

**Entwicklung und Implementierung  
eines X11 (OSF/Motif) -basierten  
Auswertungs- und Visualisierungssystems für  
QSDL-Simulatoren**

**Diplomarbeit DII**

Vorgelegt dem Fachbereich Wirtschaftswissenschaften  
der  
**Universität Gesamthochschule Essen**

von  
Rodemeyer, Christian  
Goldregenstraße 28  
45770 Marl  
Matr.-Nr. 888086

---

Erstgutachter: Prof. Dr. B. Müller-Clostermann  
Zweitgutachter: Prof. Dr. Dr. J. Hansohm

Eingereicht im Wintersemester 1995/96, 11. Studiensemester  
Voraussichtlicher Studienabschluß im Wintersemester 1995/96

# Inhaltsverzeichnis

<b>1 Einleitung .....</b>	<b>4</b>
1.1 Gegenstand und Motivation .....	4
1.2 Aufbau der Arbeit .....	5
<b>2 Spezifikation und Leistungsbewertung mit QSDL .....</b>	<b>6</b>
2.1 Spezifikation und Leistungsbewertung .....	6
2.2 Zusammenhang zwischen SDL und QSDL .....	7
2.3 Methoden der Leistungsbewertung.....	8
<b>3 Konzeption der PEV-Komponenten.....</b>	<b>10</b>
3.1 Bestandteile des QSDL-Simulationspakets .....	10
3.2 Konzept der Simulationsumgebung .....	11
3.3 Ereignisspuranalyse als Grundlage der Auswertung .....	15
3.4 Integration in den QSDL-Simulator .....	17
3.5 Aufgaben und Fähigkeiten der PEV-Komponenten .....	19
3.5.1 Auswertung.....	19
3.5.2 Visualisierung.....	21
3.5.3 Berichtsgenerierung.....	25
3.5.4 Interaktive Steuerung.....	26
<b>4 Objektorientiertes Design der PEV-Komponenten .....</b>	<b>28</b>
4.1 Erläuterung der verwendeten Booch-Notation .....	28
4.2 Basisarchitektur des PEV-Systems.....	30
4.3 Datentypen.....	33
4.4 Visualisierung.....	35
4.5 Auswertungssensoren .....	39
4.5.1 Basissensoren .....	40
4.5.2 Prozeß- und Signalorientierte Sensoren .....	42
4.5.3 Maschinen- und Requestorientierte Sensoren .....	45
4.5.4 Ereignis- und Aktivitätssensoren.....	47
4.6 Aktualisierung .....	48

---

4.7 Auswertungsberichte .....	49
<b>5 Experimentbeschreibung.....</b>	<b>50</b>
5.1 Notwendigkeit der Experimentbeschreibung .....	50
5.2 Experimentbeschreibung in C++ .....	50
5.3 Experimentbeschreibungssprache .....	52
<b>6 Abschluß und Ausblick.....</b>	<b>58</b>
<b>Abbildungsverzeichnis.....</b>	<b>60</b>
<b>Literaturverzeichnis.....</b>	<b>61</b>
<b>Eidesstattliche Versicherung.....</b>	<b>63</b>

# 1 Einleitung

## 1.1 Gegenstand und Motivation

Die Komplexität moderner Softwaresystem nimmt stetig zu. Auf Kommunikationssysteme, die inhärent parallel arbeiten, trifft diese Aussage besonders zu. Um die funktionale Korrektheit dieser Systeme zu gewährleisten wurden viele Methoden entwickelt, mit denen sich die geforderten Eigenschaften bereits während des Entwurfs überprüfen ließen. Heutzutage reicht es jedoch nicht aus, nur die Korrektheit zu gewährleisten, der Markt fordert effiziente, leistungsfähige Systeme.

Nach Klar [Klar] werden in der Praxis hauptsächlich qualitative Eigenschaften untersucht, obwohl für das Ziel, korrekte *und* leistungsfähige System zu entwickeln, auch quantitative Daten benötigt werden. An der Universität-GH Essen wurde daher von Prof. Dr. Müller-Clostermann und Mitarbeitern ein Spezifikations- und Simulationssystem entwickelt, welches auf SDL (*Specification and Description Language*) aufsetzt. Es erweitert diese rein funktionale Sprache um Aspekte der Leistungsbewertung und schafft so alle Voraussetzungen, um neben den qualitativen Daten auch die benötigten quantitativen Daten bereits in der Entwurfsphase zu ermitteln. Die quantitativen Daten werden durch Simulation des spezifizierten Systems gewonnen.

Der Entwurf und die Implementierung eines Systems zur Auswertung und Analyse der anfallenden Daten ist eine der Aufgaben dieser Arbeit. Die grafische Aufbereitung der ausgewerteten Daten, insbesondere die Visualisierung des dynamischen Systemverhaltens ist ein weitere Aufgabe. Die richtige Präsentationsform ist wichtig, um potentiellen Anwendern die Ergebnisse leichter vermitteln zu können und das System benutzerfreundlicher zu gestalten. Als Zielplattform war das Betriebssystem UNIX unter der grafischen Oberfläche X-Windows vorgegeben.

## 1.2 Aufbau der Arbeit

Da sich diese Arbeit konkret mit der Auswertung von in QSDL spezifizierten Systemen beschäftigt, erläutert Kapitel 2 zunächst die Grundlagen der Spezifikation und Leistungsbewertung mit QSDL. Kapitel 3 beschreibt die Konzeption des implementierten Auswertungs- und Visualisierungssystems. Dabei wird zunächst der Stand des QSDL-Simulationssystems zum Beginn dieser Arbeit geschildert. Darauf aufbauend wird ein globales Konzept einer Simulationsumgebung entwickelt. Anschließend werden die einzelnen Komponenten vorgestellt, und ihre Einbindung in das existierende System beschrieben. Die Auswertung legt sich dabei nicht auf Kommunikationsprotokolle fest, das Konzept ist auch für andere Simulatoren einsetzbar. Die tatsächliche Implementierung orientiert sich natürlich an den konkreten Bedürfnissen der QSDL-Simulation.

Kapitel 4 beschreibt das objektorientierte Design mit Hilfe von Klassendiagrammen. Das Verständnis des Programmdesigns ist notwendig, damit folgende Arbeiten die Auswertungs- und Visualisierungskomponenten um weitere Fähigkeiten ergänzen können. Kapitel 5 geht schließlich auf die Benutzung der Werkzeuge ein. Da sie eng in das bestehende Simulationssystem integriert wurden, kommt der Anwender mit ihnen nur indirekt in Kontakt, nämlich über die Beschreibung eines Experiment. Dort muß er die auszuwertenden Objekte festlegen, und beschreiben, in welcher Form die Ergebnisse präsentiert werden sollen.

## 2 Spezifikation und Leistungsbewertung mit QSDL

### 2.1 Spezifikation und Leistungsbewertung

In den ersten Phasen der Systementwicklung steht die Spezifikation. Nach den Prinzipien des Software Engineering versteht man darunter eine eindeutige, konsistente, vollständige und überprüfbare Beschreibung des Verhaltens und der Eigenschaften des zu entwickelnden Systems [Gh/Ja/Ma]. Die Spezifikation bildet sowohl den Rahmen der Implementierung, als auch die Basis für Wartung und Weiterentwicklung des Systems. Da Spezifikationsfehler die höchsten Kosten verursachen, beschäftigt sich die Informatik mit Methoden und Techniken, um die Anzahl diese Fehler zu minimieren.

Die Verwendung formaler Sprachen zur Spezifikation ist ein Mittel zur Erreichung dieses Ziels. Sie sind eindeutig und erlauben eine (teilweise automatische) Bewertung bestimmter Eigenschaften. SDL ist ein Beispiel für eine solche formale Beschreibungssprache. Bei der Bewertung kann zwischen qualitativen und quantitativen Kriterien unterschieden werden. Unter qualitativer Bewertung versteht man die Überprüfung funktionaler Korrektheit, die sogenannte Validation, und die Verifikation, daß bestimmte geforderte Eigenschaften von dem spezifizierten System erfüllt werden. Die Auswertung quantitativer Daten wird als Leistungsbewertung bezeichnet. Sie beschäftigt sich mit Fragestellungen bezüglich der Leistungsfähigkeit des Systems, z.B. mit der Ausführungsdauer bestimmter Operationen, oder, insbesondere in Kommunikationsprotokollen, die Anzahl übertragender Datenpakete pro Zeit.

Nach Klar [Klar] wird die Leistungsbewertung von der Praxis vernachlässigt, obwohl das Ziel die Entwicklung funktional korrekter *und* leistungsfähiger System ist. Auch Leistungsengpässe sollten aus den obigen Gründen bereits in der Spezifikationsphase erkannt werden. Die Entwicklung eines Werkzeugs zur Leistungsermittlung auf Basis von QSDL-Spezifikationen ist daher ein wesentlicher Bestandteil dieser Arbeit. Das folgende Kapitel beschreibt kurz die Spezifikationssprache SDL, sowie die zur Leistungsbewertung notwendigen Erweiterungen, die zu QSDL führen.

## 2.2 Zusammenhang zwischen SDL und QSDL

Mit der Sprache SDL (*Specification and Description Language*) läßt sich das Verhalten und die Struktur von Systemen, insbesondere Kommunikationssystemen, formal beschreiben. Durch die Empfehlung Z.100 des CCITT [CCITT] hat SDL als Standard der Telekommunikationsindustrie große Bedeutung erlangt.

Ein mit SDL beschriebenes System besteht aus einer Menge von nebenläufigen Prozessen, die über festgelegte Verbindungswege Signale austauschen und so eine Kommunikation realisieren. Signale können als Parameter weitere Daten enthalten, die zwischen den Prozessen transportiert werden. SDL-Prozesse basieren auf erweiterten endlichen Automaten, welche mit internen Variablen arbeiten und dadurch ihre Zustandsmenge im Vergleich zu den nicht erweiterten Automaten stark reduzieren können. Jedem Prozeß ist eine Signalwarteschlange zugeordnet, in der die Signale bis zu ihrer Verarbeitung gepuffert werden. Dadurch wird eine asynchrone Kommunikation zwischen den Prozessen möglich.

SDL-Modelle können durch ihre formale Beschreibung computergestützt auf ihre funktionale Korrektheit hin überprüft werden. Dies beinhaltet sowohl die Validation, daß Situationen wie z.B. Deadlocks oder unerreichbarer Code nicht auftreten, als auch die Verifikation, ob ein geforderter Dienst erbracht wird. SDL sieht keine Konzepte zur Modellierung von zeitverbrauchenden Prozessen oder der Konkurrenz um beschränkte Ressourcen vor. Quantitative Aussagen über die Leistungsfähigkeit (z.B. Anzahl übertragender Datenpakete pro Sekunde) eines mit SDL spezifizierten Modells sind daher nur sehr eingeschränkt möglich. Beispielsweise lassen sich Timer zur Modellierung von Zeitverbräuchen einsetzen [Rühl]. Dies hat aber zwangsläufig eine Modifikation der ursprünglichen Spezifikation in der Art zur Folge, daß zwischen funktionalen und leistungsorientierten Aspekten nicht mehr klar unterschieden werden kann.

Leistungsdaten sind jedoch wichtig, um die Schwachstellen eines in Entwicklung befindlichen Systems zu erkennen und seine Realisierbarkeit in einer konkreten Hardware/Softwareumgebung einschätzen zu können. QSDL (*Queueing-SDL*) erweitert SDL um die notwendigen Konzepte, die solche quantitative Aussagen ermöglichen. Für

ein mit QSDL spezifiziertes Modell ist eine Leistungsbewertung bereits vor der Implementierung mit Hilfe entsprechender Werkzeuge nur aufgrund der Spezifikation möglich.

QSDL führt das Konzept von Last und Maschine ein. Dies soll an einem Beispiel erläutert werden. Viele Protokolle komprimieren Daten, bevor sie über ein Medium gesendet werden. Die Kompression und anschließende Dekompression erledigt ein Prozessor, der dazu eine bestimmte Anzahl von Instruktionen oder Maschinenbefehlen (die Software) ausführt. Die Zeit, die für diesen Vorgang benötigt wird, ist abhängig von der Anzahl der Instruktionen und der Geschwindigkeit des Prozessors, d.h. der Zeit, die er zur Abarbeitung eines Befehls benötigt. Unter Last versteht man nun eine Maßeinheit für eine Arbeitsmenge, beispielsweise die erwähnte Anzahl von Instruktionen. Diese Größe enthält keinerlei Zeitaspekte. Die Zeit wird erst dann bekannt, wenn Last an eine Maschine (CPU) mit einer konkreten Geschwindigkeit gebunden wird. QSDL-Prozesse verbrauchen Zeit, indem sie Last an eine konkrete Maschine binden. In QSDL wird dafür der Begriff „Request“ benutzt, der an eine Maschine gesendet wird. Der sendende Prozeß ist solange blockiert, bis die Maschine die Last abgearbeitet hat. Maschinen sind den klassischen Warte-Bedien-Systemen nachempfunden. Konkurrieren mehrere Prozesse um dieselbe Maschine, kann es vorkommen, daß die Bedieneinheit bereits durch einen Auftrag belegt ist. Die ankommenden Requests oder Aufträge werden dann im Warteraum in eine Warteschlange eingereiht, aus der sie nach einer der klassischen aus der Warteschlangentheorie bekannten Bedienstrategie entnommen werden. Da der Prozeß sowohl während der Warte- als auch der eigentlichen Bedienzeit blockiert ist, kann sich die Blockierungszeit für identische Lasten situationsabhängig stark unterscheiden.

### **2.3 Methoden der Leistungsbewertung**

Die quantitativen Daten zur Leistungsbewertung lassen sich prinzipiell auf drei Arten ermitteln, nämlich durch

- 1) Messungen am realen (implementierten) System,
- 2) analytische oder numerische Betrachtungen der Spezifikation,

### 3) Messungen an einer rechnergestützten Simulation des Systems.

Methode 1) ist für das Ziel, mangelnde Leistung des Systems bereits *vor* der Implementierung zu erkennen offensichtlich nicht geeignet. Der Versuch, Leistungsdaten analytisch zu ermitteln verspricht zwar genaue und verifizierbare Ergebnisse, in der Praxis wird das benötigte mathematische Instrumentarium bei zunehmender Systemkomplexität jedoch sehr schwierig zu handhaben. Um lösbare Modelle zu erhalten müssen oft Vereinfachungen und Annahmen getroffen werden, die mit der Realität nicht übereinstimmen. Darunter leidet aber die Aussagekraft der Ergebnisse.

Um die notwendigen Daten auf dem dritten Weg, der Simulation, zu erhalten, wird zunächst ein Modell des zu untersuchenden Systems erstellt. Dieses wird von einem Rechner simuliert und das Modellverhalten beobachtet. Aus dem Verhalten des Modells kann man bei entsprechend sorgfältiger Modellierung das Verhalten des realen Systems voraussagen. Im Falle eines in QSDL spezifizierten Systems entspricht die Spezifikation weitestgehend dem Simulationsmodell, so daß der Modellierungsaufwand entsprechend gering ist. Der für 1) notwendige Aufwand der Implementierung eines Prototypen entfällt. Die von der Umwelt vorgegebenen Bedingungen, z.B. Umfang der angeforderten Dienste oder Zuverlässigkeit von Übertragungsmedien, lassen sich sehr leicht ändern, wodurch das Verhalten des Modell in verschiedenen Situationen beobachtet werden kann. Darüber hinaus ist es möglich, gezielt die relevanten Teile des Systems zu beobachten und zu modifizieren.

Diese Arbeit beschäftigt sich konkret mit der Simulation von QSDL-Modellen als Basis der Leistungsbewertung und -visualisierung. Dazu wird ein an der Universität-GH Essen in der Arbeitsgruppe von Prof. Dr. Müller-Clostermann entwickeltes Simulationspaket verwendet. Das nächste Kapitel beschreibt den Stand dieses Pakets zum Zeitpunkt dieser Arbeit und entwickelt ein Konzept, wie Leistungsbewertung und Visualisierung in den Simulator integriert werden können.

## 3 Konzeption der PEV-Komponenten

Der Name PEV steht für **P**erformance **E**valuation and **V**isualisation und beschreibt die beiden Hauptkomponenten, die in dieser Arbeit entwickelt wurden. Eine zusätzliche Anforderung war die interaktive Steuerung des Simulators und der PEV-Komponenten. Zunächst wird als Ausgangspunkt das bisher existierende QSDL-Simulationspaket beschrieben, und darauf aufbauend ein Entwurf für eine Simulationsumgebung vorgestellt. Kapitel 3.3 beschäftigt sich mit der Ereignisspuranalyse, die die Grundlage der Auswertungskomponente bildet. Die folgenden Kapitel beschreiben die konzeptionelle Einbindung der Komponenten in den bestehenden Simulator sowie ihre Aufgaben und Fähigkeiten.

### 3.1 Bestandteile des QSDL-Simulationspakets

Die QSDL Simulation ist als Simulatorgenerator auf Basis der klassischen Unix-Entwicklungsumgebung mit kommandozeilenorientierten Werkzeugen realisiert. Aus einer textuellen Spezifikation in QSDL wird ein kompletter Simulator im C++ Quellcode generiert, der anschließend noch in ein ausführbares Programm übersetzt werden muß. Dies hat den Vorteil, daß der dabei entstehende QSDL-Simulator mit der größtmöglichen Geschwindigkeit arbeitet. Demgegenüber steht der Nachteil langer Kompilationszeiten während der Entwicklungsphase, wenn die Spezifikation häufig geändert wird und jede Änderung eine Simulatorgenerierung und komplette Neuübersetzung zur Folge hat. Der Simulatorgenerator wurde von Mitarbeitern am Lehrstuhl Systemmodellierung der Universität-GH Essen entwickelt.

Der generierte Simulatorcode benutzt die Dienste der objektorientiert aufgebauten Simulationsbibliothek SCL (Simulation Class Library), die im Rahmen einer Diplomarbeit implementiert wurde [Textor]. Die Bibliothek stellt sowohl die Grundfunktionalitäten für zeitdiskrete, ereignisorientierte Simulation zur Verfügung, als auch spezielle Klassen für QSDL Konstrukte, die eine prozeßorientierte Modellierung ermöglichen. Das zugrundeliegende Simulationsparadigma geht von diskreten Zuständen aus, in denen sich das simulierte System befinden kann. Zu

diskreten Zeitpunkten kann eine Zustandsänderung stattfinden. Dieser Zustandswechsel wird als Ereignis bezeichnet (obwohl genaugenommen ein Ereignis die Ursache eines Zustandswechsels ist). Die QSDL-spezifischen Klassen umfassen Maschinen, Prozesse, Timer, Signale, Requests und die dazugehörige Warteschlangenverwaltung. Dadurch bleibt der generierte Code auf einem sehr hohen Abstraktionsniveau, die ursprüngliche Spezifikation läßt sich problemlos wiedererkennen.

Die SCL enthält eine Schnittstelle, an die alle Simulationsereignisse während der Laufzeit des Simulators weitergegeben werden. Die Ereignisse befinden sich größtenteils dem Abstraktionsniveau von QSDL, beispielsweise die Ankunft von Requests an Maschinen oder Änderung von Prozeßzuständen. Wichtig ist, daß die QSDL-Objekte als Ereignisparameter ebenfalls übermittelt werden. Diese Objekte haben wiederum Attribute, wie z.B. Namen, auf die zugegriffen werden kann. Bisher wurde diese Schnittstelle nur zur Generierung von Message-Sequence-Charts (MSCs) genutzt, sie ist aber mächtig genug, um auf ihrer Grundlage die für eine Leistungsbewertung notwendigen Daten zu ermitteln. Wie dies geschieht erläutern die folgenden Kapitel.

