

**HUMBOLDT-UNIVERSITÄT ZU BERLIN**

**INSTITUT FÜR INFORMATIK**

**LEHRSTUHL FÜR SYSTEMANALYSE**



**Diplomarbeit**

**ODEM<sub>x</sub>**

**Neue Lösungen für die Realisierung von C++-Bibliotheken  
zur Prozesssimulation**

**Ralf Gerstenberger**

**Betreuer:**

**Prof. Dr. Joachim Fischer**

**Dr. Klaus Ahrens**

**Berlin, 13. Februar 2003**



## **Dank**

Für meine Betreuung im Studium danke ich Herrn *Prof. Dr. Joachim Fischer*, der auch den Anstoß für die Diplomarbeit gegeben hat.

Herrn *Dr. Klaus Ahrens* gilt mein besonderer Dank für seine Unterstützung bei der Erstellung der Arbeit und für seine Anregungen.

Für die Erläuterung seiner Sparc-Lösung danke ich Herrn Martin von Löwis.

Ebenso möchte ich mich bei Herrn Michael Piefel bedanken, der mich bei der Anfertigung der Studienarbeit unterstützte.

Weiterhin gilt mein Dank allen Beteiligten an den Projekten **ODEM** und **Simring**, die diese Arbeit ermöglicht haben, sowie allen Mitarbeitern des Lehrstuhls für Systemanalyse für ihre Anregungen und Kritiken.

Abschließend danke ich meiner Familie für ihre Unterstützung.



---

## Inhaltsverzeichnis

Zusammenfassung.....	1
1 Vorwort .....	3
2 Vorüberlegungen.....	5
2.1 Grundlagen der Prozesssimulation.....	5
2.2 SimBeans .....	10
2.2.1 JavaBeans .....	11
2.2.2 DEVS-Formalismus .....	11
2.2.3 Architektur von <i>SimBeans</i> .....	13
2.2.4 Datenerfassung und Auswertung mit <i>SimBeans</i> .....	13
2.3 ODEM.....	15
3 ODEMx .....	17
3.1 Design .....	17
3.1.1 Coroutine / CoroutineContext .....	18
3.1.1.1 Anwendung.....	18
3.1.1.2 Stack-basierte Implementation.....	22
3.1.1.3 Anpassung der Stack-Implementation an Sparc-Rechner.....	23
3.1.1.4 Win32-Implementation .....	23
3.1.2 Process / Simulation .....	24
3.1.2.1 Anwendung.....	25
3.1.2.2 Anwendungsbeispiele .....	27
3.1.2.3 Scheduling und Berechnung einer Simulation.....	33
3.1.3 Bewertung der Designentscheidungen .....	34
3.2 Objekt-Objekt-Kopplung .....	35
3.2.1 Struktur von Observation .....	35
3.2.2 Realisierung.....	38
3.2.3 Bewertung der Objekt-Objekt-Kopplung.....	39
3.3 Erfassung und Auswertung der Daten .....	42
3.3.1 Zielsetzungen einer Simulation .....	42
3.3.2 Anforderungen .....	43
3.3.3 Lösungen .....	44
3.3.3.1 Trace .....	44
3.3.3.2 Report.....	57
3.3.3.3 Anwendung von <i>Observation</i> .....	63
3.4 Verwendung von Templates am Beispiel Ressourcen .....	65
3.4.1 Bin und Res .....	65
3.4.2 BinT.....	67
3.4.3 ResT .....	67
4 Ausblicke.....	67
4.1 Introspection/ Inspection.....	67
4.2 Ein generischer Ressourcenbaustein .....	67
Anhang A: Dokumentation von ODEMx .....	67
Literaturverzeichnis .....	67

---



---

## Zusammenfassung

Im Rahmen dieser Diplomarbeit werden, anhand einer neuen Bibliothek für Prozesssimulationen in C++ (**ODEMx**), vier Lösungen für die Realisierung von Simulationsbibliotheken entwickelt und untersucht. Diese Lösungen sind:

- ein neues Design der Umsetzung von Prozessen der Prozesssimulation in C++,
- ein an JavaBeans angelehntes Kopplungsverfahren für Objekte,
- ein neues und zwei weiterentwickelte Konzepte für die einheitliche Erfassung und Weiterverarbeitung von Simulationsdaten und
- die Verwendung von C++-Templates (am Beispiel von Modellbausteinen für die Modellierung von Ressourcen).

