **Clear scopes when:** Nutzt man eingebettete Diagramme, so werden diese nach der Visualisierung zurückgesetzt. Man kann den Zeitpunkt bestimmen, an dem dies passieren soll. *Visualization starts* setzt die Diagramme am Anfang einer neuen Visualisierung zurück, *visualization finishes* direkt am Ende der aktuellen Visualisierung.

Das Anzeigeverhalten einer Marke kann zusätzlich als lokale Option manipuliert werden. Wählt man das zugehörige Modellelement an, so kann man bei der Option **Display status** <Adresse des Objekts> zwischen fünf Möglichkeiten wählen. Bei *Animating* wird die Marke wie in Abschnitt 4.2.2 beschrieben animiert. Die Optionen *Static on Source Node*, *Static on Target Node* und *Static on middlemost Bend Point* zeigen die Marke unbewegt am Quellelement, Zielelement oder auf dem mittleren Knickpunkt der Verbindungslinie, wie auch in Abschnitt 4.2.1 erörtert wurde. Die Option *Invisible* sorgt dafür, dass keine Marke für das Datum generiert wird.

Neben der Visualisierung durch Marken bietet KViD die Möglichkeit, visualisierbare Elemente in das Modelldiagramm einzubetten. Diese müssen, anders als Marken, explizit vom Benutzer in das Modelldiagramm eingefügt werden.

## A.4. Hinzufügen von eingebetten Elementen

Ein eingebettetes Visualisierungselement kann mit jedem Modellelement verbunden werden, aber nicht mit Verbindungslinien. Dies wurde beschlossen, da automatisches

Layout einen Graphen, in dem eine Kante mit einer Kante verbunden ist, nicht verarbeiten kann. Da automatisches Layout ein Kernelement des KIELER-Projekts ist, sollten Unterprojekte sich dem anpassen. Damit das Visualisierungselement etwas anzeigen kann, muss es mit einem Modellelement verbunden werden, für welches Daten von der Datenquelle geliefert werden. Ein solches Modellelement wird wie in Abschnitt 5.2 adressiert. Zum Hinzufügen eines Diagrammelements wählt man

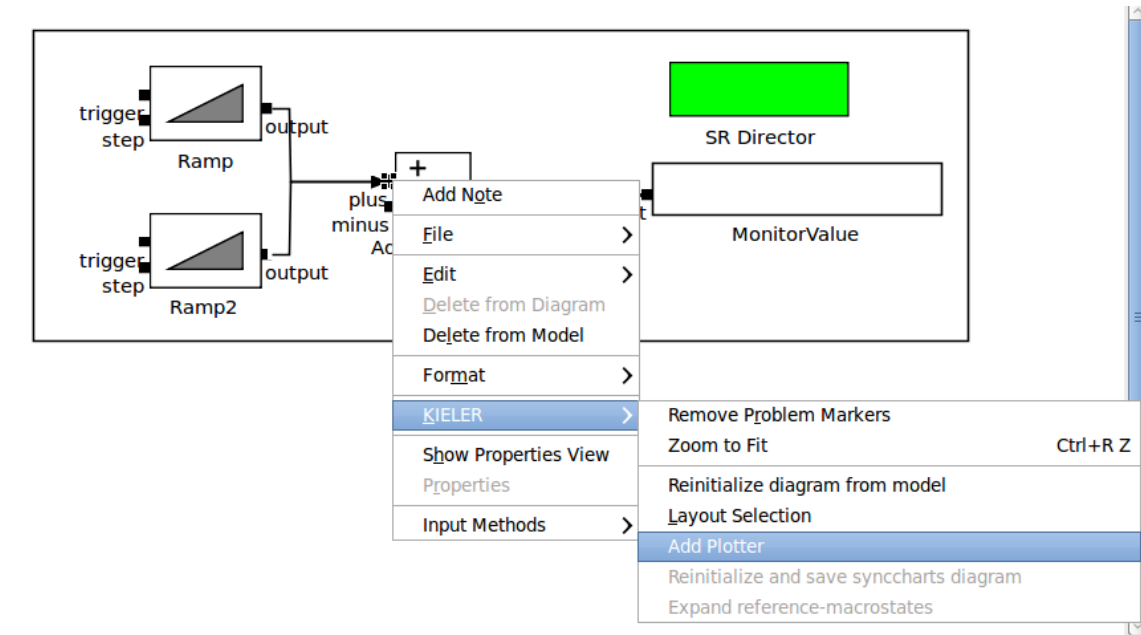


Abbildung A.2.: Hinzufügen eines Diagrammelements

beispielsweise ein Modellelement aus, klickt mit der rechten Maustaste darauf und wählt dann im KIELER Menü den Menüpunkt *Add Plotter* aus. Die Menüstruktur und der entsprechende Menüpunkt sind in Abbildung A.2 dargestellt.