### **3.2 Konzept der Simulationsumgebung**

Der reine QSDL-Simulator ist für den Systementwickler noch kein geeignetes Werkzeug. Aus der Spezifikation wird zwar automatisch ein Simulationsmodell generiert, doch nach dem Start verhält sich der Simulator wie eine Black-Box auf die der Benutzer keinen Zugriff hat. Er simuliert für eine (optional als Parameter angegebene) Zeitspanne das System, erzeugt jedoch keine sichtbaren Ausgaben, und läßt auch keine Eingriffe in die Simulation zu. Der Anwender will mit dem Simulator aber sein System untersuchen, feststellen, ob das Verhalten mit seinen Erwartungen übereinstimmt. Dazu ist es nötig, daß der Anwender laufend über den Systemzustand informiert wird, er detailliert den Aggregationsgrad der ihm präsentierten Informationen, sowie die zu untersuchenden Komponenten bestimmen kann. Weiterhin sollte er die Möglichkeit haben, die Simulation in kritischen Situationen anzuhalten oder zu verlangsamen, um das Verhalten im Detail betrachten zu können. Abbildung 1 zeigt das Konzept einer Simulationsumgebung, das diese Ziele widerspiegelt.

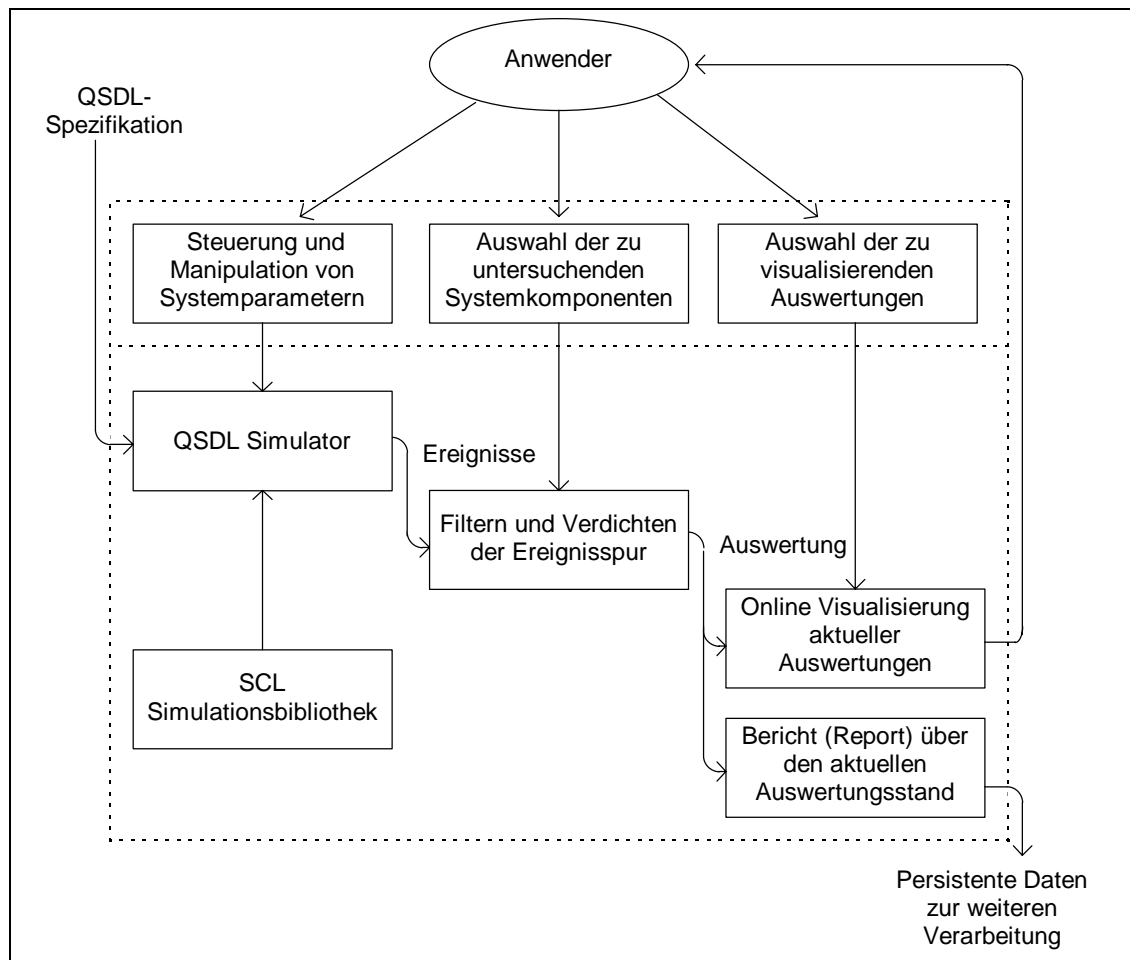


Abbildung 1: Konzept der Simulationsumgebung

Der gestrichelte Block symbolisiert die Simulationsumgebung. Der primärer Input ist eine QSDL-Spezifikation, die automatisch in einen ausführbaren QSDL-Simulator umgewandelt wird. Idealerweise werden dem Anwender Werkzeuge zur Spezifikation angeboten, die über die Fähigkeiten eines einfachen Texteditor hinausgehen. Das Tool SDT ist ein gutes Beispiel dafür, wie Spezifikationen grafisch erstellt werden können. Der eigentliche Simulator benutzt die von der SCL zur Verfügung gestellten Dienste und produziert dabei eine Menge von Simulationsereignissen, z.B. die Ankunft eines Signals an einem Prozeß. In komplexen Systemen treten Millionen dieser Ereignisse auf, so daß sie gefiltert und verdichtet werden müssen, um sinnvolle und verständliche Auswertungen zu erhalten.

Auswertungen können sich auf die gesamte Simulationszeit oder ein kurzes Intervall beziehen. Als Beispiel sei die durchschnittliche Warteschlangenlänge vor einer Maschine genannt: langfristig gesehen kann die Warteschlangenlänge gegen einen

konstanten Wert streben (wenn sich das System in einem stationären Zustand befindet), das Mittel über die letzten 100 Zeiteinheiten kann aber stark davon abweichen. Die Kenntnis beider Werte gibt Auskunft über das dynamische Verhalten des Systems. Die langfristigen Auswertungsmaße streben (idealerweise) gegen konstante Werte, während die kurzfristigen stärker schwanken. Die aktuellen Auswertungen werden daher bereits während der Simulation online in Form von Grafiken visualisiert, um dem Anwender die Dynamik des Systems offenzulegen. Mögliche Visualisierungen sind z.B. Kurven- oder Balkendiagramme. Ein wichtiger Grund für die grafische Darstellung ist deren bessere Überschaubarkeit. Es ist verblüffend, wie viele Spezifikationsfehler oder Anomalien durch bloßes Betrachten von Grafiken offensichtlich werden, die in einer reinen Zahlenkolonne kaum auffallen würden.

Die grafische Darstellung kann immer nur einen Überblick über ein begrenztes Zeitintervall geben, der sich zudem im Zeitablauf verändert. Um „harte Zahlen“ zu erhalten muß es deshalb möglich sein, die ermittelten Daten permanent festzuhalten. Dies sollte zu jedem beliebigen Zeitpunkt möglich sein. Die ausgegebenen Daten müssen mit anderen Programmen weiterverarbeitet werden können, um sie z.B. in Präsentationen oder Statistikprogramme zu übernehmen. Da im System aufgrund der hohen Datenmenge nicht alle Auswertungen für jeden Simulationszeitpunkt gespeichert werden können (sondern nur für die visualisierte Zeitspanne), ist ein Bericht über den aktuellen Stand der Auswertung ausreichend.

Der obere Abschnitt der Abbildung 1 stellt den starken Einfluß dar, den der Anwender auf das Simulationssystem haben muß. Meist interessieren ihn nur einige wenige Ausschnitte des Systems. Es muß ihm daher möglich sein, die zu untersuchenden Systemkomponenten und die darauf anzuwendenden Auswertungen gezielt auszuwählen. Idealerweise geschieht die Auswahl interaktiv, beispielsweise werden die verfügbaren Systemkomponenten in Form hierarchischer Listen angeboten, aus denen der Anwender die gewünschten Elemente z.B. durch Anklicken mit der Maus auswählt. Für jede gewählte Komponente lassen sich auf die gleiche Art und Weise die durchzuführenden Auswertungen festlegen. Die Auswahl zu untersuchender Komponenten sollte auch während der Simulation möglich sein, um z.B. gezielt nach den Ursachen für ein unerwartetes Systemverhalten suchen zu können.

Da sich auf dem Bildschirm nur eine begrenzte Informationsmenge anzeigen läßt, muß der Anwender diejenigen Auswertungen bestimmen können, welche visualisiert werden sollen. Nur er weiß, welche Auswertungen zu welchem Zeitpunkt relevant sind. Dieser Vorgang sollte ebenfalls interaktiv und während der Simulation möglich sein.

Die oben beschriebenen Eingriffsmöglichkeiten in die Auswertungs- und Visualisierungskomponenten führen dazu, auch für den Simulator an sich eine interaktive Steuerung zuzulassen. Dies befähigt den Anwender, den Simulationslauf jederzeit anzuhalten oder fortzusetzen, sowie die Geschwindigkeit des Simulators (bzw. der Visualisierung) einzustellen. Dadurch hat er die Chance, das Systemverhalten in kritischen Abschnitten detailliert zu betrachten. In diesem Zusammenhang sollte auch eine Schnittstelle zu den Systemparametern angeboten werden. Darunter werden Variablen verstanden, die z.B. die Fehlerwahrscheinlichkeit auf einem Übertragungsmedium oder die Last, die an ein System angelegt wird, bestimmen. Der Benutzer ändert diese exogenen Variablen, um das System gezielt in kritische zu Situationen steuern, und das Verhalten in diesen Situationen zu studieren.

Der Visualisierungsaspekt und die starke Interaktion mit dem Benutzer legen es nahe, die Simulationsumgebung unter einer grafischen Benutzeroberfläche zu implementieren. Aufgrund der bereits erfolgten Implementierung des Simulatorekerns unter dem Betriebssystem UNIX bietet sich das X-Windows System vom M.I.T. an [Jones]. Dieses stellt die Grundfunktionen einer grafischen Benutzeroberfläche mit Fenstertechnik zur Verfügung. Als Spezifikation für das Verhalten der Benutzeroberfläche wird OSF/Motif verwendet. Es basiert bei der Benutzerinteraktion auf den gleichen Prinzipien, die auch die Grundlage von Microsoft Windows oder dem Presentation Manager von OS/2 bilden. Dadurch kann die Bedienung von einem großen Anwenderkreis sofort verstanden werden.

Der Schwerpunkt dieser Arbeit lag auf der Realisierung der Auswertungs und Visualisierungskomponente. Diese wurden vollständig entworfen, implementiert und in den bestehenden Simulator integriert. Für eine vollständige Ausarbeitung (und Implementierung) der Komponenten, die die Benutzerinteraktion betreffen, reichten die drei Monate einer Diplomarbeit nicht aus. Trotzdem wurde eine rudimentäre, interaktive

Steuerung implementiert, und eine Möglichkeit geschaffen, die auszuwertenden Komponenten und Visualisierungen in Form von Konfigurationsdateien festzulegen. Die sicherlich noch notwendigen Ausgestaltungen bleiben weiteren Arbeiten vorbehalten.

### **3.3 Ereignisspuranalyse als Grundlage der Auswertung**

Messungen liefern die zur Leistungsbewertung benötigten Daten. Die rechnergestützte Simulation läßt prinzipiell Messungen an beliebigen Stellen des Simulationsmodells zu, und dies mit einem, im Gegensatz zu physischen Modellen, sehr geringen Aufwand. Die Festlegung der Meßstellen kann z.B. durch Instrumentierung geschehen. Dies bedeutet, daß in die Spezifikation des Simulationsmodells Anweisungen eingefügt werden, die die gewünschten Daten ermitteln und eventuell bereits eine erste Auswertung vornehmen. Im Falle des hier betrachteten Simulationssystems wäre eine Instrumentierung sowohl auf Ebene der QSDL-Spezifikation wie auf der des generierten C++-Quellcodes denkbar. Der große Vorteile der Instrumentierung ist in der problemorientierten und quellbezogenen Auswertung zu sehen (vgl. [Klar] Kapitel 6): der Anwender kann den Abstraktionsgrad der Auswertung frei bestimmen und dabei mit den gleichen Symbolen und Begriffen arbeiten, die bereits in der Systemspezifikation zur Anwendung gekommen sind.

Diesem Vorteil stehen aber einige Nachteile gegenüber. Durch das Einfügen zusätzlicher Anweisungen erhöht sich einerseits die Komplexität der Spezifikation und damit auch die Wahrscheinlichkeit Fehler zu begehen, andererseits wird die Auswertung mit der Systemspezifikation vermischt. Jedes neue Experiment mit dem Simulationsmodell führt somit zu einer Änderung der Spezifikation, unter Umständen ist ohne eine sorgfältige Dokumentation nicht mehr erkennbar, welche Teile der Spezifikation das System und welche nur die Messungen beschreiben. Auch die Wiederverwendbarkeit der Auswerteroutinen in neuen, anderen Systemen wird erschwert: einerseits können sie über die gesamte Spezifikation verstreut sein, andererseits sind sie speziell für ein bestimmtes Modell und eine bestimmte Messung angepaßt sind. Aufgrund der Arbeitsweise des QSDL-Simulators ergibt sich ein weiterer

Nachteil: jede Änderung des Experimentaufbaus hat eine komplette Neugenerierung des Simulatorcodes und dessen langwierige Übersetzung zur Folge.

Eine flexiblere und allgemeinere Lösung ist die Ereignisspuranalyse. Wie schon erwähnt lassen sich die dynamischen Veränderungen im System auf Ereignisse zurückführen. Die Aufzeichnung aller Ereignisse in ihrer Entstehungsreihenfolge wird als Ereignisspur bezeichnet. Jedes Ereignis kann außer einem Zeitstempel (Eintrittszeitpunkt) noch weitere Attribute besitzen. Die interessierenden Leistungsdaten werden aus der Analyse der Ereignisspur gewonnen. Ein Problem stellt die Ermittlung der Ereignisse dar: an physischen Modellen können sie mit Sensoren gemessen werden, in einem rechnergestützten Simulator muß die Software die entsprechenden Schnittstellen bereitstellen. Wichtig ist, daß die Ereignisgenerierung für den Systemmodellierer transparent geschieht, ansonsten entstehen die gleichen Nachteile wie bei der Instrumentierung.

Wenn das Simulationssystem eine vom simulierten Modell unabhängige und standardisierte Ereignisschnittstelle bereitstellt, können Analyse- und Visualisierungswerkzeuge entwickelt werden, die nur auf Ereignisebene arbeiten. Dadurch lassen sie sich für alle denkbaren Modelle wiederverwenden. Eine von der Systemspezifikation getrennte Experimentbeschreibung wird dadurch zwingend notwendig, da die Werkzeuge über die zu sammelnden Daten und Auswertungen informiert werden müssen. Die vom QSDL-Simulator benutzte Simulationsbibliothek SCL bietet eine solche Ereignis-Schnittstelle, so daß die Ereignisspuranalyse als die beste Methode für das zu entwickelnden Auswerte- und Visualisierungssystem erscheint.

Ein Nachteil dieser Vorgehensweise ist es, daß sich die Ereignisse auf einem tieferen Abstraktionsniveau befinden können, so daß Informationen, die in der Systemspezifikation bereits bekannt sind, verloren gehen. Diese müssen in der Experimentbeschreibung dem Auswertesystem bei Bedarf erneut mitgeteilt werden. Der Niveau-Unterschied zwischen den von der SCL generierten Ereignissen und der Spezifikation ist nur sehr gering, so daß relativ wenige Informationen verloren gehen. Auf Details gehen die folgenden Kapitel ein.

Die Ereignisspur eines realistischen Systems kann leicht einige zehntausend Ereignisse pro Sekunde umfassen. Ein Analysewerkzeug muß daher die Informationen sehr stark verdichten, bevor sie dem Anwender zur endgültigen Bewertung präsentiert werden. Dabei werden folgende Phasen durchlaufen:

- Zugriff auf die Ereignisspur
- Filtern der relevanten Ereignisse
- Zusammenfassen zusammengehörender Ereignisse
- Statistische Auswertungen und Aggregation

Diese Analyseschritte könnten alle nacheinander im Batchverfahren ablaufen, wobei jedoch die gesamte Ereignisspur und die Zwischenergebnisse gespeichert werden müßten. Aufgrund der Größe der Ereignisspur ist es pragmatisch gesehen günstiger, die Analysephasen sofort zum Zeitpunkt des Ereigniseintritts zu durchlaufen, so daß nur die interessierenden Daten in aggregierter Form gespeichert werden. Programmtechnisch können die einzelnen Schritte jedoch immer noch getrennt implementiert werden. Die Online-Auswertung hat darüber hinaus den Vorteil, daß die Ergebnisse bereits während der Simulation angezeigt werden können. Der Anwender erkennt dadurch eventuelle Fehler früher und kann durch den Abbruch der Simulation Zeit sparen.

### **3.4 Integration in den QSDL-Simulator**

Die SCL enthält eine Trace-Klasse, von der im generierten Simulatorcode genau eine Instanz angelegt wird. An diese Objekt werden alle von der SCL erkennbaren Simulationsereignisse gesendet. Das 'Senden' wird durch den Aufruf einer virtuellen Funktion der Trace-Klasse mit den dazugehörigen Parametern realisiert. Die Idee, wie die Auswertungskomponente in den Ereignisstrom eingeklinkt werden kann, besteht darin, von der Trace-Klasse eine weitere Klasse abzuleiten, die die virtuellen Ereignisfunktionen redefiniert und die übergebenen Parameter an die Auswertungskomponente weiterleitet. Der bestehende Simulator muß nur dahingehend geändert werden, daß statt der Trace-Klasse eine Instanz der abgeleiteten Ereignisverteilerklasse erzeugt wird. Nach der Übersetzung des Simulatorcodes, wird der erzeugte Objektcode mit dem Objektcode der PEV-Komponenten zum ausführbaren

Simulator zusammengelinkt. Der Start des Simulators bewirkt dann automatisch die Online-Ereignisspuranalyse. Die PEV-Module sind unabhängig vom simulierten System und nehmen nur Dienste der Simulationsbibliothek SCL in Anspruch. Daher brauchen sie nur einmal übersetzt werden und liegt dann in Form einer Bibliothek vor. Die Experimentbeschreibung wird zur Laufzeit aus einer Datei gelesen.

Der Nachteil dieser Lösung liegt darin, daß über diese Schnittstelle nur Ereignisse übertragen werden können, die der SCL bekannt sind (Abbildung 2). Das gleiche gilt für die Ereignisparameter, bei denen ebenfalls nur die in der SCL definierten Klassen übergeben werden. Der generierte Simulationscode enthält aber typischerweise immer Spezialisierungen dieser Klassen. Ihre Aufgabe ist es ja, als Basis für die QSDL-Konstrukte zu dienen, wie z.B. die Klasse Prozeß. Im Simulationsmodell gibt es nun eine abgeleitete Prozeßklasse, die vielleicht noch zusätzliche lokale Attribute enthält. Auf diese neuen Attribute kann nicht zugegriffen werden. Wie jedoch noch gezeigt wird, reichen die übermittelten Informationen für eine große Zahl sinnvoller Auswertungen aus.

Prozeß:	<b>ProcessCreate, ProcessDelete, StateChange, ActionChange, SpontaneousTransition</b>
Signale:	<b>MessageSend, MessageConsume, MessageSave, MessageDrop, MessageReceive</b>
Maschinen:	<b>MachineCreate, MachineDelete</b>
Requests:	<b>RequestIssue, RequestFinish, RequestStart, RequestStop</b>
Timer:	<b>TimerSet, TimerReset, TimerFire</b>
Allgemein:	<b>SchedulerInit, SchedulerStop, SimulationEnd, TimeChange</b>

Abbildung 2: Ereignisse und Klassen der SCL-Schnittstelle

Die PEV-Komponenten sollen möglichst unabhängig von den Simulatorkomponenten arbeiten, um eine getrennte Entwicklung und Wartung zu erlauben. Es bietet sich daher an, die Ereignisschnittstelle der SCL auch zur Ansteuerung der Visualisierungs- und Steuerungskomponente zu verwenden. Dies läßt sich dadurch realisieren, daß bei jedem Simulationsereignis zuerst eine Routine angesprungen wird, die gegebenenfalls Fensterdarstellungen aktualisiert und eventuelle Benutzereingaben verarbeitet. Dies

ermöglicht eine quasi parallele Ausführung aller drei Komponenten auf der Basis des kooperativen Multitaskings, so daß die Programme für eine echte parallele Ausführung bereits vorbereitet sind.