Die in dieser Arbeit entwickelte Prozess-Realisierung erlaubt die Kapselung von Simulationen. Dadurch werden mehrere parallele Simulationen innerhalb eines Programms sowie die Schachtelung von Simulationen ermöglicht. (3.1 Design)

Das entworfene Kopplungsverfahren bietet eine dynamische Kopplung von Objekten. Durch Konventionen wird dabei eine einheitliche Verwendung erreicht. Die Unterstützung des Verfahrens durch den Anwender und den Entwickler der Bibliothek wird, auf der Basis der Konventionen, durch Hilfskonstrukte erleichtert. (3.2 Objekt-Objekt-Kopplung)

Für die Erfassung und Weiterverarbeitung von Simulationsdaten werden drei Konzepte entwickelt. Das erste Konzept realisiert die Verarbeitung von Ablaufdaten einer Simulation. Dabei werden Modell und Verarbeitung durch eine verallgemeinerte Ereigniskommunikation gekoppelt, welche die Austauschbarkeit von Auswertungskomponenten ebenso wie die Erweiterung um neue Ereignisse ermöglicht. Das zweite Konzept unterstützt die Erzeugung zusammenfassender Berichte über Simulationen. Durch das Zusammenspiel einzelner Modellkomponenten lassen sich mehrere, voneinander unabhängige, Berichte erstellen. Das Konzept erlaubt, aufgrund einer vereinheitlichten Datenübertragung mit Hilfe von Tabellen, die Unterstützung unterschiedlicher Ausgabeformate. Das dritte Konzept nutzt das Kopplungsverfahren für die dynamische und selektive Erfassung und Weiterverarbeitung von Simulationsereignissen. (3.3 Erfassung und Auswertung der Daten)

Am Beispiel von Modellbausteinen für die Modellierung von Ressourcen wird die Verwendung von C++-Templates untersucht. Es werden zwei Bausteine (BinT, ResT) bereitgestellt, welche die Verwaltung konkreter Elemente unterstützen. (3.4 Verwendung von Templates am Beispiel Ressourcen)<sup>1</sup>

Die durch den Autor entwickelte Bibliothek **ODEMx** ist zusammen mit ihrer Dokumentation unter <http://www.odemx.de> zu finden.

---

<sup>1</sup> Die Design-Änderungen sind im Vergleich zu ODEM von Bedeutung. Das entwickelte Kopplungsverfahren kann nutzbringend in allen Simulationsbibliotheken verwendet werden, die nicht auf Java oder Java-ähnlichen Sprachen basieren. Die drei Konzepte zur Erfassung und Weiterverarbeitung von Simulationsdaten sind allgemein anwendbar.





---

## 1 Vorwort

Mein erster Kontakt zum wissenschaftlichen Bereich der Computersimulation geht auf eine Veranstaltung zum Thema Prozesssimulation von Prof. Fischer zurück. Damals wurden verschiedene Aspekte der prozessorientierten Computersimulation behandelt. Für praktische Beispiele kam die C++-Bibliothek **ODEM** zum Einsatz. Diese Bibliothek bietet Bausteine für die Modellierung und stellt Rahmenbedingungen für Simulationen bereit.

Im Zusammenhang mit meiner Studienarbeit [Ger01] setzte ich mich erneut mit der **ODEM**-Bibliothek auseinander. Das Thema der Studienarbeit war die Erweiterung der Bibliothek um eine grafische Ausgabe der Simulationsabläufe. In der Folge der Studienarbeit wurde **ODEM** in ein gleichnamiges *Open-Source*-Projekt übertragen [<http://sourceforge.net/projects/odem>].

**ODEM** hat neben der Lehre auch in Drittmittel- und Forschungsprojekten Anwendung gefunden. Beispielsweise wird es in angepasster Form als Simulator für SDL-Modelle verwendet (SITE). Eine zweite Anwendung erfolgt im Projekt **Simring**. Dabei wird **ODEM** für die Simulation von Abläufen innerhalb eines Ringwalzwerkes benutzt. Gerade die Erfahrungen des Projekts **Simring** zeigten jedoch, dass **ODEM** in einigen Aspekten verbessert werden sollte. Insbesondere die Techniken für Ausgabe und Verarbeitung der Ergebnisdaten einer Simulation sind noch unzureichend.