Nach Hinzufügen eines Visualisierungselements bietet es sich an, automatisches Layout auf das Modelldiagramm anzuwenden.

Möchte ein Entwickler KViD um weitere Elemente dieser Art erweitern, so kann er sich an dem in Abschnitt 5.7.1 angegebenen Beispiel orientieren.

## B. Literaturverzeichnis

- [1] Charles André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.
- [2] Bernd Baumgarten. *Petri-Netze : Grundlagen und Anwendungen*. Spektrum Akademischer Verlag, Heidelberg, 1996.
- [3] Nils Beckel. View Management for Visual Modeling. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2009. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nbe-dt.pdf>.
- [4] Stuart K. Card, Jock Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, January 1999.
- [5] Scott Dyer. A Dataflow Toolkit for Visualization. *Computer Graphics and Applications, IEEE*, 10(4):60 – 69, 1990.
- [6] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.
- [7] Esterel Technologies. *SCADE Technical Manual*, 5.1 edition, February 2006.
- [8] Jarosław Francik. Algorithm Animation using Data Flow Tracing. In Stephan Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 668–671. Springer Berlin / Heidelberg, 2002.
- [9] Hauke Fuhrmann and Reinhard von Hanxleden. On the Pragmatics of Model-Based Design. In Christine Choppy and Oleg Sokolsky, editors, *Foundations of Computer Software. Future Trends and Techniques for Development*, volume 6028 of *Lecture Notes in Computer Science*, pages 116–140. Springer Berlin / Heidelberg, 2010.
- [10] Hauke Fuhrmann and Reinhard von Hanxleden. Taming Graphical Modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, LNCS, Oslo, Norway, October 2010. Springer.

## B. Literaturverzeichnis

- [11] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [12] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
- [13] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
- [14] Stephan Knauer. KEV – KIELER Environment Visualization – Beschreibung einer Zuordnung von Simulationsdaten und SVG-Graphiken. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2010.
- [15] Edward A. Lee. Model-driven development - from object-oriented design to actor-oriented design. Extended abstract of an invited presentation at Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop) Chicago, September 2003. <http://ptolemy.eecs.berkeley.edu/publications/papers/03/MontereyWorkshopLee/>.
- [16] Jeff Magee, Nat Pryce, Dimitra Giannakopoulou, and Jeff Kramer. Graphical Animation of Behavior Models. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 499–508, 2000.
- [17] MathWorks, Inc. *Simulink 7 User's Guide*. The Mathworks, Inc., 3 Apple Hill Drive, Natick, MA, October 2008. [http://www.mathworks.com/access/helpdesk/help/pdf\\_doc/simulink/sl\\_using.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf).
- [18] Tony Modica, Enrico Biermann, and Claudia Ermel. An ECLIPSE Framework for Rapid Development of Rich-featured GEF Editors based on EMF Models. In *GI-Workshop on Methodological Development of Modelling Tools*, 2009.
- [19] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse development using the graphical editing framework and the eclipse modeling framework*. IBM Corp., Riverton, NJ, USA, 2004.
- [20] Christian Motika. Semantics and execution of domain specific models—KlePto and an execution framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf>.
- [21] Ricardo Orosco and Roberto Moriyón. Reducing the Effort of building Object-oriented Visualizations. In *The Twenty-Second Annual International Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings.*, pages 568–573, 1998.

- [22] Arne Schipper. Layout and Visual Comparison of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, December 2008. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf>.
- [23] Buntarou Shizuki, Etsuya Shibayama, and Masashi Toyoda. Static Visualization of dynamic Data Flow Visual Program Execution. In *Sixth International Conference on Information Visualisation, 2002. Proceedings.*, pages 713 – 718, 2002.