### **3.5 Aufgaben und Fähigkeiten der PEV-Komponenten**

In Kapitel 3.2 wurden allgemein die Komponenten vorgestellt, die zur vollständigen Verwirklichung der Simulationsumgebung benötigt werden. Dieses Kapitel beschreibt den Leistungsumfang der in dieser Arbeit entwickelten Komponenten. Diese umfassen schwerpunktmäßig die Auswertung auf Basis von SCL-Ereignissen und die Visualisierung der Auswertungsergebnisse. Daneben existiert eine interaktive Steuerung und die Möglichkeit der Berichtsgenerierung. Der Experimentbeschreibung ist das Kapitel 5 gewidmet.

#### **3.5.1 Auswertung**

Bei Experimenten mit physischen Modellen werden Daten durch Meßfühler ermittelt, die an den interessierenden Stellen in das Modell eingesetzt werden. Die Auswertungskomponente nimmt diese Vorgehensweise zum Vorbild und implementiert eine Menge von Sensoren. Genauso wie es unterschiedliche Meßfühler gibt, gibt es für jeden speziellen Auswertungsdatentyp einen entsprechenden Sensor. Der Anwender entscheidet, an welchen Stellen die Sensoren im simulierten System plaziert werden. Im Gegensatz zur physischen Meßsonde übernimmt der Softwaresensor auch Analysefunktionen. Dazu filtert er die Ereignisspur und faßt sie gegebenenfalls zu Ereignissen höherer Ordnung zusammen. Neben dem aktuellen Wert können Sensoren somit auch statistische Daten liefern. Als positiver Nebeneffekt brauchen dadurch nicht alle erfaßten Daten gespeichert werden, sondern nur deren aggregierte Form.

Nach der Art der statistischen Aufbereitung lassen sich drei Klassen von Sensoren unterscheiden, wobei einige Sensoren auch zu mehreren Klassen gehören können:

1. Die erfaßten Meßwerte lassen sich als Stichprobe begreifen. Dann stehen die folgenden statistischen Daten (bezogen auf alle Ereignisse) zur Verfügung:

- Minimum
  - Maximum
  - Mittelwert
  - Varianz
  - Standardabweichung
  - Mittelwert über ein festes Zeitintervall
2. Die Häufigkeit eines Ereignisses wird gezählt (z.B. Übertragungsfehler pro Sekunde):
- Anzahl Ereignisse insgesamt
  - Durchschnittliche Anzahl Ereignisse pro Zeit
  - Durchschnittliche Anzahl Ereignisse in einem festen Zeitintervall
3. Die Häufigkeit mehrerer Ereignisse wird gezählt. Die relative Häufigkeit in Bezug auf die Gesamtheit aller Ereignisse kann ausgegeben werden.

Natürlich kann nicht für jede denkbare Auswertung ein passender Sensor zur Verfügung gestellt werden, aber durch eine entsprechende Parametrisierung der Sensorklassen können doch viele der Auswertungswünsche erfüllt werden. Durch den objektorientierten Ansatz ist es aber auch möglich, die Implementierung um sehr spezielle, auf das konkrete simulierte Modell abgestimmte Sensorklassen zu ergänzen. Die abgeleiteten Klassen können dabei bereits bestehende Funktionalitäten der Basissensoren weiter nutzen, wie z.B. Statistik und Visualisierung. Im folgenden werden die zur Verfügung stehenden Sensorklassen aufgezählt.

- Warteschlangenlänge vor Prozessen und Maschinen
- Verteilung der Warteschlangenlänge (wie groß ist die relative Häufigkeit einer Warteschlangenlänge von  $n$ ), anwendbar auf Prozesse und Maschinen
- Wartezeit von Signalen, ohne oder mit Unterscheidung nach dem Signaltyp
- Wartezeit von Requests, ohne oder mit Unterscheidung nach dem Requesttyp
- Durchlaufzeit (Warte- und Bedienzeit) für Requests, ohne oder mit Unterscheidung nach dem Requesttyp
- Relative Häufigkeit der von einem Prozeß empfangenen Signale
- Relative Häufigkeit von einem Prozeß gesendeter Signale

- Relative Häufigkeit von einem Prozeß gesendeter Requests
- Relative Häufigkeit der von einer Maschine empfangenen Requests
- Globale Signal- und Requesthäufigkeit im gesamten System
- Zustandshäufigkeit: die Wahrscheinlichkeit, das sich ein Prozeß zu einem zufälligen Zeitpunkt im Zustand  $X$  befindet
- Maschinenauslastung: die Wahrscheinlichkeit, das eine Maschine einen Request bearbeitet

Eine weitere sehr flexible Auswerteklasse bilden die Ereignis- und Aktivitätssensorien. Eine Aktivität wird durch ein beginnendes und ein beendendes Ereignis definiert und besitzt üblicherweise eine von Null verschiedene Dauer. Für diese Sensoren ist ein Ereignis die Ankunft oder das Senden eines Signals bzw. Request an einen Prozeß oder eine Maschine. Der Signal- und Requesttyp sowie der Maschinen- und Prozeßname ist parametrisiert. Der Zugriff auf Signalparameter ist leider nicht möglich, da diese in der Spezifikation enthaltenen Informationen über die SCL nicht ermittelt werden können. Es existieren aber noch weiter spezialisierte Sensoren, denen über die Experimentbeschreibung die fehlenden Daten mitgeteilt werden können. Beispielsweise könnte die Übermittlung eines Signals mit der Übertragung eines Datenpakets bestimmter Länge assoziiert sein, und damit die Datenübertragungsrates ermittelt werden.

Ein Ereignissensor kann zur Ermittlung der Ereignishäufigkeit und der Zeitdauer zwischen den Ereignissen herangezogen werden. Ein Aktivitätssensor ermittelt die Aktivitätshäufigkeit und die Aktivitätsdauer.

### 3.5.2 Visualisierung

Die von den Sensoren ermittelten Ergebnisse müssen dem Anwender in angemessener Form präsentiert werden. Eine wesentliche Anforderung an diese Arbeit war es, dabei auch das dynamische Verhalten des simulierten Systems zu visualisieren, d.h. die Veränderung der Meßwerte im Ablauf der Zeit. Eine grafische Lösung ist in diesem Fall einer textuellen überlegen: es können mehr Informationen übersichtlicher dargestellt werden. Wenn z.B. nur die Veränderung in einem gleitenden Zeitintervall mit 100 Meßwerten textuell dargestellt werden sollte, würde diese eine Spalte von 100 Zahlen

bedeuten, die sich  $n$ -mal pro Sekunde ändern. In der grafischen Version bilden die 100 Meßwerte eine Kurve, die einen guten Überblick über das dynamische Verhalten gibt. Während der Simulation ist die Kurve animiert, d.h. der betrachtete Zeitausschnitt wandert mit der verstreichenden Simulationszeit. Es ist bemerkenswert, wie viele Spezifikationsfehler bereits durch das bloße Betrachten von Grafiken entdeckt werden können, ohne aufwendige Analyse und Validationsverfahren anwenden zu müssen. Abbildung 3 zeigt ein typisches Visualisierungsszenario.

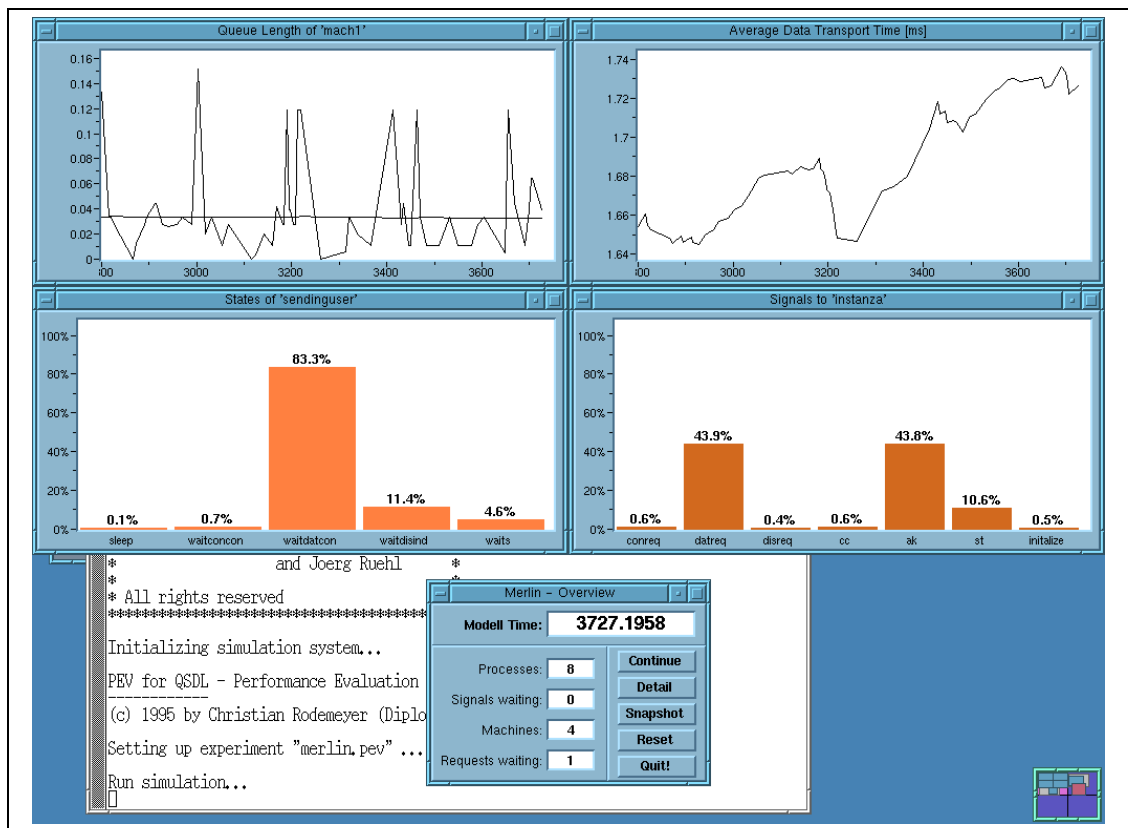


Abbildung 3: Typische Visualisierung eines Simulationslaufes

Bei der Realisierung der Visualisierungskomponente wurde auf größtmögliche Flexibilität Wert gelegt. Es können im Prinzip beliebig viele Fenster geöffnet werden. Jedes Fenster enthält ein Diagramm, in welchem gleichzeitig die Analysedaten mehrerer Sensoren angezeigt werden können. Ein und derselbe Sensor kann seine Daten auch in verschiedenen Fenstern anzeigen, um Vergleiche mit anderen Sensoren zu ermöglichen. Die Möglichkeiten die das X-Window System zum Verschieben und Ändern der Fenstergröße anbietet, werden unterstützt. Bei einer Größenänderung wird der Fensterinhalt proportional angepaßt, ohne daß die Gesamtinformation verloren geht. Im

Gegensatz zur anderen X11 basierten Programmen, werden die Fenster beim Start nicht wahllos auf dem Bildschirm verstreut, sondern automatisch so angeordnet, daß alle gleichzeitig sichtbar sind, keine Überlappungen auftreten und die zur Verfügung stehende Bildschirmfläche optimal ausgenutzt wird.

Kurven- und Balkendiagramme sind die Basistypen der Anzeige. Kurvendiagramme visualisieren die Veränderung eines Meßwertes im Zeitverlauf. Die Dimension des Meßwertes ist an der Y-Achse, die der Zeit an der X-Achse aufgetragen. Jede Kurve besteht aus einer bestimmten Anzahl von Meßpunkten. Um den Benutzer die explizite Angabe des Wertebereichs zu ersparen, der in einem Diagramm angezeigt werden soll, kann sich das Diagramm automatisch so skalieren, daß sich die gesamte Kurve immer im sichtbaren Bereich befindet. Diese Funktion ist besonders dann nützlich, wenn sich der Wertebereich im Lauf der Simulation verschieben kann. Enthält ein Diagramm mehrere Kurven, so werden die Kurven in verschiedenen Farben gezeichnet.

Balkendiagramme werden zur Veranschaulichung relativer Häufigkeiten verwendet. Sie können daher nur zur Anzeige von Häufigkeitssensoren eingesetzt werden. Ein Häufigkeitsdiagramm kann nur Sensoren enthalten, die sich auf den gleichen Ereignistyp beziehen, z.B. Signale oder Prozeßzustände. Die Namen der Ereignistypen, die denen der Spezifikation entsprechen, werden auf der X-Achse unter den Balken dargestellt. Die Visualisierungskomponente ist durch Zusammenarbeit mit der SCL und den Sensoren in der Lage, diese Namen automatisch zu ermitteln. Der Anwender muß lediglich angeben, welche Sensoren in welchem Fenster angezeigt werden sollen. Die Höhe eines Balkens ist proportional zur relativen Häufigkeit, die auch textuell in Prozentdarstellung ausgegeben wird.

Die Visualisierungskomponente ist eng mit den Auswertungssensoren gekoppelt, um die gewünschte Online-Anzeige zu realisieren. Während die Sensoren immer nur den aktuellen Wert enthalten, soll jedoch die Wertveränderung im Zeitablauf dargestellt werden. Dazu speichern die Diagrammtypen intern eine Folge von Meßwerten. Es gibt zwei Methoden, wie die Meßwerte von den Sensoren zu den Diagrammen gelangen können, die hier als 'synchrone' und 'asynchrone' Übertragung bezeichnet werden. Bei der synchronen Übertragung werden bei jedem Zustandswechsel des Systems die Werte

aller Sensoren in den Zwischenspeicher der Visualisierungskomponente übertragen. Da nicht sichergestellt ist, daß die Zeit zwischen den Zustandswechseln in gleichmäßigen Intervallen verstreicht, kann die Anzeige dadurch ‘ruckartig’ aufgebaut werden und sich mit einer sehr ungleichmäßigen Geschwindigkeit bewegen (linkes Fenster in Abbildung 4), dafür wird aber jedes Ereignis angezeigt. Bei der asynchronen Übertragungsmethode läuft der Simulator mit den Sensoren quasi parallel mit der Darstellung in den Anzeigefenstern. In festen Zeitabständen (CPU-Zeit) werden die aktuellen Werte der Sensoren abgefragt und in den Zwischenspeicher übertragen. Dadurch erscheinen die resultierenden Kurven glatter (rechtes Fenster in Abbildung 4), da je nach eingestelltem Zeitabstand entsprechend viele Ereignisse übersprungen werden. Als positiver Nebeneffekt läuft die Simulation in diesem Modus schneller (da nur noch jedes  $n$ -te Ereignis visualisiert wird) und es kann ein größerer Zeitraum überblickt werden, die Information wird aber auch ungenauer. Daher ist es möglich, je nach Informationsbedürfnis, beide Übertragungsarten einzusetzen.

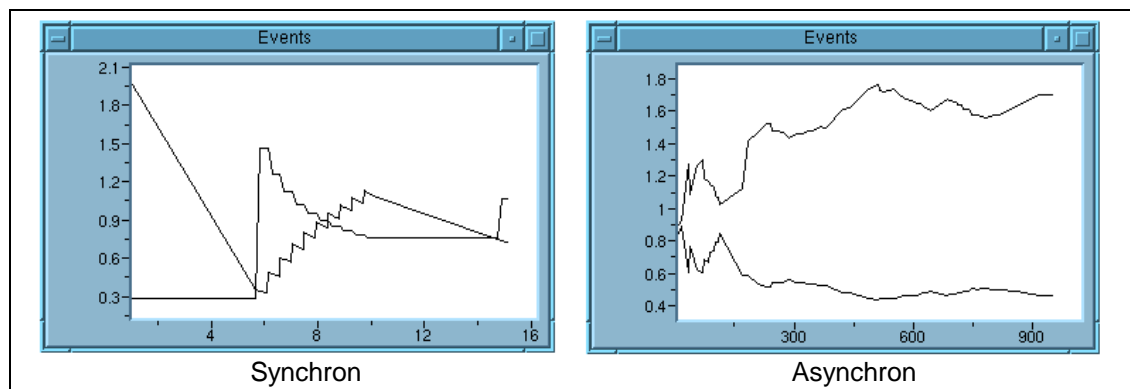


Abbildung 4: Unterschied zwischen synchroner und asynchroner Visualisierung

Eine besondere Form der Kurvendiagramme sind *Gantt-Diagramme* zur Darstellung der Prozeßzustandsänderung im Zeitverlauf (siehe Abbildung 5). Sie sind allerdings nicht mit denen aus der Betriebswirtschaftslehre bekannten, in der Produktionsplanung eingesetzten Gantt-Diagrammen identisch (vgl. [Klar]). An den hier verwendeten Diagrammform läßt sich ablesen, in welcher Reihenfolge und für welchen Zeitraum, ein Prozeß bestimmte Zustände annimmt. An der Y-Achse sind die möglichen Prozeßzustände aufgetragen, die X-Achse ist wie bei allen Kurvendiagrammen mit der Zeitachse identisch. Ein Gantt-Diagramm läßt sich nur bei synchroner Aktualisierung

sinnvoll interpretieren, da bei asynchroner Übertragung Zustandswechsel verloren gehen, und damit das Prozeßverhalten falsch beschrieben wird.

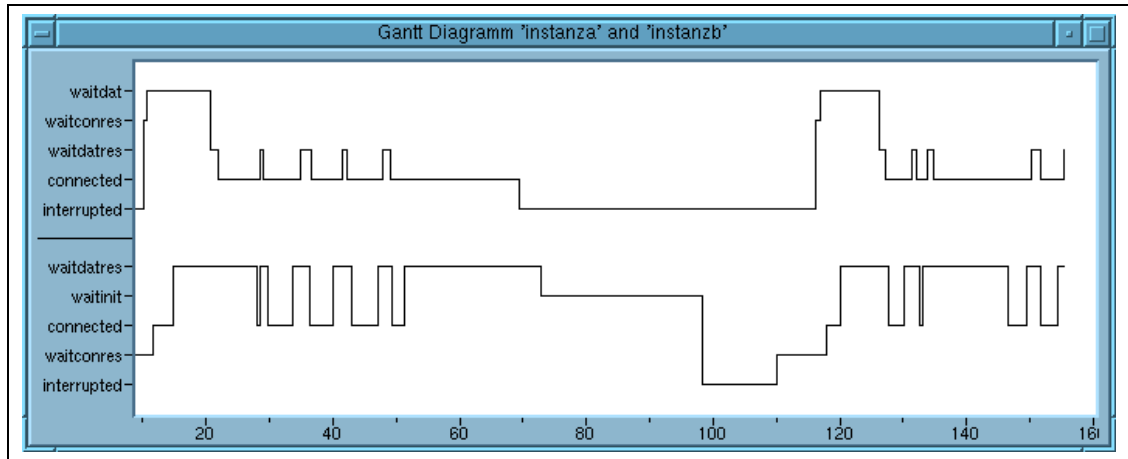


Abbildung 5: Gantt Diagramm

### 3.5.3 Berichtgenerierung

Ein Nachteil der grafischen Darstellung ist es, daß ihr keine exakten Werte entnommen werden können, und daß aufgrund der Systemdynamik immer nur ein kurzer Ausschnitt sichtbar ist. Nach einer gewissen Zeit werden die alten Daten durch neuere überschrieben. Um diesen Nachteil zu umgehen, gibt es eine Berichtskomponente, die die laufende Auswertung in einer Datei permanent sichern kann. Das Ergebnis wird als Bericht in Form eines formatierten ASCII-Textes erzeugt (siehe Abbildung 6), so daß eine Weiterverarbeitung mit den üblichen textorientierten UNIX-Werkzeugen möglich ist.