Demgegenüber gibt es neue Trends und Entwicklungen sowohl im Bereich der Computersimulation als auch in der Programmiersprache C++. Simulationssysteme setzen inzwischen verstärkt auf Komponenten, während sich in C++ die *Standard Template Library* (STL) durchsetzt. Darüber hinaus werden neue Ideen und Konzepte für die Verwendung von C++-*Templates* entwickelt (siehe [Ale01]).

Aus den Anwendungen der **ODEM**-Bibliothek und dem Interesse für jene Entwicklungen kommt die Motivation für diese Diplomarbeit. Ihr Ziel ist es, neue Techniken und Wege zu untersuchen. Dafür wurde eine Bibliothek entworfen, die einerseits auf Technologien aus **ODEM** aufbaut, aber andererseits auch neue Lösungen anstrebt. Der Name dieser Bibliothek ist **ODEMx**.



## 2 Vorüberlegungen

### 2.1 Grundlagen der Prozesssimulation

In diesem Abschnitt sollen Grundlagen der Prozesssimulation anhand einiger Begriffe erläutert werden.

#### *System*

Ein System ist ein aus mehreren Teilen bestehendes Gebilde, welches eine bestimmte Bedeutung hat und das den folgenden Bedingungen genügt:

- Ein System hat immer eine Systemidentität die gewährleistet, dass stets erkennbar ist, was zum System gehört.
- Die Teile eines Systems stehen zueinander und möglicherweise zu Objekten außerhalb des Systems oder zu anderen Systemen in Beziehungen und bilden dadurch seine Bedeutung.
- Die Teile eines Systems können selbst wiederum Systeme oder atomare Objekte sein.

Ein System ist immer ein geistiges Konstrukt, das den Umgang mit bestimmten Phänomenen ermöglichen soll. Diese Phänomene können der materiellen Umwelt entspringen oder fiktiver Natur sein.

In den verschiedenen Bereichen der Wissenschaften gibt es Systeme unterschiedlicher Ausprägung. Die Sozialwissenschaften beschäftigen sich beispielsweise mit Systemen, die das Zusammenleben zwischen Menschen beschreiben. In der Physik werden Systeme untersucht, deren Teile physikalische Eigenschaften besitzen. Weitere Beispiele für verschiedene Systemarten sind Rechtssysteme, philosophische Systeme und Wirtschaftssysteme. Das wesentliche Kriterium der Systeme, mit denen sich die Computersimulation beschäftigt, ist ihre formale Beschreibbarkeit. Für die Computersimulation sind dabei dynamische Systeme, deren Struktur sich ebenso wie die Eigenschaften ihrer Teile ändern können, typisch.

Systeme haben zu jedem Zeitpunkt einen Zustand, der sich aus ihrer Struktur und den Zuständen aller Teile ergibt. Dieser Zustand wird mathematisch mit Hilfe von Zustandsgrößen definiert. Die Zustandsgrößen eines Systems sind eine minimal große Menge, voneinander unabhängiger Variablen. Jeder einzelne Zustand des Systems ergibt sich aus den konkreten Werten aller Zustandsgrößen. Die Menge aller möglichen Zustände eines Systems bildet den Zustandsraum des Systems.

Bei Systemen, in denen Zeitabläufe eine Rolle spielen, unterscheidet man zwischen Systemen, deren Zustand sich kontinuierlich ändert und solchen, in denen Zustandsänderungen nur zu diskreten Zeitpunkten auftreten. Weiterhin wird allgemein in offene Systeme und geschlossene Systeme unterteilt. Ein System wird offen genannt, wenn Wirkungsbeziehungen zwischen seinen Teilen und seiner Umgebung bestehen. Andernfalls ist ein System geschlossen.

#### *Simulation und Modell*

Simulation ist die Nachbildung des Verhaltens realer oder fiktiver Systeme mit Hilfe von Modellen. Ein Modell stellt dabei ein vereinfachtes System dar, welches in bestimmten Aspekten das gleiche Verhalten wie sein Vorbild zeigt. Abhängig

## 2.1 Grundlagen der Prozesssimulation

---

von der Blickrichtung auf ein System und dem Untersuchungsziel können verschiedene Modelle eines Systems gebildet werden.

Der Zweck einer Simulation ist die Untersuchung des Systemverhaltens, ohne Einfluss auf das ursprüngliche System zu nehmen. Das Verhalten eines Systems umfasst dabei seine Zustandsänderungen und, im Falle eines offenen Systems, seine Aktionen an der Schnittstelle zur Systemumgebung.