Er enthält die Auswertungen aller Sensoren, auch derjenigen, die nicht zur Online-Visualisierung vorgesehen sind. Jeder Sensor kann mehrere Auswertungen gleichzeitig durchführen. Diese lassen sich in langfristige und intervallbezogene unterteilen. Der Bericht enthält nur die langfristigen Auswertungen, d.h. er spiegelt den stationären Zustand des Systems wieder. Die intervallbezogenen sind nur für die Visualisierung des aktuellen Zustands interessant und werden daher nicht ausgegeben. Andererseits enthält der Bericht auch Daten, die nicht visualisiert werden, z.B. die absoluten Häufigkeiten oder Zeitanteile von Ereignissen oder Zuständen. An dieser Stelle sei noch auf eine Besonderheit der Häufigkeitszähler hingewiesen: Wenn bestimmte Ereignisse nie

auftreten, bzw. Zustände eine Dauer von Null haben, gehen sie nicht in die Auswertung ein und erscheinen weder in der Visualisierung noch in Berichten. Obwohl dies offensichtlich erscheint, muß man sich daran erinnern, wenn die Simulation ursprünglich auf einem SDL-Modell basiert. In diesen funktionalen Modellen gibt es nämlich regelmäßig Zustände, die nur für die Dauer Null angenommen werden. Diese Zustände erscheinen selbst dann nicht, wenn sie bei der Simulation von einem Prozeß angenommen werden. Für sinnvolle Berichte, muß auch der Zeitverbrauch modelliert werden.

```

PEV-Report for experiment 'Merlin - Test' at 1062.01:
=====

State frequency of process 'instanza':
=====
Frequency of | Relative | Absolute
-----+-----+-----
interrupted  | 3.82%  | 40.60
waitconres   | 1.10%  | 11.71
waitinit     | 14.12% | 150.00
waitdatres   | 80.95% | 859.71

Event 'EvInSig':
=====
Min      Max      Avg      Var      Dev
0.24381  50.244  2.0662  25.811  5.0805

Counts = 514, CountsPerTime = 0.48399

Activity 'DatReq/-Con':
=====
Min      Max      Avg      Var      Dev
0.44952  50.693  1.7914  15.724  3.9654

Counts = 512, CountsPerTime = 0.4821

[...]

```

Abbildung 6: Textueller Analysebericht

### 3.5.4 Interaktive Steuerung

Das Fenster aus Abbildung 7 ist die Schnittstelle, über die der Anwender während der Simulation in das System eingreifen kann. Im oberen Abschnitt wird die aktuelle Zeit im simulierten Modell gezeigt. Die linke Seite enthält vier Felder, die Auskunft über die Anzahl der Prozesse und der im gesamten System wartenden Signale, sowie die Anzahl der Maschinen und der wartenden Requests geben. Diese Informationen sind nützlich, um einen groben Überblick über das System zu bekommen: findet man z.B. weniger Prozesse als erwartet, oder bemerkt man eine über alle Grenzen wachsende Warteschlangenlänge, so deutet dies auf einen schweren Fehler hin.

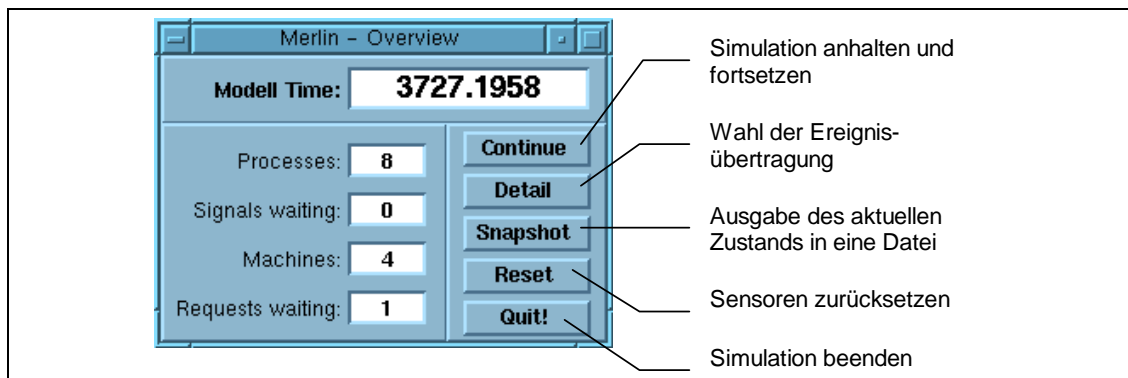


Abbildung 7: Interaktive Simulatorsteuerung

Die Schaltflächen auf der rechten Seite erlauben es, die Simulation jederzeit zu stoppen und wieder fortzusetzen. Dies ist nützlich, um z.B. die Diagrammfenster genauer zu betrachten. Hinter dem Detail (Rough) Button verbirgt sich eigentlich nur die Umschaltung zwischen der synchronen und der asynchronen Grafikaktualisierung, der sichtbare Effekt besteht jedoch in einer verlangsamten oder beschleunigten Simulationsgeschwindigkeit. Mit dem Snapshotkommando kann der im vorigen Kapitel beschriebene Bericht des aktuellen Zustands in eine Datei geschrieben werden. Die Reset-Funktion gibt allen Sensoren den Befehl, alle bisher eingetretenen Ereignisse zu „vergessen“, d.h. sie gehen nicht mehr in die Auswertung ein. Der Zustand des Simulators ändert sich dadurch aber nicht, nur die Sensoren werden beeinflusst. Der letzte Button dient zum Beenden des Simulators. Dabei wird implizit auch ein Abschlußbericht ausgegeben.

## 4 Objektorientiertes Design der PEV-Komponenten

Dieses Kapitel beschreibt die Realisierung des in Kapitel 3 vorgestellten Konzepts. Auf die Erläuterung des ca. 7000 Zeilen umfassenden C++ Quellcodes wird dabei verzichtet.<sup>1</sup> Statt dessen wird das objektorientierte Programmdesign anhand der von Grady Booch [Booch] vorgestellten Methode weitestgehend grafisch beschrieben. Dabei wird nur ein kleiner Teil der sehr umfassenden und leistungsstarken Methodik eingesetzt, nämlich die Klassendiagramme, welche im folgenden Kapitel erläutert werden. Die anschließenden Kapitel verwenden Klassendiagramme, um jeweils einen spezifischen Ausschnitt aus der Systemarchitektur zu präsentieren. Dabei werden die Aufgaben der einzelnen Klassen grob skizziert. Da es für die Interpretation der ermittelten Ergebnisse notwendig ist, die Arbeitsweise der jeweiligen Sensoren genau zu verstehen, geben die Kapitel über die Sensorklassen genaue Definition zu den ermittelten Leistungsmaßen.

### 4.1 Erläuterung der verwendeten Booch-Notation

Ein Klassendiagramm zeigt die Existenz von Klassen und ihre Beziehungen untereinander. In einem objektorientierten System lassen sich damit die Verantwortlichkeiten der Klassen und deren Struktur beschreiben. Man erhält somit eine logische Sicht auf die Architektur des Systems.

Die von dieser Arbeit benötigten Komponenten der Notation zeigt Abbildung 8. Das Symbol für eine Klasse ist ein wolkenförmiger, gestrichelter Umriß. Er enthält den unterstrichenen Namen der Klasse, sowie eine Aufzählung von Methoden und Attributen. Es ist nicht wichtig, *alle* Methoden und Attribute einer Klasse in der grafischen Notation aufzulisten, dies würde nur zu einer unübersichtlichen Darstellung führen. Methoden und Attribute sollten nur dann erscheinen, wenn sie zum Verständnis der Klassenspezifikation erforderlich sind.

---

<sup>1</sup>Der komplette Quelltext ist am Lehrstuhl „Systemmodellierung“ am FB6 Mathematik/Informatik verfügbar. Ansprechpartner ist Marc Diefenbruch.

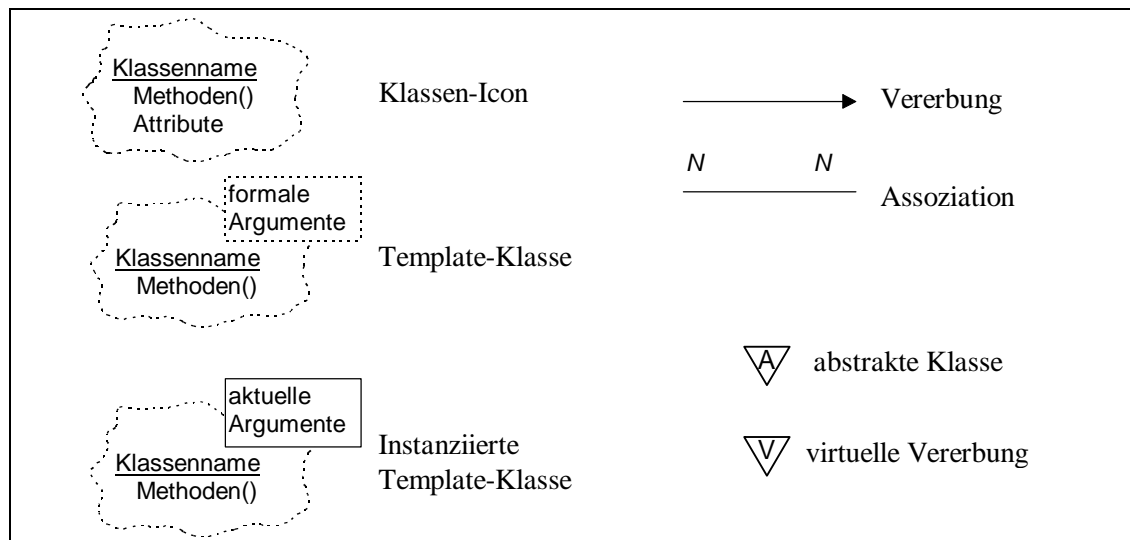


Abbildung 8: Elemente der Booch-Notation für Klassendiagramme

In C++ gibt es die Möglichkeit Template-Klassen zu definieren. Diese parametrisierten Klassen bilden „Schablonen“, aus denen durch Aktualisierung der formalen Parametern ganze Klassenfamilien gebildet werden können. Häufig werden Template-Klassen als Container-Klassen, z.B. Listen, eingesetzt. Das Template spezifiziert dann das allgemeine Verhalten der Liste, während der verwaltete Elementtyp durch Parameter beschrieben wird. In der Notation wird eine Template-Klasse durch ein zusätzliches, gestricheltes Rechteck gekennzeichnet, in dem die formalen Argumente aufgelistet werden. Der instanziierten Klasse ist ein durchgezogenes Rechteck angeheftet, das die aktuellen Argumente enthält.

In einer Softwarearchitektur existieren Klassen nicht in Isolation, sondern haben immer einen Bezug zu anderen Klassen. Zur Beschreibung werden hier nur die Vererbungsbeziehung und die einfache Assoziation benutzt. Die Vererbung wird durch einen Pfeil von der abgeleiteten zur Basisklasse dargestellt. Eine einfache Linie zwischen zwei Klasse bezeichnet eine Assoziation, die zunächst einmal nur anzeigt, daß die Klassen in irgendeiner Art und Weise zusammenarbeiten. Die Natur der Beziehung muß textuell erläutert werden. Die Kardinalität einer Assoziation kann optional an den Endpunkten der Linie angegeben werden. Die Zahlen geben an wie viele Instanzen dieser Klassen miteinander verbunden sind. *N* steht dabei für beliebig viele.

Schließlich gibt es noch das Symbol eines nach unten zeigendes Dreiecks. Enthält es ein „**A**“, so wird es zur Kennzeichnung einer abstrakten Basisklasse benutzt. Es wird dazu

in der entsprechenden Klasse plziert. Eine abstrakte Klasse enthält „pure virtual“ Methoden, die zwar die Schnittstelle (Argumentliste) definieren, deren Verhalten aber von abgeleiteten Klassen implementiert werden muß. Abstrakte Klassen können nicht instanziiert werden. Ein „V“ wird über eine Vererbungsbeziehung gelegt und gibt an, daß es sich dabei um eine virtuelle Vererbungsbeziehung handelt. Steht eine Klasse bei Mehrfachvererbung über verschiedene Pfade mit ein und derselben Basisklasse in Vererbungsbeziehung, enthält ein Objekt dieser Klasse mehrere Instanzen der Basisklasse. Die virtuelle Vererbung sorgt dafür, daß nur eine einzige Instanz der Basisklasse erzeugt wird.

## 4.2 Basisarchitektur des PEV-Systems

Das PEV-System besitzt auf zweierlei Weise eine ereignisorientierte Architektur: zum einen basiert die Auswertung auf den Simulatorereignissen, zum anderen meldet auch das X11-Fenstersystem Ereignisse, die ordnungsgemäß verarbeitet werden müssen, um die grafische Oberfläche zu realisieren. Der Simulator meldet die auftretenden Ereignisse an eine Instanz der Klasse SCTrace, indem er die LogEvent Funktion aufruft. Der genaue Ereignistyp und die Ereignisparameter sind in den LogEvent Parametern kodiert. Von der innerhalb der SCL definierten Klasse SCTrace leitet das PEV-System die PEEventDispatcher Klasse ab. Sie redefiniert die virtuelle LogEvent Funktion, um den Ereignisstrom „abzuhorchen“. Genauso wie es nur genau eine Instanz der SCTrace-Klasse geben kann, existiert auch nur ein PEEventDispatcher-Objekt. Die Vererbungsbeziehung zwischen beiden Klassen ist die einzige aktive Verbindung zwischen dem PEV-System und dem QSDL-Simulator. Weil das PEV-System nur eine untergeordnete Komponente ist, wird es vom Kontrollfluß nur dann erreicht, wenn der Simulator die LogEvent Funktion aufruft. Die Konsequenz ist, daß diese Funktion die gesamte Arbeit erledigen muß, wie z.B. Ereignisanalyse, Fensteraktualisierung, Abarbeitung ausstehender X11-Ereignise oder Reaktion auf Benutzereingaben. Vor diesem Hintergrund ist die in Abbildung 9 gezeigte Klassenarchitektur zu sehen.

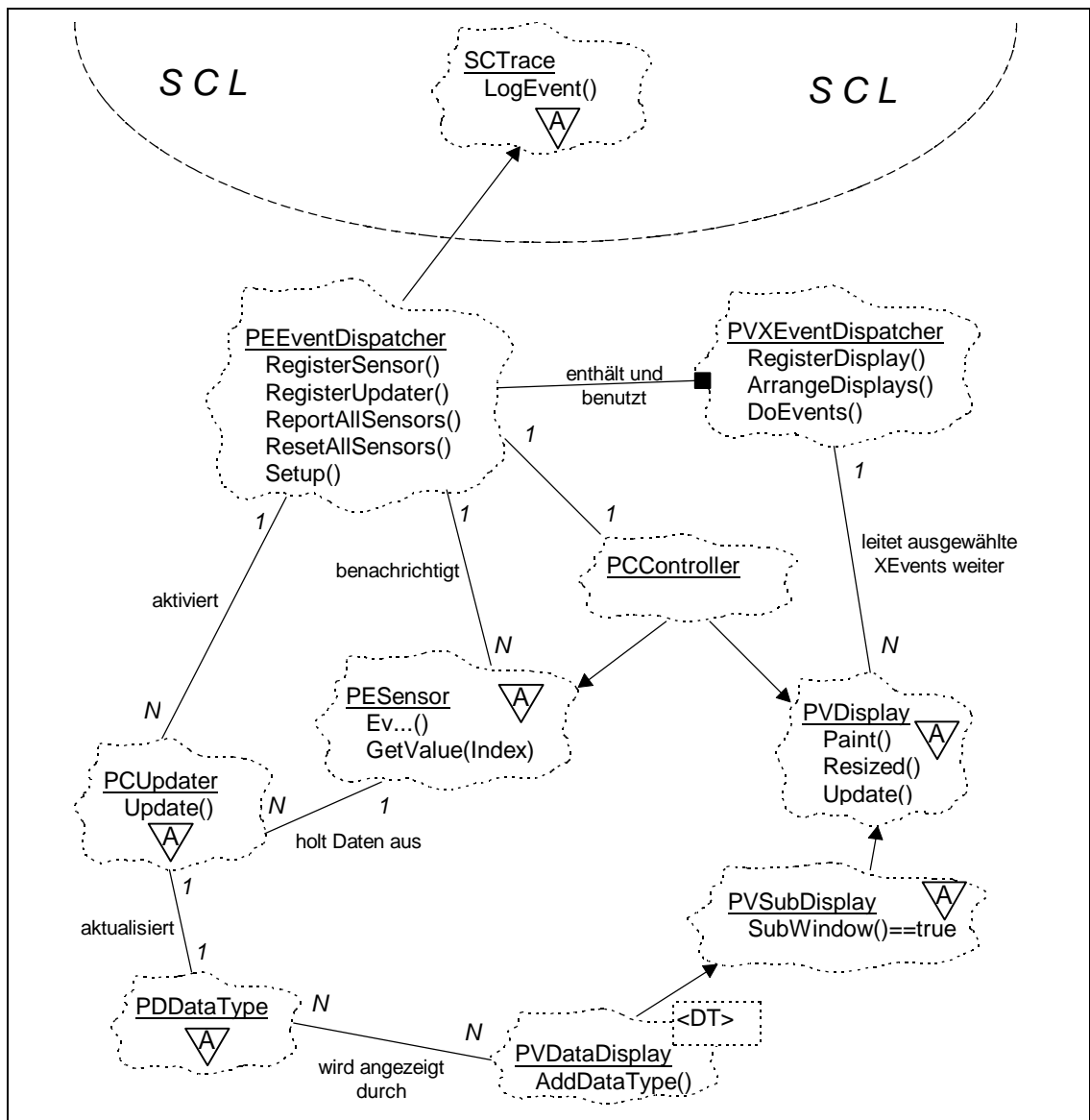


Abbildung 9: Übersicht der PEV-Architektur

**PEEventDispatcher:** Diese Klasse koordiniert die Ausführung aller PEV-Komponenten. Wenn sie der Kontrollfluß über die LogEvent-Funktion erreicht, aktiviert sie das PVXEventDispatcher Objekt, damit die anstehende X11-Ereignisse verarbeitet werden können. Die LogEvent-Parameter werden in ein aus Abbildung 2 bekanntes Ereignis dekodiert, welches dann an alle Sensoren weitergeleitet wird, die sich dafür haben registrieren lassen. Die Verwaltung aller im System existierenden Sensoren und Updater ist eine weitere Aufgabe. Wenn sich die Modellzeit ändert, wird ein Aktivierungssignal an alle Updater gesendet. Auch der assoziierte PVXEventDispatcher wird benachrichtigt, damit dieser ein Aktualisierungssignal an alle PVDisplay Instanzen

sendet. Der PEEventDispatcher kann an alle Sensoren ein Reset-Signal senden oder sie veranlassen, einen Report zu erzeugen. Die Setup() Methode ist in der Lage, eine Experimentbeschreibungsdatei zu parsen, und die benötigten Sensoren und Displays dynamisch zur Laufzeit zu erzeugen.

**PCUpdater:** Alle Instanzen dieser Klasse werden durch ein PEEventDispatcher Objekt verwaltet. Jede Instanz ist mit je einem Objekt der Klasse PESensor und PDDataType verbunden. Erhält sie ein Aktivierungssignal, fragt sie den aktuellen Zustand des zugeordneten Sensors ab und übermittelt die Daten an ein PDDataType Objekt, welches sie für ein PVDataDisplay speichert.

**PESensor:** Die Basisklasse aller Auswertungen. Alle Instanzen dieser Klasse müssen im PEEventDispatcher registriert werden. Dabei teilt das Sensor-Objekt dem EventDispatcher mit, bei welchen Ereignissen es benachrichtigt werden möchte. Aufgrund dieser Ereignisbotschaften ermittelt der Sensor die geforderten Maße. Ein Sensor kann mehrere Auswertungen gleichzeitig durchführen (z.B. Mittelwert, Minima und Maxima ermitteln). Diese Werte können via GetValue() von anderen Objekten abgefragt werden.