### *Computersimulation*

Die Computersimulation ist die Simulation mit Hilfe von (digitalen) Computern. Die verwendeten Modelle lassen sich in Computersprachen formulieren und durch Computer berechnen. Dabei kommen nur rein deterministische Modelle in Frage. Stochastisches Verhalten von Systemen lässt sich jedoch mit Pseudozufallszahlen imitieren. Pseudozufallszahlen sind berechnete Folgen von Zahlen deren statistische und stochastische Eigenschaften echten Zufallszahlen gleichen. Bei der Umsetzung von Computersimulationen verfolgt man vier verschiedene Ansätze:

Der erste Ansatz für die Realisierung einer Simulation ist die vollständige Programmierung eines Modells in einer allgemeinen Computersprache (z.B. C++). Dafür sind alle Aspekte des Modells, zusammen mit den für die Simulation notwendigen Rahmenbedingungen, zu implementieren. Der Vorteil dieses Ansatzes ist seine große Flexibilität. So lassen sich mit ihm z.B. an bestimmte Systeme angepasste Modellbildungsmethoden realisieren. Weiterhin sind Optimierungen in allen Bereichen der Simulation möglich, weshalb dieser Ansatz unter anderem in Computerspielen und anderen zeitkritischen Anwendung eingesetzt wird. Die Nachteile dieser Vorgehensweise sind der hohe Aufwand, der für jede Simulation betrieben werden muss und die minimale Wiederverwendbarkeit seiner Modelle.

Der zweite Ansatz für die Umsetzung von Computersimulationen ist die Programmierung in einer speziellen Computersprache, die allgemeine Techniken der Computersimulation bereits im Sprachumfang enthält. Dadurch lassen sich viele Simulationaufgaben mit vergleichsweise geringem Aufwand realisieren. Die Spezialisierung solcher Sprachen bringt jedoch gleichzeitig Nachteile. Simulationssprachen sind meist für andere Anwendungsbereiche ungeeignet oder leiden unter geringer Effizienz.

Eine Vermittlung zwischen den ersten beiden Ansätzen stellt die Erweiterung von allgemeinen Computersprachen durch Bibliotheken oder Frameworks für die Computersimulation dar.<sup>1</sup> In diesem Fall werden die Rahmenbedingungen für die Realisierung einer Simulation sowie häufig genutzte Bausteine durch die Bibliothek oder das Framework vorgegeben. Dabei sollen die Vorteile spezieller Simulationssprachen in die allgemeinen Computersprachen übertragen werden, ohne deren positive Eigenschaften zu verlieren. Leider lassen sich einige Konzepte der Simulationssprachen nur unzureichend oder ineffizient umsetzen, so dass oftmals Hilfskonstrukte notwendig sind. Dieser Ansatz erreicht deshalb nicht die Klarheit der Simulationssprachen.

Für bestimmte Anwendungsbereiche kommt auch die Modellierung mit Hilfe von Bausteinen in Frage. Dafür stehen oftmals visuelle Werkzeuge zur Verfügung, mit denen sich schnell vorzeigbare Ergebnisse produzieren lassen. Der Nachteil dieser Lösungen ist die Einschränkung auf bestimmte Anwendungsbereiche.

---

<sup>1</sup> Ein Vertreter dieser Kategorie ist ODEM (siehe 2.3 ODEM).

### *Ereignissimulation*

Neben den verschiedenen Ansätzen für die Umsetzung von Computersimulationen existieren unterschiedliche Konzepte für die Modellierung und die Berechnung von Systemen. Eines dieser Konzepte ist die Ereignissimulation, die auch als *Next-Event-Simulation* bezeichnet wird. Sie stellt ein frühes Verfahren der zeitdiskreten Computersimulationen dar. Das Verhalten eines Systems wird dabei als eine Folge von Ereignissen modelliert. Jedes Ereignis verändert den Zustand des Systems und bestimmt die nachfolgenden Ereignisse.

Im Allgemeinen betrachtet die Ereignissimulation Zeitabläufe. Ereignisse werden dabei mit Zeitpunkten in der Modellzeit verknüpft. Die Modellzeit bildet die Zeit, in der sich das Vorbildsystem befindet (Systemzeit), isomorph im Modell ab.