**PDDataType:** Ein Speicher für die von einem Sensor ermittelten Daten. Objekte dieser Klasse können auch mehrere aufeinanderfolgende Messungen speichern und so das zeitliche Verhalten festhalten.

**PVXEventDispatcher:** Genau eine Instanz dieser Klasse ist jedem PEEventDispatcher Objekt zugeordnet. Es übernimmt die Kommunikation mit dem X11-Fenstersystem. Ankommende X-Ereignisse werden der Warteschlange entnommen, bedarfsweise aggregiert, und in Form von Paint und Resize Anweisungen an die PVDisplay Objekte gesendet.

**PVDisplay:** Sämtliche PEV-Fenster basieren auf Instanzen dieser Klasse. Mit jeder Instanz ist ein X11-Fenster assoziiert, grundlegende Funktionen zur Implementierung eines OSF/Motif konformen Verhaltens und Erscheinungsbildes werden bereitgestellt. Das spezielle Verhalten, z.B. was in dem Fenster dargestellt wird, muß von abgeleiteten Klassen implementiert werden.

**PVSubDisplay:** Eine von PVSubDisplay abgeleitete Klasse, deren Instanzen mit untergeordneten Fenstern assoziiert sind. Ein Subfenster ist in einem anderen Fenster enthalten und wird z.B. mit diesem verschoben, verkleinert oder vergrößert.

**PVDataDisplay:** Ein spezialisiertes PVSubDisplay, welches ein oder mehrere assoziierte PDDataType Objekte anzeigt. Es handelt sich hierbei um eine Template-Klasse, die für verschiedene, von PVDataTyp abgeleitete Klassen instanziiert werden. Die instanziierten Klassen werden dann noch weiter spezialisiert, um eine den Datentypen angemessene Darstellung zu erreichen. Beispiele dafür zeigt Abbildung 3.

**PCController:** Diese Klasse realisiert die interaktive Benutzersteuerung, die in Abbildung 7 gezeigt wird. Sie steht in einer eins zu eins Beziehung mit der PEEventDispatcher Klasse, weswegen nur eine Instanz des PCControllers existieren kann. Ihre Aufgabe ist es, grobe Informationen über das System zu sammeln und anzuzeigen und die Anwendereingaben an den PEEventDispatcher weiterzuleiten. Der PCController ist daher sowohl ein PESensor als auch ein PVDisplay. Die Sensorfunktionalität ist notwendig, um die aktuelle Modellzeit und die Anzahl der Prozesse, Maschinen etc. zu ermitteln. Das PVDisplay wird zur Darstellung dieser Information und zur Kommunikation mit dem Anwender benötigt.

### 4.3 Datentypen

Zur Speicherung von Auswertungsergebnissen und weiteren Daten wurden die Datentypklassen entworfen. Die Aufgabe der Basisklasse **PDDataType** ist es, Polymorphismus zu ermöglichen, und bei Programmende den von den Instanzen in Anspruch genommenen Speicherplatz wieder freizugeben. Dieser Mechanismus kann durch die Funktion DisableAutoDelete() deaktiviert werden, wenn die Speicherfreigabe auf andere Art und Weise geregelt ist. Abbildung 10 zeigt das Klassendiagramm der implementierten Datentypen.

**PDPoint:** Speichert einen Meßpunkt. Das X-Element entspricht dem Zeitpunkt der Messung (Modellzeit), das Y-Element dem Meßwert.

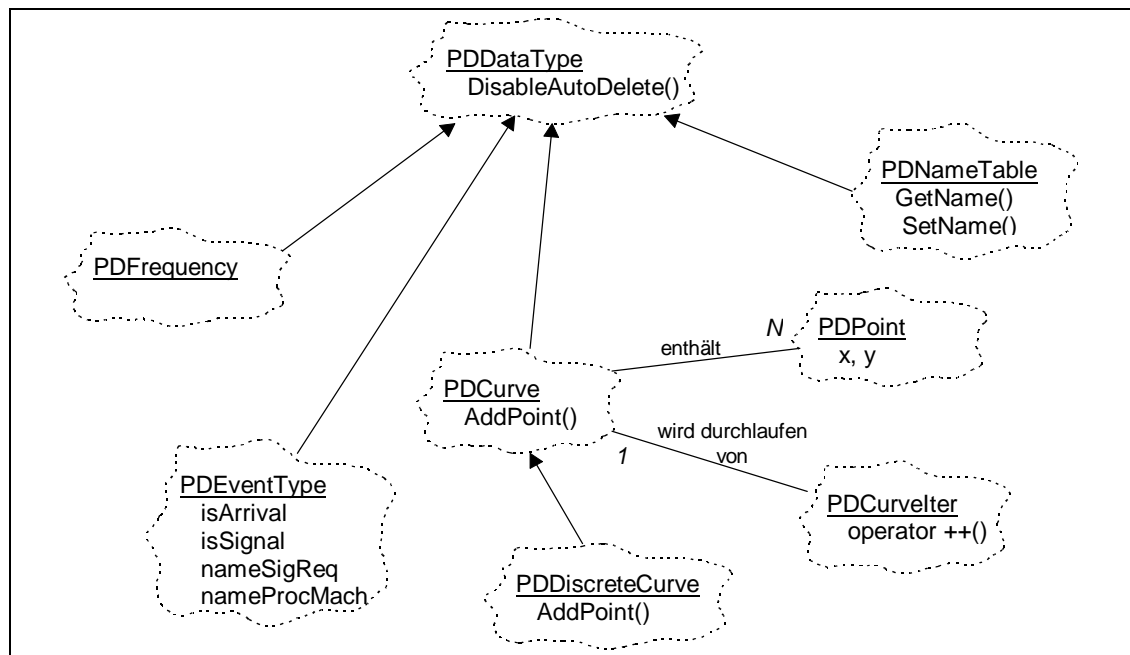


Abbildung 10: Klassendiagramm Datentypen

**PDCurve:** Zusammenfassung einer Menge von PDPoint Objekten zu einer Kurve. Die maximale Anzahl der Punkte muß bei der Konstruktion festgelegt werden. Die Methode AddPoints fügt der Kurve einen neuen Meßpunkt hinzu, wobei dieser jünger als alle in der Kurve enthaltenen Punkte sein muß. Besteht die Kurve bereits aus der maximal zulässigen Anzahl von Punkten, wird der „älteste“ Punkt entfernt.

**PDCurveIter:** Diese Klasse implementiert einen Iterator, mit dem die in einem PDCurve Objekt enthaltenen Punkte zeitlich sortiert ausgelesen werden können. Bei der Konstruktion muß der Iterator an eine konkrete PDCurve-Instanz gebunden werden, diese Beziehung bleibt über die gesamte Lebensdauer des Iterators bestehen.

**PDDiscreteCurve:** Einige Auswertungen liefern nur diskrete Werte. Würde die Anzeige nun einfach zwei Meßpunkte durch eine Linie verbinden, entsteht beim Anwender der Eindruck, daß auch alle dazwischenliegenden Werte angenommen werden, d.h. das sich die Meßwerte kontinuierlich ändern. Um diesen Effekt zu verhindern wurde die Klasse PDCurve spezialisiert: wird einem PDDiscreteCurve Objekt ein neuer Punkt hinzugefügt, wird intern ein zweiter Punkt mit identischer X-Koordinate erzeugt, die Y-Koordinate gleicht der des vorherigen Punktes. Dadurch besteht die Kurve bei der Darstellung in einem Diagramm nur aus horizontalen und vertikalen Linien, wodurch

die Visualisierung ihrer diskreten Natur unterstützt wird. Überall dort, wo PDCurve Objekte erwartet werden, können auch PDDiscreteCurve Objekte verwendet werden.

**PDFrequency:** Auswertungen über absolute und relative Häufigkeiten benutzen diese Klasse, um ihrer Ergebnisse zu speichern. Sie ist analog einem dynamischen Array implementiert, über einen numerischen Index kann das dazugehörig Ereignis angesprochen werden. Wenn der Index zum ersten mal benutzt wird, legt das PDFrequency Objekt automatisch einen neuen Zähler an, indem die absolute Häufigkeit gespeichert wird. Über den Index läßt sich dann auf den Zähler zugreifen, er kann um einen beliebigen Wert erhöht werden, und die absolute und relative Häufigkeit kann ermittelt werden. Die relative Häufigkeit  $f$  eines Ereignisses  $i$  ist definiert als

$$f_i = \frac{f_i}{f_j}$$

**PDNameTable:** Eine dynamisches Array, dessen Elemente Strings sind. Es wird dazu benutzt, den in der SCL verwendeten numerischen IDs Namen zuzuordnen. Dadurch kann der Benutzer mit den aus der Spezifikation bekannten Namen arbeiten.

**PDEventType:** Instanzen dieser Klasse werden zur Definition einfacher Ereignisse benutzt. Das boolsche Attribut „isSignal“ dient dabei zur Unterscheidung, ob sich das Ereignis auf ein Signal oder einen Request bezieht. Das Attribut „nameSigReq“ enthält dementsprechend den Signal oder Requestnamen. „isArrival“ dient zur Unterscheidung, ob das Ereignis durch die Ankunft oder das Senden eines Objekts ausgelöst wird. Maschinen können nur Requests empfangen, Prozesse können Requests und Signale senden sowie Signale empfangen. Wenn also „isSignal“ und „isArrival“ falsch sind, steht in „nameProcMach“ der Maschinenname, ansonsten der Prozessname.

## 4.4 Visualisierung

Die Visualisierung der Auswertungsergebnisse basiert auf dem X11-Fenstersystem. Eine Menge von Visualisierungsklassen (Abbildung 11) unterstützt die objektorientierte Fensterverwaltung. Die Basisklasse **PVDisplay** stellt die Verbindung zu einem X11-Fenster her. Die Fensterereignisse des X11-Systems werden von der assoziierten PVXEventDispatcher-Klasse (siehe Abbildung 9) vorverarbeitet und über die virtuellen

Funktionen `Paint()` und `Resized()` an die dazugehörigen `PVDisplay` Objekte weitergeleitet. Jede `PVDisplay` Instanz muß beim `PVXEventDispatcher` registriert sein, damit das dazugehörige Fenster auf dem Bildschirm erscheint. Die Methode `Update()` wird aufgerufen, wenn sich der Inhalt des Fensters geändert hat und neu dargestellt werden muß. Die Spezialisierung des `PVDisplays` geschieht in zwei Hauptrichtungen, die durch die abstrakten Klassen **`PVSubDisplay`** und **`PVFrameDisplay`** eingeleitet werden. Ein `PVFrameDisplay` Objekt ist ein Rahmenfenster, welches ein untergeordnetes Fenster enthält, welche über die `PVSubDisplay` Klasse angesprochen wird. Beide Klassen implementieren das Zusammenspiel von über- und untergeordnetem Fenster. Die nächste Stufe der Spezialisierung schafft die Verbindung zu den `PDDataType` Klassen, deren Instanzen die zu visualisierenden Informationen enthalten. Die Templateklassen **`PVDataDisplay<DT>`** und **`PVFrameDisplay<DT>`** ermöglichen es, jedem Display-Objekt eine beliebige Menge von Datentypen hinzuzufügen. Da jedes `PVDataDisplay` in genau einem `PVFrameDisplay` enthalten ist, braucht der Programmierer nur mit den Frame-Objekten arbeiten. Diese reichen alle relevanten Informationen automatisch an das untergeordnete `DataDisplay`-Objekt weiter. Die von `PVDataDisplay` abgeleiteten Klassen sind für die eigentliche Visualisierung in Form von Kurven- und Balkendiagrammen zuständig. Außerdem übernehmen sie die Aufgabe der Autoskalierung, d.h. der automatischen Größenanpassung, damit alle Informationen sichtbar sind. Die Spezialisierungen der `PVFrameDisplay`-Klasse sorgen für die korrekte Beschriftung der Diagrammachsen. Die auf dem Bildschirm sichtbaren Diagrammtypen werden durch die folgenden Klassen implementiert:

**`PVFreqDisplay`**: Zeigt relative Häufigkeiten in Form von Balkendiagrammen an. Die Höhe eines Balkens ist proportional zu seiner Häufigkeit. Die Breite der Balken wird automatisch so skaliert, daß alle Balken gleichzeitig sichtbar sind. Dazu wird eine Instanz der `PVMapper`-Klasse benutzt. Die Häufigkeitsinformation stammt aus einem Objekt der Klasse `PDFrequency`. Häufigkeiten von Null werden in der Darstellung ignoriert.

**`PVFreqFrame`**: Beschriftet ein Balkendiagramm mit einer Prozentskala an der Y-Achse und den Häufigkeitsnamen entlang der X-Achse. Die Häufigkeitsnamen werden als Referenz auf ein `PDNameTable` Objekt übergeben. Daraus folgt, daß bei mehreren

angezeigten Datentypen alle Häufigkeiten denselben Namen haben müssen. Die Zuordnung von Namen zu Häufigkeit geschieht über deren Index, d.h. Häufigkeit Nr. 2 gehört zu Name Nr. 2 usw. Dabei wird auch beachtet, daß das PVFreqDisplay-Subfenster nur Häufigkeiten ungleich Null darstellt.

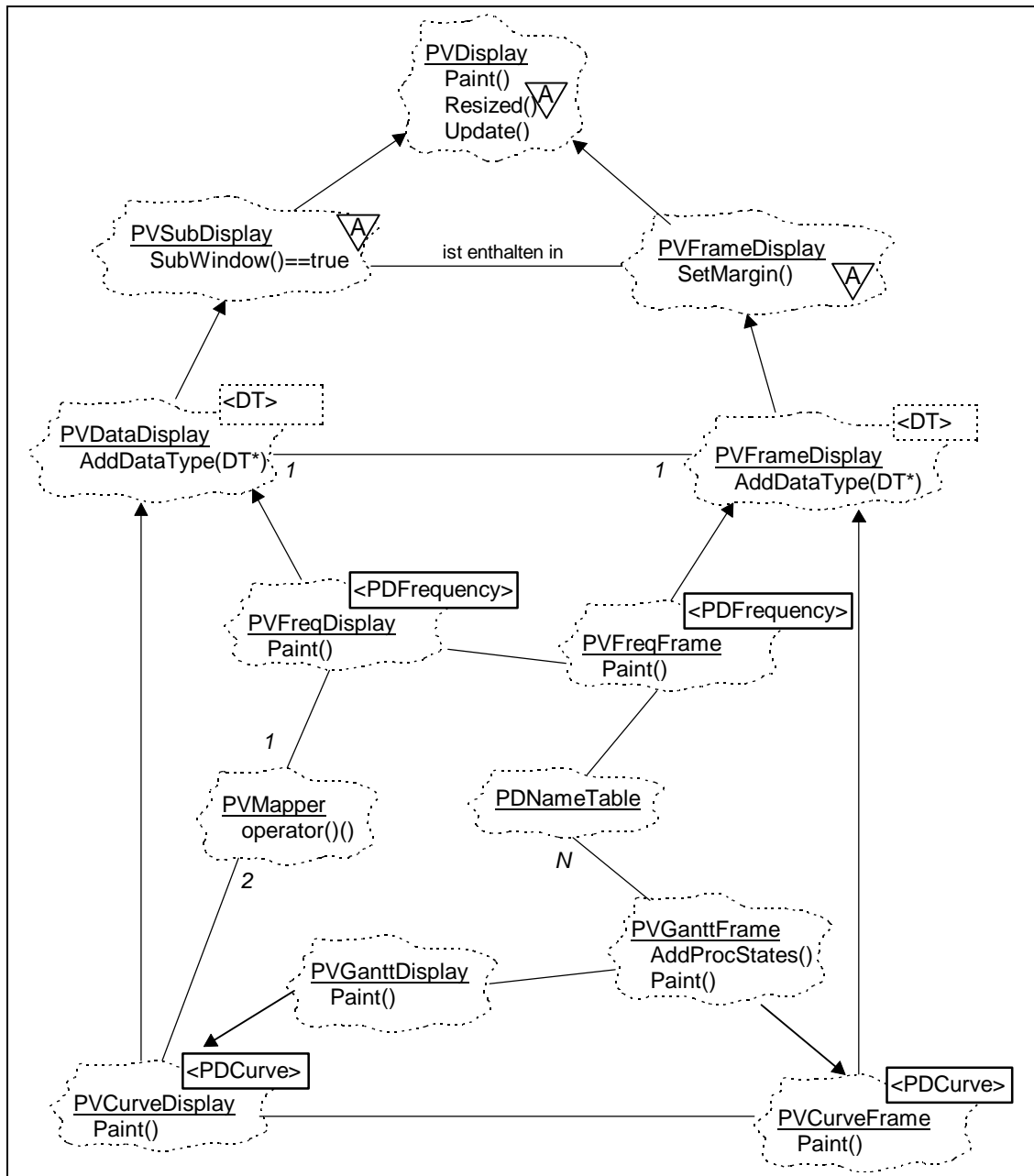


Abbildung 11: Klassendiagramm Visualisierung

**PVCurvesDisplay:** Verbindet alle im zugeordneten Datentyp PDCurve enthaltenen Meßpunkte durch gerade Linien und zeichnet so ein Kurvendiagramm. Der angezeigte Ausschnitt wird so skaliert, daß alle Punkte sichtbar sind. Die Extrempunkte der Kurve

werden in der Nähe des Fensterrandes positioniert, um die zur Verfügung stehende Fläche möglichst gut auszunutzen. Für die Skalierung werden zwei PVMapper-Objekte benötigt, eines für die X- und eines für die Y-Dimension. Durch einen Parameter läßt sich festlegen, ob die Y-Achse immer im Nullpunkt beginnt, oder ob der Anfang ebenfalls skalierbar ist.

**PVCurvesFrame:** Beschriftet die Achsen eines Kurvendiagramms unter Berücksichtigung der von der assoziierten PVCurvesDisplay-Instanz vorgenommenen Skalierung. Der Abstand der Haupt- und Teilstrichmarkierungen wird automatisch ermittelt, die Beschriftung erfolgt im Exponentialformat.

**PVGanttDisplay:** Gantt Diagramme sind programmtechnisch gesehen nur spezialisierte Kurvendiagramme. Die Y-Koordinate eines Punktes wird allerdings als Prozeßzustand interpretiert und nicht als numerischer Wert. Wenn das Display mit mehr als einer Kurve verbunden ist, wird die Anzeige in horizontal voneinander getrennte Bereiche geteilt, so daß sich die Kurven nicht überlappen. Es können nur PDDiscreteCurve-Objekte visualisiert werden, die ihre Daten von PESStateFrequency Sensoren beziehen.

**PVGanttFrame:** Spezialisierung der PVCurvesFrame-Klasse. Die Beschriftung der Zeitachse bleibt unverändert, während die Y-Achse die Namen der Prozeßzustände anzeigt. Da jeder Prozeß andere Zustandsnamen haben kann, muß ein PVGanttFrame-Objekt mit so vielen PDNameTable Instanzen verbunden werden, wie Prozesse visualisiert werden. Dazu dient die Methode AddProcStates(). Die PDNameTable-Objekte müssen in der gleichen Reihenfolge angegeben werden, wie die dazugehörigen PDDiscreteCurve Objekte. Ansonsten kommt es zu einer falschen Beschriftung.

**PVMapper:** Diese Klasse hat die Fähigkeit, Werte eines logischen Koordinatensystems auf das Pixel-Koordinatensystems eines X-Fensters abzubilden. Dazu muß der logische Ursprung, der Pixelursprung im Fenster, die logische Ausdehnung sowie die Ausdehnung in Pixeln bekannt sein. Jede Instanz kann zur Transformation einer Dimension verwendet werden. Für zweidimensionale Koordinatensystems (wie sie z.B. in Kurvendiagrammen vorkommen) müssen zwei Instanzen benutzt werden.

## 4.5 Auswertungssensoren

Die verschiedenen Auswertungen werden von einer Menge von Sensoren durchgeführt. Für jeden Auswertungstyp gibt es eine spezielle Sensorklasse. Die Basisklasse **PESensor** ist Vorfahre aller Sensoren. Sie definiert für jedes aus der SCL stammende Ereignis (siehe Abbildung 2) eine virtuelle Funktion. Diese Funktion wird zum Zeitpunkt des Ereignisses vom PEEventDispatcher aufgerufen, wenn der Sensor dort registriert ist. Jede spezielle Sensorklasse implementiert die Funktion `NotifyOnEvent()`, die festlegt, von welchen Ereignissen der Sensor benachrichtigt werden möchte.

Die **PESensor** Spezialisierungen lassen sich in zwei Gruppen aufteilen: die erste Gruppe definiert allgemeine Hilfsfunktionen zur Auswertung, wie z.B. die Mittelwertbildung oder Häufigkeitszähler. Sie können nicht direkt instanziiert werden, weil sie keine Ereignisfunktionen redefinieren. Dies geschieht durch die zweite Gruppe. Ihre Aufgabe ist es, das Eintreten bestimmter Ereignisse zu verarbeiten. Die ermittelten Daten werden durch die Funktionen der Klassen aus Gruppe Eins gespeichert und aggregiert. Voraussetzung ist natürlich, daß die Klassen der Gruppe Zwei über Mehrfachvererbung auch von den Klassen der ersten Gruppe abgeleitet sind. Jeder Sensor führt Auswertungen für genau ein Objekt der Spezifikation aus, z.B. einen Prozeß.

Ein Beispiel soll den Nutzen dieser Vorgehensweise verdeutlichen: in einem QSDL-Modell besitzen sowohl Prozesse als auch Maschinen Warteschlangen. Den Anwender interessieren Maße wie durchschnittliche Warteschlangenlänge, Wartezeit usw. Prinzipiell unterscheidet sich die Auswertung einer Prozeßwarteschlange nicht von der einer Maschinenwarteschlange. Die SCL-Ereignisse, die sich mit Prozessen und Maschinen beschäftigen sind jedoch disjunkt. Die allgemeine Auswertung wird nun in einer von **PESensor** abgeleiteten Klasse zusammengefaßt. Von dieser Klasse kann dann z.B. eine Sensorklasse abgeleitet werden, welche nur auf die Ereignisse reagiert, die eine Veränderung der Prozeßwarteschlange zur Folge haben. Zur Auswertung werden die Funktionen der allgemeinen Hilfsklasse benutzt. Analog läßt sich der Maschinensensor implementieren, der Änderungen der Maschinenwarteschlange registriert und in Aufrufe der allgemeinen Auswerteklasse übersetzt.

### 4.5.1 Basissensoren

Die folgenden abstrakten Basissensoren stellen Funktionen zur Verfügung, die von vielen weiter spezialisierte Sensoren verwendet werden. Sie sind über eine virtuelle Vererbungsbeziehung mit der Basisklasse **PESensor** verbunden, weil dieser Sensor als Empfänger der SCL-Ereignisse in jeder Vererbungshierarchie nur einmal existieren darf.

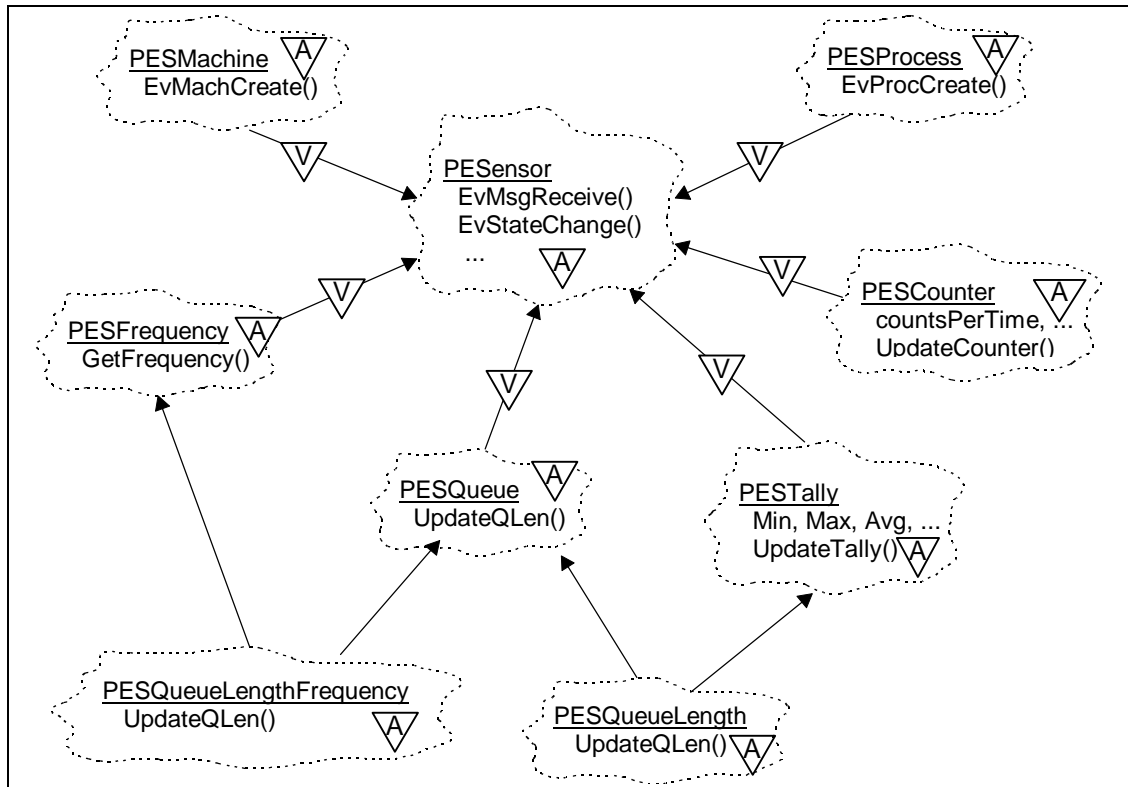


Abbildung 12: Klassendiagramm Basissensoren

**PESTally**: Wertet eine Folge von gewichteten reellen Werten aus. Minimum Maximum, Durchschnitt, Varianz und Standardabweichung werden „on the fly“ ermittelt, d.h. dieser Sensor speichert nur aggregierte Werte und nicht die gesamte Zahlenfolge. Aus den Summen der Gewichte  $s_g = \sum_i g_i$ , der gewichteten Werte

$s_x = \sum_i g_i x_i$  und der quadrierten gewichteten Werte  $s_q = \sum_i (g_i x_i)^2$  lassen sich die

gewünschten Maßzahlen jederzeit ermitteln:  $\bar{x} = \frac{s_x}{s_g}$ ,  $\text{var} = \frac{s_q - \frac{s_x^2}{s_g}}{s_g - 1}$  und  $\text{dev} = \sqrt{\text{var}}$ .

Für die Visualisierungskomponente wird auch der Mittelwert über ein festes Zeitintervall ermittelt. Dieser Wert gibt Auskunft über den aktuellen Zustand. Die Intervallgröße läßt sich als Parameter bei der Konstruktion angeben. Alle berechneten Werte lassen sich über GetValue() auslesen.

**PESCounter:** Diese Klasse dient zum Zählen von Impulsen, die z.B. durch das Eintreten bestimmter Ereignisse ausgelöst werden. Außer der absoluten Anzahl der Impulse kann auch der Durchsatz, d.h. die Anzahl der Impulse geteilt durch die Simulationszeit, sowie der aktuelle Durchsatz, Impulse pro Zeitintervall, ausgegeben werden. Wie bei PESTally kann auch hier die Größe des Zeitintervalls bei der Konstruktion angegeben werden.

**PESFrequency:** Ein PESFrequency Objekt zählt Häufigkeiten unterschiedlicher Dinge, z.B. die Dauer, mit der sich ein Prozeß in einem bestimmten Zustand befindet. Jede Häufigkeit wird durch einen Index identifiziert, im obigen Beispiel wäre es der Zustandsindex. Außerdem wird die Summe aller Häufigkeiten gebildet, so daß relative und absolute Häufigkeiten ermittelt werden können. Zur Durchführung dieser Aufgabe werden die von PDFrequency angebotenen Dienste verwendet.

**PESQueue:** Basisklasse für Warteschlangenauswertungen. Über die SCL-Ereignisschnittstelle ist es nicht möglich, die aktuelle Länge einer Signal- oder Requestwarteschlange zu ermitteln. Daher wird PESQueue zur Verwaltung der aktuellen Warteschlangenlänge benutzt. Die Funktion UpdateQLen() muß von abgeleiteten Klassen aufgerufen werden, wenn sich die Warteschlangenlänge ändert.

**PESQueueLength:** Führt eine Auswertung auf Basis der PESTally-Klasse über die Warteschlangenlänge durch. Bei der Berechnung der durchschnittlichen Länge wird die Dauer berücksichtigt, mit der eine Warteschlange eine bestimmte Länge annimmt. Hat sie z.B. für 6 ZE eine Länge von 0, und für 2 ZE die Länge von 1, so wird die durchschnittliche Länge als  $\frac{6 \cdot 0 + 2 \cdot 1}{6 + 2} = 0.25$  angegeben.

**PESQueueLengthFrequency:** Eine weitere Warteschlangenauswertung, diesmal auf Basis der PESFrequency-Klasse. Dabei wird die Häufigkeit jeder möglichen Länge getrennt erfaßt. Die relativen Häufigkeiten können als Wahrscheinlichkeit der

dazugehörigen Länge interpretiert werden, die Klasse als ganzes ermittelt somit die Verteilung der Warteschlangenlänge.

**PESMachine:** Bindet einen Sensor an eine konkrete, benannte Maschine des simulierten Modells. Der Maschinenname muß bei der Konstruktion als Parameter angegeben werden. Sobald der Sensor die Erzeugung einer Maschine registriert (EvMachCreate Ereignis), legt er die von der SCL benutzte interne Kennung in einem Attribut ab. Dieses Attribut wird von abgeleiteten Klassen zur Entscheidung darüber benutzt, ob sich ein Ereignis auf die assoziierte Maschine bezieht oder nicht.

**PESProcess:** Bindet analog zu PESMachine einen Sensor an einen konkreten, benannten Prozeß des simulierten Modells.

#### 4.5.2 Prozeß- und Signalorientierte Sensoren

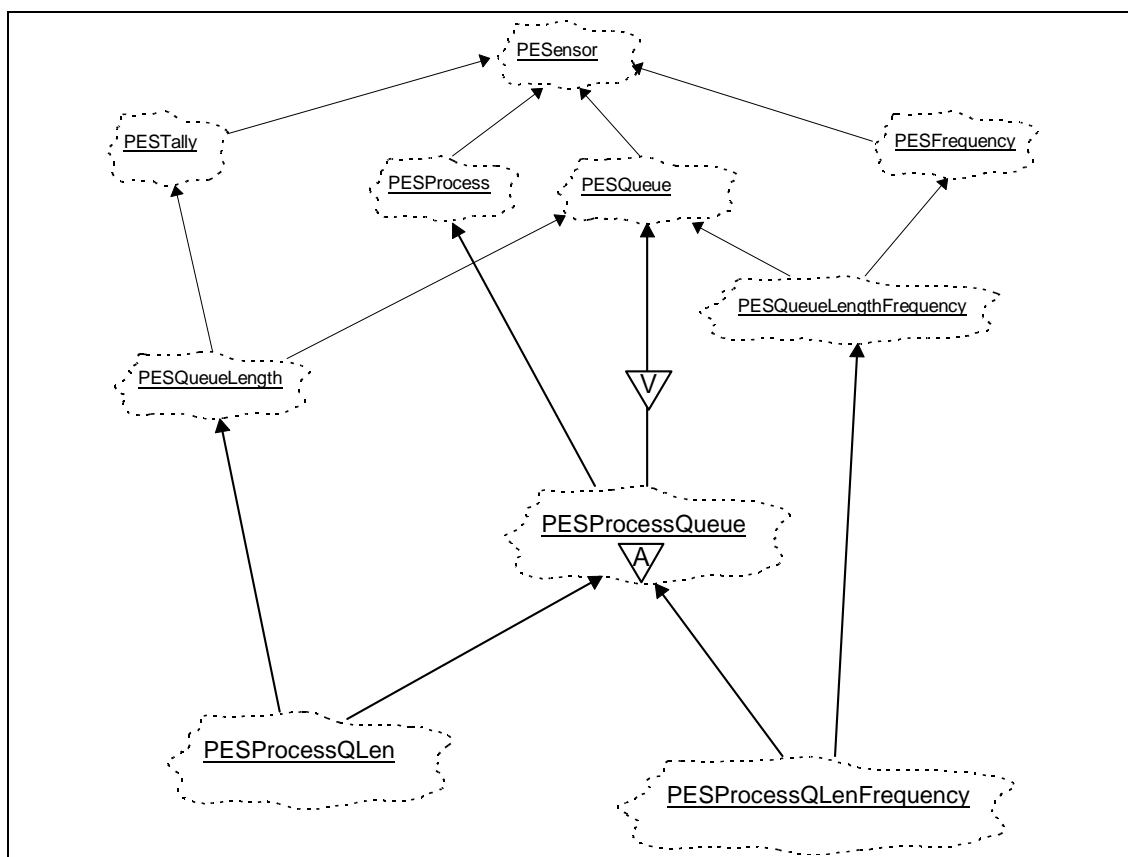


Abbildung 13: Sensoren für Signalwarteschlangen

Die in Abbildung 13 gezeigten Sensoren befassen sich mit der Auswertung von Signalwarteschlangen, wobei die im vorherigen Kapitel vorgestellten abstrakten Klassen neu zusammengefaßt und spezialisiert werden.

**PESProcessQueue:** Faßt PESQueue und PESProcess zusammen und spezialisiert sie so, daß Auswertungen über die zu einem konkreten Prozeß gehörende Signalwarteschlange möglich werden.

**PESProcessQLen:** Auf PESQueueLength basierende Auswertung der zu einem Prozeß gehörenden Signalwarteschlange.

**PESProcessQLenFrequency:** Auf PESQueueLengthFrequency basierende Auswertung der zu einem Prozeß gehörenden Signalwarteschlange.

Abbildung 14 zeigt weitere Sensoren, die Prozeß- und Signalorientierte Auswertungen durchführen:

**PESProcessNumber:** Sensoren dieser Klasse zählen die Instanzen eines dynamischen Prozeßtyps. In QSDL ist es möglich, Prozesse zur Laufzeit zu erzeugen und zu zerstören. Über deren Anzahl kann eine auf PESTally basierende Auswertung durchgeführt werden. Die Anzahl der Instanzen wird mit ihrer Lebensdauer gewichtet, so daß der Mittelwert als die zu einem beliebigen Zeitpunkt erwartbare Anzahl von Prozessen interpretiert werden kann. Der Name des zu beobachtenden Prozeßtyps muß bei der Konstruktion angegeben werden.

**PESStateFrequency:** Diese Klasse ermittelt die Häufigkeit, mit der sich der zugeordnete Prozeß in einem bestimmten Zustand befindet. Für diese Aufgabe ist sie von PESProcess und PESFrequency abgeleitet. Neben ihrer Hauptaufgabe ermittelt sie auch die Namen der Prozeßzustände, welche von der Visualisierungskomponente benötigt werden. Diese Namen werden durch die Basisklasse PDNameTable gespeichert. Die Ermittlung der Namen muß relativ aufwendig über einen Sensor geschehen, weil die SCL Namen nur als Parameter einer Ereignisfunktion übergibt. Solange ein Prozeßzustand nicht vorkommt, solange ist der Name dieses Zustands für die Auswertungskomponente unbekannt. Für Ganttprogramme wird außerdem der aktuelle Zustand über GetValue() bereitgestellt.

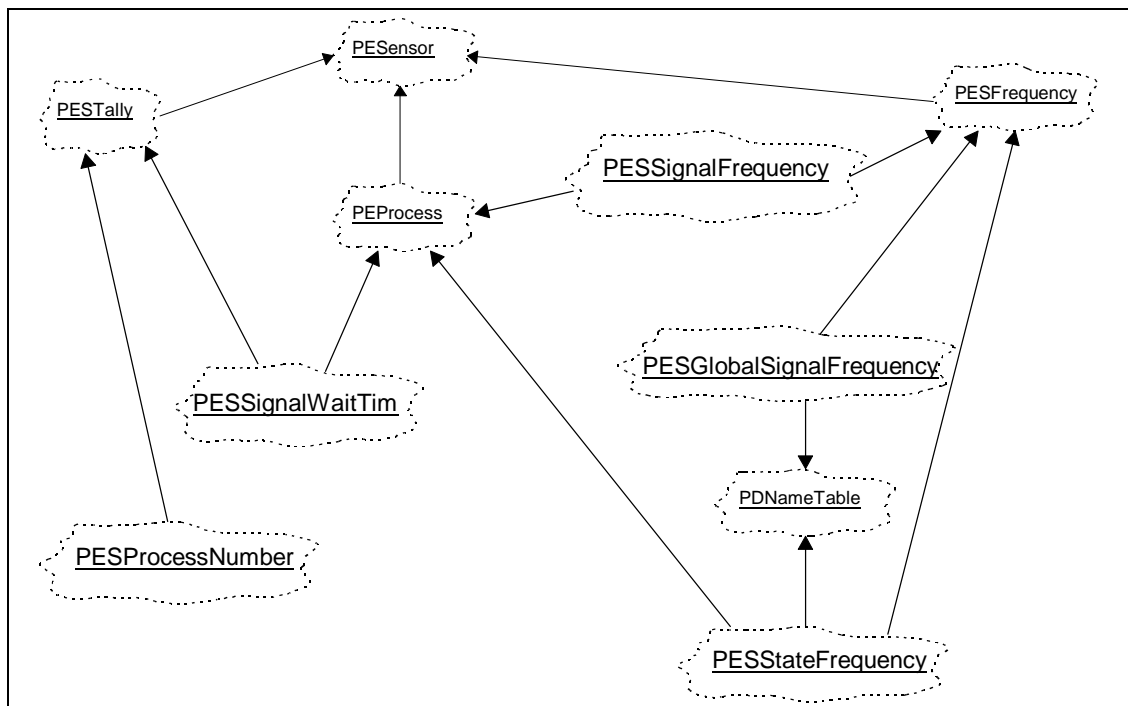


Abbildung 14: Prozeß- und Signalorientierte Sensoren

**PESSignalWaitTime:** Spezialisierung von PESTally und PEProcess, um die Wartezeit von Signalen vor einem Prozeß auszuwerten. Die Wartezeit beginnt zu dem Zeitpunkt, wo ein Signal in die Warteschlange eingefügt wird (MessageReceive), sie endet mit dem Entnahmezeitpunkt (MessageConsume). Die Zeiten können entweder für alle Signaltypen oder nur für einen bestimmten Typ erfaßt werden. Im zweiten Fall muß der Name des Signaltyps bei der Konstruktion als zusätzliche Parameter angegeben werden.

**PESSignalFrequency:** Diese Klasse erfaßt die Häufigkeiten, mit der Signaltypen an einem Prozeß ankommen oder von diesem ausgesandt werden. Die Richtung der Signale (Ankunft oder Abgang) muß als Parameter bei der Konstruktion angegeben werden.

**PESGlobalSignalFrequency:** Die Häufigkeiten aller Signaltypen im gesamten simulierten System werden von dieser Klasse erfaßt. Jede Ankunft eines Signaltyps an einem Prozeß erhöht die Häufigkeit. Eine wichtige Nebenaufgabe dieses Sensors ist es, die Namen der Signaltypen zu ermitteln. Zu diesem Zweck ist PDNameTable eine Basisklasse, analog zur Vorgehensweise bei der PESStateFrequency-Klasse.

### 4.5.3 Maschinen- und Requestorientierte Sensoren

Fast alle Klassen, die Prozeß- und Signalorientiert arbeiten, werden analog für Maschinen und Requests implementiert. In der Implementierung beschränkt sich die Arbeit auf die Definition neuer Ereignisfunktionen zur Datensammlung, während bei der eigentlichen Auswertungen auf gemeinsame Basisklassen zurückgegriffen wird. Dadurch ist eine erneute Implementierung überflüssig.

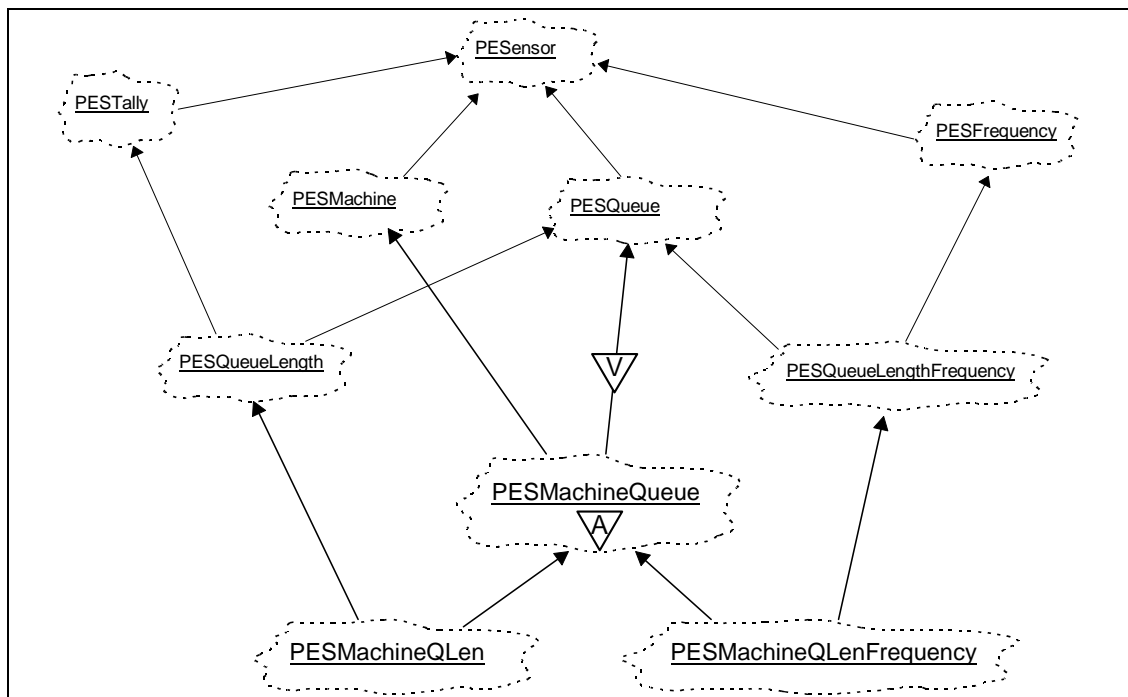


Abbildung 15: Sensoren für Requestwarteschlangen

**PESMachineQueue:** Basisklasse zur Auswertung der Requestwarteschlange einer Maschine. Bei den Maschinen ist eine Besonderheit der SCL zu beachten: ein Request wird erst dann aus der internen Warteschlange entfernt, wenn er vollständig bearbeitet wurde. Eine Warteschlangenlänge von eins bedeutet also nicht, daß ein Request wartet, er befindet sich bereits in Bearbeitung. Das bedeutet, daß die Warteschlangenlänge in diesem Fall korrekterweise als Population der Aufträge an einer Maschine bezeichnet werden müßte. Damit aber die Verwandtschaft zu den Warteschlangensensoren erkennbar bleibt, wurde die Klassen weiterhin mit Queue und QLen benannt.

**PESMachineQLen:** Auf PESQueueLength basierende Auswertung der Population an einer konkreten Maschine.

**PESMachineQLenFrequency:** Auf PESQueueLengthFrequency basierende Auswertung der Populationshäufigkeit an einer konkreten Maschine.

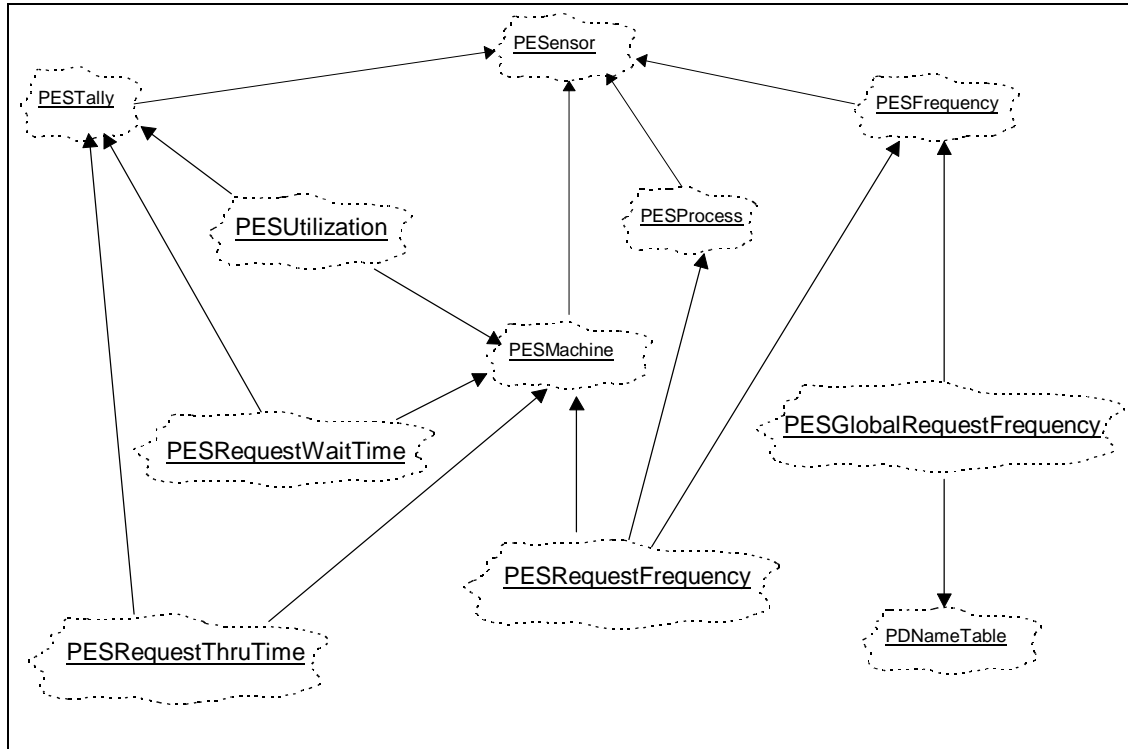


Abbildung 16: Maschinen- und Requestorientierte Sensoren

**PESUtilization:** Berechnet die Auslastung einer Maschine. Der Auslastungsgrad einer Maschine ist definiert als die Anzahl der in Bearbeitung befindlichen Requests dividiert durch die Anzahl der zur Verfügung stehenden Bediener. Der Auslastungsgrad ändert sich, sobald die Bearbeitung eines Auftrags beginnt (RequestStart) oder endet (RequestStop). Der mit der Zeitdauer gewichtete Auslastungsgrad wird an die Basisklasse PESTally zur weiteren Auswertung übergeben.

**PESRequestWaitTime:** Auf PESTally basierende Auswertung der Wartezeit eines Requests. Die Wartezeit ist definiert durch den Eintritt in die Requestwarteschlange (RequestIssue) und dem Beginn der Bearbeitung (RequestStart). In die Auswertung können die Zeiten aller Requesttypen oder nur die eines bestimmten Typs eingehen. Im letzteren Fall muß der Requestname als Parameter bei der Konstruktion angegeben werden. Diese Klasse kann mit der zum Zeitpunkt dieser Arbeit vorliegenden SCL-Version nur solche Maschinen korrekt auswerten, die nach einer nichtunterbrechenden Strategie arbeiten.

**PESRequestThruTime:** Berechnet analog zu PESRequestWaitTime die Durchlaufzeit eines Auftrags durch eine Maschine, d.h. die Zeit, die zwischen der Ankunft an der Maschine (RequestIssue) und dem Ende der Bearbeitung (RequestFinish) vergeht.

**PESRequestFrequency:** Ermittelt die Häufigkeiten mit der Requesttypen von Maschinen empfangen oder von Prozessen ausgesandt werden, wobei auf die Dienste der Basisklasse PESFrequency zurückgegriffen wird. Die zu überwachende Richtung der Requests (Ankunft oder Abgang) ist parametrisiert. Da Requests zwar von Maschinen empfangen, aber nur von Prozessen gesendet werden können, ist diese Klasse sowohl von PESProcess als auch von PESMachine abgeleitet. Je nachdem ob das Senden oder das Empfangen von Requests überwacht werden soll, wird die richtige Basisklasse aktiviert. Der bei der Konstruktion anzugebende Name bezieht sich dementsprechend auf einen Prozeß oder eine Maschine.

**PESGlobalRequestFrequency:** Ermittelt die Häufigkeiten aller im gesamten System auftretenden Requesttypen, indem ihre Ankunft an allen Maschinen überwacht wird. Die eigentliche Auswertung wird von der Basisklasse PESFrequency durchgeführt. Diese Klasse ermittelt außerdem die Namen aller jemals im System aufgetretenen Requests und speichert sie durch die Basisklasse PDNameTable. Die Namen werden von den Visualisierungskomponenten benötigt.

#### 4.5.4 Ereignis- und Aktivitätssensoren

Die Sensoren der Klassen **PESEvent** und **PESActivity** aus Abbildung 17 erlauben es, Auswertungen über benutzerdefinierte Ereignisse und Aktivitäten durchzuführen. Sie sind damit die flexibelsten Sensoren der Auswertungskomponente.

Ein **PESEvent**-Sensor ist von den Auswertungsklassen PESTally und PESCounter abgeleitet. Bei jedem Eintreten eines benutzerdefinierten Ereignisses wird die durch das Counter-Objekt festgehaltene Häufigkeit erhöht. Die Zeit, die zwischen zwei Ereignissen vergeht wird durch das PESTally-Objekt ausgewertet. Das zu beobachtende Ereignis wird durch eine Instanz der Klasse PDEventType definiert. Sie ist bei der Konstruktion des Sensors als Parameter anzugeben.

Ein **PESActivity** beobachtet Aktivitäten, welche durch zwei Ereignisse eingeschlossen sind. Mit dem ersten Ereignis beginnt die Aktivität, das zweite beendet sie. Beide Ereignisse sind Parameter des Sensors. Wie beim PESEvent besitzt auch der Aktivitätssensor die Basisklassen PESTally und PESCounter. Der PESCounter wird zur Auswertung der Häufigkeit der Aktivität eingesetzt, während das PESTally-Objekt die Dauer einer Aktivität auswertet.

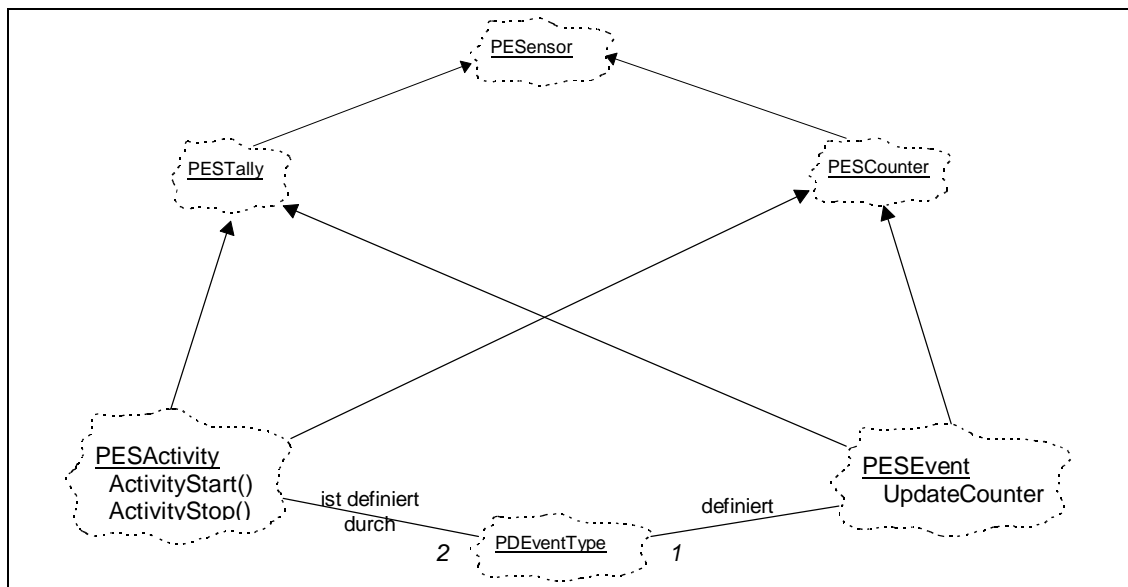


Abbildung 17: Ereignis- und Aktivitätssensoren

## 4.6 Aktualisierung

In periodischen Abständen muß der aktuelle Zustand der Sensoren abgefragt und in die mit den Displays verbundenen Datenspeicher übertragen werden. Diese Aufgabe wird von Objekten der Klasse **PCUpdater** erfüllt. Sie müssen sich beim PEEventDispatcher registrieren lassen und erhalten daraufhin regelmäßig ein Update-Signal. Die abstrakte Basisklasse ist durch zwei spezialisierte Updater implementiert:

**PCFrequencyUpdater:** Überträgt die Häufigkeiten aus einem PESFrequency-Sensor in ein PDFrequency Objekt. Bei der Konstruktion wird der Updater mit beiden Objekten permanent verbunden.

**PCurveUpdater:** Trägt genau einen Meßwert eines beliebigen Sensors als neuen Punkt in ein PDCurve Objekt ein. Da ein Sensor üblicherweise mehrere Auswertungen

gleichzeitig durchführt, muß bei der Konstruktion auch ein Indexwert angegeben werden, der der Sensormethode GetValue() als Parameter übergeben wird, um einen bestimmten Meßwert abzufragen. Gültige Indexwerte sind in den Klassen PESTally, PESCounter und PESQueueLength definiert. Instanzen dieser Klasse werden auch zur Aktualisierung von PDDiscreteCurve Objekten eingesetzt.

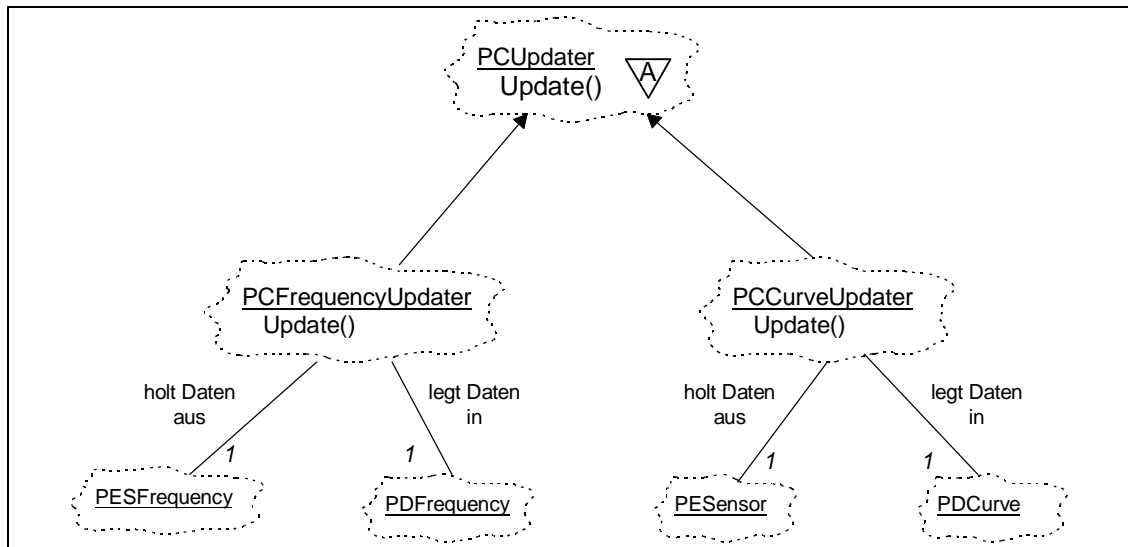


Abbildung 18: Klassendiagramm Aktualisierung

## 4.7 Auswertungsberichte

Die Implementierung des Berichtsgenerators enthält eine lokal auf die Sensorklassen verteilte und eine im PEEventDispatcher zentralisierte Komponente. Jede Sensorklasse redefiniert die virtuelle Methode Report(). Instanzierbare Klassen geben eine kurze Beschreibung aus, und benutzen dann die Report-Methode ihrer Auswertungs-basisklassen auf, also PESTally, PESCounter oder PESFrequency um die gesammelten Daten auszugeben. Der PEEventDispatcher enthält ein Stream-Objekt, welches den Zugriff auf die Berichtsdatei organisiert. Die Methode ReportAllSensors() der PEEventDispatcher Klasse schreibt lediglich einen Berichtskopf in diese Datei und ruft anschließend für jeden registrierten Sensor die Report-Funktion auf, wobei das Stream-Objekt als Parameter übergeben wird.

## 5 Experimentbeschreibung

### 5.1 Notwendigkeit der Experimentbeschreibung

Die Simulation eines Modells hat den Zweck, Erkenntnisse zu gewinnen. Jede Simulation wird dabei als Experiment verstanden. Wie in der Realität der Aufbau des Experiments Auswirkungen darauf hat, welche Daten ermittelt werden können, muß auch das Simulationsexperiment sorgfältig geplant werden. Das PEV-System stellt viele unterschiedliche Sensoren zur Analyse bereit, doch der Anwender muß entscheiden, welche Elemente des Modells untersucht werden, und welche Sensoren dazu eingesetzt werden. Die Visualisierungskomponente benötigt außerdem Hinweise darüber, wie die ermittelten Daten grafisch aufbereitet werden sollen. Diese Informationen gehören in die Beschreibung des Experiments. Es ist sinnvoll, die Experimentbeschreibung von der Modellbeschreibung zu trennen: einerseits wird es dadurch möglich, mehrere unterschiedliche Experimente mit ein und demselben Modell durchzuführen, andererseits wird jeder Einfluß des Experiments auf das Modell ausgeschlossen. Wenn jedes Experiment nur einen Teilaspekt des Modellverhaltens untersucht, kann die Analyse wesentlich strukturierter und einfacher organisiert werden, als eine Totaluntersuchung des gesamten Systemverhaltens. Diese Vorgehensweise unterstützt außerdem die Erfahrung, daß die ersten Analyseergebnisse die Aufmerksamkeit auf andere Aspekte des Systems lenken. Das bedeutet, daß in der Praxis nicht nur ein Experiment durchgeführt wird, sondern mehrere aufeinander aufbauende Experimente. Dabei werden die Untersuchungen schrittweise verfeinert und verzweigt. Die folgenden Kapitel schildern die Möglichkeiten, die das PEV-System zur Experimentbeschreibung im obigen Sinne bietet.

### 5.2 Experimentbeschreibung in C++

Idealerweise sollte der Anwender in der Lage sein, die zu untersuchenden Objekte vor oder während der Simulationszeit interaktiv auszuwählen. Dies gibt ihm die größtmögliche Freiheit bei der Analyse des Systemverhaltens. Es hat sich aber bereits

sehr früh gezeigt, daß eine Benutzeroberfläche mit diesen Leistungsmerkmalen im Rahmen einer Diplomarbeit aus zeitlichen Gründen nicht realisierbar ist. Daher sollte es genügen, die notwendigen Sensoren und Anzeigefenster direkt im C++ Code anzugeben, entweder manuell oder in Zusammenarbeit mit dem Codegenerator. Die Funktionen und Konstruktoren der PEV-Klassen sind für diese Vorgehensweise entworfen und optimiert worden. Für jedes neue Experiment müßte lediglich die Methode Setup() des PEEventDispatchers neu implementiert werden. Die Erzeugung eines Sensors zur Analyse der Warteschlangenlänge hätte folgendes Aussehen:

```
PESProcessQLen* QLA = new PESProcessQLen("instanza");  
RegisterSensor(QLA);
```

Um die Auswertungen als Kurve in einem Fenster anzuzeigen wäre es zunächst erforderlich, eine Instanz der PDCurve-Klasse für die Datenspeicherung anzulegen und sie über ein Updater-Objekt mit dem Sensor zu verknüpfen:

```
PDCurve* AvgA = new PDCurve(32);  
RegisterUpdater(new PCCurveUpdater(QLA, AvgA, PESTally::avg));
```

Anschließend wird ein Kurvendiagramm durch Instanziierung der Klasse PVCurveFrame angelegt:

```
PVCurveFrame* QCurve = new PVCurveFrame("Average Qlen");  
xEventDispatcher.RegisterDisplay(QCurve);
```

AvgA und weitere Datentypobjekte können dann dem Fenster QCurve zur Anzeige übergeben werden, auch die Zuordnung an mehrere Fenster ist möglich:

```
QCurve->AddDataType(AvgA);  
QCurve->AddDataType(AvgB);  
...  
ProcCurve->AddDataType(AvgA);
```

Diese Methode bietet die größtmögliche Flexibilität der Experimentbeschreibung und die prinzipielle Möglichkeit, durch Spezialisierung der existierenden Sensorklassen selbstdefinierte Auswertungen durchzuführen. Leider zeigten sich im praktischen Einsatz einige Nachteile: Die Sensorkonstruktion ist in einigen Situationen komplexer als das obige Beispiel vermuten läßt, insbesondere kann die implizite Konstruktion

weiterer Objekte notwendig werden, einige Auswertemaße können nur in Objekten vom Typ `PDDiscreteCurve` oder `PDFrequency` gespeichert werden, nicht jedes Display kann jeden Datentyp anzeigen, etc. Der Anwender muß also erstens grundlegende C++ Kenntnisse besitzen, und zweitens sämtliche Klassen und ihre Beziehungen überblicken. Es hat sich gezeigt, daß sich diese Kenntnisse nur schwer vermitteln lassen und für Personen, die die PEV-Komponenten einfach nur benutzen wollen, sehr verwirrend sein können. Daher wurde eine Experimentbeschreibungssprache entwickelt, die einen einfacheren Zugang zum PEV-System erlaubt.

### 5.3 Experimentbeschreibungssprache

Die Durchführung eines PEV-Experiments kann in einer Konfigurationsdatei beschrieben werden. Syntax und Semantik der verwendeten Sprache erläutert dieses Kapitel. Die Syntax wird mit Hilfe Backus-Naur-Form (BNF) spezifiziert, wobei folgende Vereinbarungen gelten: Terminale werden fett gedruckt und im Falle von Sonderzeichen mit Anführungszeichen umgeben. Der vertikale Strich '|' bedeutet „oder“. Ausdrücke in geschweiften Klammern ('{ }') treten null oder beliebig oft auf. Optionale Ausdrücke werden in eckigen Klammern ('[ ]') plaziert. Die linke und rechte Seite einer Produktionsregel werden durch ein '::=' getrennt, die Regel selbst wird durch einen Punkt abgeschlossen. Die Syntax der Experimentbeschreibung ist durch folgende Grammatik spezifiziert:

```

ExperimentDescription ::=
  PEV-Experiment ':' ExpName
  Specification ':' SpecName
  Report ':' ReportPara
  CurvePoints ':' PointsPara
  ScaleAdaption ':' AdaptionPara
  DefaultInterval ':' IntervalPara
  SensorCreation ':' '{' {SensorDefinition} '}'
  DisplayCreation ':' '{' {DisplayDefinition} '}'

ExpName ::= String ';'
SpecName ::= String ';'
ReportPara ::= String [' ',' Real] ';'
PointsPara ::= Integer ';'
AdaptionPara ::= Integer ';'
IntervalPara ::= Real ';'

```

Die Nichtterminale `Ident`, `String`, `Integer` und `Real` sind wie in Programmiersprachen üblich definiert (z.B. C++). Außerdem sind Kommentare im C++ Stil möglich, sie beginnen mit einem doppelten Slash „//“ und reichen bis zum Ende einer Zeile. Wie man sieht, ist die Sprache sehr einfach aufgebaut. Mit einer Menge von Schlüsselwörtern und dazugehörenden Parametern werden die Eigenschaften des Experiments beschrieben. Sie haben eine feste Reihenfolge, von der nicht abgewichen werden kann.

Das Startterminal **PEV-Experiment** definiert eine Beschreibung des Experiments, welche in Berichten und im Steuerungsfenster erscheint. Der nach **specification** anzugebende Spezifikationsname muß mit dem Modellnamen identisch sein, damit jedes Experiment eindeutig einem Modell zugeordnet werden kann und nicht versehentlich Experimente mit der falschen Spezifikation durchgeführt werden. Hinter dem Schlüsselwort **Report** muß der Name der Datei angegeben werden, in welche der Auswertungsbericht geschrieben wird. Optional kann ein Intervall definiert werden, an dessen Ende automatisch ein Bericht ausgegeben wird und alle Sensoren ein Reset-Signal erhalten. Die Eigenschaft **CurvePoints** bestimmt die Anzahl der Meßpunkte, aus denen sich die Meßkurven zusammensetzen. Dieser Wert gilt für alle Kurven im Experiment. **scaleAdaption** definiert für Kurvendarstellungen einen Bereich am Fensterand, den die Kurve nicht verlassen darf. Liegt kein Punkt der Kurve mehr in diesem Bereich, wird die Anzeige so skaliert, daß die Kurve wieder im zulässigen Bereich liegt. Der anzugebende ganzzahlige Wert wird als Prozentanteil der Fenstergröße interpretiert, gültige Größen liegen zwischen 5 und 50. Das **DefaultInterval** ist die Zeitdauer, die allen intervallbezogenen Auswertungen zugrunde liegt, deren Dauer in der Sensordefinition nicht explizit angegeben wird.

Das Schlüsselwort **sensorCreation** leitet eine Liste von Sensordefinitionen ein. Die Syntax zur Erzeugung eines Sensors lautet wie folgt:

```
SensorDefinition ::= SensorType Ident `:` SensorParas `;`.
```

Nach dem Sensortyp muß ein Bezeichner (`Ident`) angegeben werden, der den erzeugten Sensor eindeutig definiert. Unter diesem Namen kann der Sensor in der Displaybeschreibung angesprochen werden. Die Sensorparameter beschreiben die Zuordnung

zu Modellelementen und weitere Eigenschaften. Ihre genauer Aufbau wird durch den Sensortyp bestimmt. Tabelle 1 listet alle zulässigen Sensortypen und die dazugehörigen Parameter auf. In der Spalte „Klasse“ sind die Auswertungen markiert, die ein Sensortyp durchführt. T, C und F stehen für Tally, Counter und Frequency. Die Auswertungsklasse bestimmt, welche Daten (siehe Tabelle 3) in Kurvendiagrammen angezeigt werden können.

<i>Sensortyp</i>	<i>Klasse</i>			<i>Parameter</i>
	<b>T</b>	<b>C</b>	<b>F</b>	
<b>ProcessNumber</b>	⊗			ProcessName [,Interval]
<b>ProcessQLen</b>	⊗			ProcessName [,Interval]
<b>ProcessQLenFreq</b>			⊗	ProcessName
<b>ProcessStateFreq</b>			⊗	ProcessName
<b>ProcessSigWaitTime</b>	⊗			SignalName, ProcessName [,Interval]
<b>ProcessInSigFreq</b>			⊗	ProcessName
<b>ProcessOutSigFreq</b>			⊗	ProcessName
<b>ProcessOutReqFreq</b>			⊗	ProcessName
<b>MachineQLen</b>	⊗			MachineName [,Interval]
<b>MachineQLenFreq</b>			⊗	MachineName
<b>MachineUtilization</b>	⊗			MachineName [,Interval]
<b>MachineReqWaitTime</b>	⊗			RequestName, MachineName [,Interval]
<b>MachineInReqFreq</b>			⊗	MachineName
<b>MachineReqThruTime</b>	⊗			RequestName, MachineName [,Interval]
<b>GlobalSigFreq</b>			⊗	-
<b>GlobalReqFreq</b>			⊗	-
<b>SimpleEvent</b>	⊗	⊗		String, In Out, Signal Request, SigReqName, ProcMachName [,Interval]
<b>SimpleActivity</b>	⊗	⊗		String, In Out, Signal Request, SigReqStart, ProcMachStart, In Out, Signal Request, SigReqStop, ProcMachStop [,Interval]

Tabelle 1: Parameter in Abhängigkeit vom Sensortyp

Sämtliche mit Name endende Parameter sind als `String` definiert. Ihre Namen sind selbsterklärend. Der bei Tally und Counter Sensoren zulässige optionale Parameter `Interval` überschreibt den bei `DefaultInterval` angegebenen Wert. Die Definition eines Sensors in der Experimentbeschreibung führt zur Instanziierung der gleichnamigen Sensorklasse aus Kapitel 4. Die Schlüsselwörter `SimpleEvent` und

**simpleActivity** erwarten als ersten Parameter einen Namen, unter dem die ermittelten Daten in der Berichtsdatei ausgegeben werden.

Nach der Beschreibung der im Experiment einzusetzenden Sensoren, wird durch **DisplayCreation** die Definition der auf dem Monitor erscheinenden Fenster und Daten eingeleitet. Dazu wird eine Liste von Displays angegeben, wobei jedes Display einem Fenster entspricht. Die Syntax hat folgende Form:

```

DisplayDefinition ::= DispType DispName ``' DispParas `';'.
DispType ::= Curves | FixedCurves | Gantt | Freqs.
DispName ::= String.
DispParas ::= SensorData {`,` SensorData} `;'.
SensorData ::= CurveData | GanttData | FreqData.
CurveData ::= Ident `(' ValueIndex [`,` ColorName] `)'.
GanttData ::= FreqData.
FreqData ::= Ident [`,` ColorName `)'].
ValueIndex ::= TallyIndex | CounterIndex | FreqIndex | cql.
TallyIndex ::= min | max | avg | avi | var | dev.
CounterIndex ::= cnt | cpt | cpi.
FreqIndex ::= Integer.
ColorName ::= String.

```

Jedes Display muß mit einem Namen versehen werden, der als Fensterüberschrift angezeigt wird. Darauf folgt die Liste der anzuzeigenden Sensoren. In runden Klammern kann optional der Name einer Farbe angegeben werden, mit der die Darstellung erfolgen soll. Wenn der Farbname nicht in der X11-Farbdatenbank (vgl. [Nye2]) definiert ist, wird eine (unvorhersehbare) Ersatzfarbe benutzt.

<i>Displaytyp</i>	<i>Aufgabe</i>	<i>Darstellbare Sensortypen</i>
<b>Curves</b>	Kurvendiagramm	alle
<b>FixedCurves</b>	Kurvendiagramm mit Ursprung im Nullpunkt	alle
<b>Gantt</b>	Prozeßzustände im Zeitverlauf	nur <b>ProcessStateFreq</b>
<b>Freqs</b>	Relative Häufigkeiten	alle Typen, die mit <b>Freq</b> enden; aber kein Mischen zulässig

Tabelle 2: Zulässige Display-Sensor Kombinationen

Nicht jeder Sensor kann von jedem Display dargestellt werden. Zulässige Kombinationen zeigt Tabelle 2. Sensoren vom Typ Tally und Counter führen mehrere Auswertungen parallel durch. Der Benutzer muß entscheiden, welche Ergebnisse angezeigt werden sollen. Dies geschieht durch die Angabe des `ValueIndex` Parameters. Je nach Auswertungsklasse sind die in Tabelle 3 aufgelisteten

Schlüsselwörter erlaubt. Tabelle 1 läßt sich die Auswertungsklasse eines Sensors entnehmen. Dabei ist anzumerken, daß Sensoren auch zu mehreren Klassen gleichzeitig gehören können. Zur Klasse QLen gehören die Sensoren **ProcessQLen** und **MachineQLen**.

Klasse	ValueIndex	Bedeutung
Tally	<b>min</b>	Minimum aller Werte
	<b>max</b>	Maximum aller Werte
	<b>avg</b>	Durchschnitt aller Werte
	<b>avi</b>	Durchschnitt der Werte im letzten Intervall
	<b>var</b>	Varianz der Wertemenge
	<b>dev</b>	Standardabweichung der Wertemenge
Counter	<b>cnt</b>	Absolute Anzahl Zählimpulse
	<b>cpt</b>	Anzahl Zählimpulse pro Zeit
	<b>cpi</b>	Anzahl Zählimpulse pro Intervall
QLen	<b>cql</b>	Aktuelle Warteschlangenlänge

Tabelle 3: Bedeutung des ValueIndex-Parameter

Als Abschluß zeigt Abbildung 19 eine beispielhafte Experimentbeschreibung. Sie führt zu dem in Abbildung 2 gezeigten Ergebnis.

```
// Experimentbeschreibung für das Merlin-Protokoll
// -----

Experiment      : "Merlin2 - Overview";
Specification   : "merlin2";
Report         : "m2report.txt";
CurvePoints    : 64;
ScaleAdaption  : 10;
DefaultInterval: 8;

SensorCreation:
{
  ProcessStateFreq SF_SU: "sendinguser";
  MachineQLen     MQL1 : "mach1";
  ProcessInSigFreq InSig: "instanza";
  SimpleActivity  DTT  : "DatReq/-Con",
                        In, Signal, "datreq", "instanza",
                        Out, Signal, "datcon", "instanza";
}

DisplayCreation:
{
  Curves "Queue Length of 'mach1'":
    MQL1(avg, "green"), MQL1(avi, "red");

  Curves "Average Data Transport Time [ms]":
    DTT(avg, "black");

  Freqs "States of 'sendinguser'":
    SF_SU;

  Freqs "Signals to 'instanza'":
    InSig("forestgreen");
}
```

Abbildung 19: Beispiel einer Experimentbeschreibung

## 6 Abschluß und Ausblick

Zusammenfassend kann gesagt werden, daß die mit dieser Arbeit verfolgten Ziele erreicht wurden. Das implementierte Auswertungs- und Visualisierungssystem läuft stabil und wird von den bisherigen Anwendern als nützliches Werkzeug empfunden. Weitere Arbeiten, die sich dieses Werkzeugs bedienen, sind in Vorbereitung. Der Implementierungserfolg ist nicht zuletzt der sehr guten SCL-Ereignisschnittstelle zu verdanken, die praktisch alle benötigten Funktionen bereitstellte. Die freie Verfügbarkeit von Linux, einem auf PC-Basis realisierten POSIX-konformen UNIX-Betriebssystem, hat die Implementierung erst ermöglicht. Allen am Linux-Projekt beteiligten Personen sei hiermit gedankt.

Einige Erweiterungen sind noch wünschenswert: Die Ereignis- und Aktivitätssensoren könnten durch komplexe Ereignisse weiter verbessert werden. Ein komplexes Ereignis besteht aus der Verknüpfung einfacher Ereignisse mit logischen Operatoren und Prädikaten. Auch die Einbeziehung von Signalparametern und lokalen Prozeßvariablen in die Auswertung ist anzustreben, doch wird dies auch Auswirkungen auf die SCL und den Simulatorgenerator haben.

Die im Konzept vorgestellte Oberfläche zur interaktiven Auswertung würde den Nutzen der PEV-Komponenten weiter steigern. Prinzipiell sind die implementierten Sensoren und Displays darauf vorbereitet, auch während der Simulation dynamisch erzeugt zu werden, wenn die auszuwertenden SCL-Objekte bekannt sind. Mit einer solchen, an Debugging-Werkzeugen angelehnten Oberfläche wäre eine vollständigen interaktive Analysemöglichkeit gegeben.



## Abbildungsverzeichnis

Abbildung 1: Konzept der Simulationsumgebung .....	12
Abbildung 2: Ereignisse und Klassen der SCL-Schnittstelle .....	18
Abbildung 3: Typische Visualisierung eines Simulationslaufes .....	22
Abbildung 4: Unterschied zwischen synchroner und asynchroner Visualisierung .....	24
Abbildung 5: Gantt Diagramm.....	25
Abbildung 6: Textueller Analysebericht .....	26
Abbildung 7: Interaktive Simulatorsteuerung .....	27
Abbildung 8: Elemente der Booch-Notation für Klassendiagramme.....	29
Abbildung 9: Übersicht der PEV-Architektur.....	31
Abbildung 10: Klassendiagramm Datentypen.....	34
Abbildung 11: Klassendiagramm Visualisierung.....	37
Abbildung 12: Klassendiagramm Basissensoren .....	40
Abbildung 13: Sensoren für Signalwarteschlangen.....	42
Abbildung 14: Prozeß- und Signalorientierte Sensoren .....	44
Abbildung 15: Sensoren für Requestwarteschlangen.....	45
Abbildung 16: Maschinen- und Requestorientierte Sensoren .....	46
Abbildung 17: Ereignis- und Aktivitätssensoren .....	48
Abbildung 18: Klassendiagramm Aktualisierung .....	49
Abbildung 19: Beispiel einer Experimentbeschreibung.....	57

## Literaturverzeichnis

- [Booch] *Booch, Grady*: Objektorientierte Analyse und Design. Bonn: Addison-Wesley Publishing Company, 1994.
- [CCITT] Recommendation Z.100: CCITT Specification and Description Language (SDL). ITU-T, 1994.
- [Gh/Ja/Ma] *Ghezzi, Carlo; Jazayeri, Mehdi; Mandrioli, Dino*: Fundamentals of Software Engineering. London, Prentice-Hall International, 1991.
- [Haviland] *Haviland, Keith; Salama, Ben*: UNIX System Programming. Wokingham: Addison-Wesley Publishing Company, 1987.
- [Hogrefe] *Hogrefe, Dieter*: Estelle, LOTOS und SDL: Standard-Spezifikationsprachen für verteilte Systeme. Berlin: Springer-Verlag 1989.
- [Husain] *Husain, Kamran; Parker, Tim; et al.*: Linux Unleashed. Indianapolis: Sams Publishing 1995.
- [Jones] *Jones, Oliver*: Einführung in das X-Window-System. München, Wien, London: Carl Hanser Verlag & Prentice Hall International, 1991.
- [Klar] *Klar, Rainer et al*: Messung und Modellierung paralleler und verteilter Rechensysteme. Stuttgart: B.G. Teubner, 1995.
- [Kobara] *Kobara, Shiz*: Visual Design with OSF/Motif. Reading, Massachusetts: Addison-Wesley Publishing Company, 1991.
- [McMinds] *McMinds, Donald L.*: Mastering OSF/Motif Widgets. Second Edition, Reading Massachusetts: Addison-Wesley Publishing Company, 1993.
- [Nye 1] *Nye, Adrian*: The Definite Guides to the X Window System Volume One: Xlib Programming Manual. Third Edition, Sebastopol CA: O'Reilly & Associates, 1992.

- [Nye 2] *Nye, Adrian: The Definite Guides to the X Window System Volume Two: Xlib Reference Manual. Third Edition, Sebastopol CA: O'Reilly & Associates, 1992.*
- [Nye 3] *Nye, Adrian: The Definite Guides to the X Window System Volume Four: X Toolkit Intrinsic Programming Manual. Second Edition, Sebastopol CA: O'Reilly & Associates, 1990.*
- [OSF] *Open Software Foundation: OSF/Motif Programmer's Guide. Englewood Cliffs, New Jersey: Prentice Hall, 1990.*
- [Rühl] *Ruehl, Jörg: SDL-basierte Modelle zur Leistungsbewertung von TCP. Essen: Diplomarbeit Wirtschaftsinformatik, Universität-GHS-Essen, 1995.*
- [Stroustrup] *Stroustrup, Bjarne: Die C++ Programmiersprache. Zweite Auflage, Bonn: Addison-Wesley Publishing Company, 1992.*
- [Textor] *Textor, Wolfgang: Ein Macintosh-Simulationswerkzeug für Kommunikationsprotokolle. Dortmund: Diplomarbeit Informatik IV, Universität Dortmund, 1994.*

## **Eidesstattliche Versicherung**

„Ich versichere an Eides Statt durch meine Unterschrift, daß ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe und mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.“

Marl, den 03.11.1995

Christian Rodemeyer