### *Prozesssimulation*

Ein zweites Konzept für die Modellierung und Berechnung von Systemen ist die Prozesssimulation. In der Prozesssimulation werden Systeme mit Hilfe von Prozessen nachgebildet, die wiederum zusammenhängende (zeitliche) Folgen von Aktionen, Änderungen des Systemzustandes, darstellen.

Man unterscheidet zwischen diskreter und kontinuierlicher Prozesssimulation. Die diskrete Prozesssimulation gestattet, im Gegensatz zur kontinuierlichen, lediglich schrittweise Veränderungen des Systemzustandes. Es ist jedoch möglich, kontinuierliche Prozesse in der diskreten Prozesssimulation annäherungsweise zu modellieren.

Die diskrete Prozesssimulation und die Ereignissimulation sind miteinander eng verwandt. Eine diskrete Prozesssimulation lässt sich immer in eine äquivalente Ereignissimulation überführen, indem alle Aktionen der Prozesse in Ereignisse umgewandelt werden. Die Zusammenfassung der Aktionen in Prozesse ist jedoch für viele Anwendungsfälle der Computersimulation der intuitivere und praktischere Ansatz.

Prozesse haben aktive und passive Phasen. In den aktiven Phasen werden Aktionen ausgeführt, in den passiven Phasen wird auf externe Ereignisse gewartet. Externe Ereignisse können z.B. das Erreichen eines bestimmten Zeitpunktes (in der Modellzeit) oder Systemzustandes sein.

### *Ereigniskalender*

Der Ereigniskalender ist eine gedachte Zeitlinie (Modellzeit), auf der stets die nächsten Aktionen der Prozesse eingetragen werden. Anhand dieses Kalenders kann ein Simulator entscheiden, welchen Prozess er als nächstes ausführen muss. Dabei werden die Phasen in denen keinerlei Ereignisse stattfinden bei der Simulation übersprungen.

### *Koroutinen – Threads*

Die Umsetzung diskreter Prozesse in einer Programmiersprache wie C++ läuft darauf hinaus, ihre Aktionen als Befehle der Programmiersprache zu kodieren. Ein Prozess, der die globalen Zustandsgrößen Z1, Z2 und Z3, unterbrochen durch die Wartephasen W1 und W2, verändert, könnte in C++ wie folgt aussehen:

## 2.1 Grundlagen der Prozesssimulation

---

```
void P1() {  
    ...  
    ::Z1=1;  
  
    L1: // Warte W1  
        ::Z2=87;  
  
    L2: // Warte W2  
        ::Z3=0;  
    ...  
}
```

Dabei ist jedoch die Realisierung des Wartens problematisch. Die Funktion P1 müsste an den Punkten L1 und L2 unterbrochen und später, nach Erreichen der entsprechenden Modellzeit, an der gleichen Stelle fortgesetzt werden. Während dieser Wartephasen führen im Allgemeinen andere Prozesse ihre Aktionen aus, weshalb sich folgendes Schema ergeben könnte:

Modellzeit	void P1() {	void P2() {
1	::Z1=1; L1:// Warte W1	
2		::Z1=2; R1:// Warte W3
3	::Z2=87; L2:// Warte W2	
4		::Z1=3; R2:// Warte W4
5	::Z3=0;	
6		::Z1=4;
	}	}

**Tabelle 1**

Mit normalen Funktionsaufrufen lässt sich die Interaktion der beiden Prozesse offensichtlich nicht realisieren. P2 muss sich vor der Rückkehr nach seiner ersten Aktion R1 als Punkt zur Fortsetzung merken. In [Hel99] wird eine Lösung nach dem folgenden Schema für die Programmierung von P2 vorgestellt:

```
void P2() {  
    static unsigned int s=0;  
    switch (s) {  
    case 0:  
        ::Z1=2;  
        ::Scheduler.schedule(P2, W3);  
        ++s;  
        return;  
    case 1:  
        ::Z1=3;  
        ::Scheduler.schedule(P2, W4);  
        ++s;  
        return;  
    case 2:  
        ::Z1=4;  
        ++s;  
        return;  
    default:  
        // Error: Process finished  
    }  
}
```

Der Scheduler realisiert dabei den Ereigniskalender der Simulation und erlaubt mit `schedule` die Funktion `P2` für einen späteren Aufruf einzuplanen.

Die Programmierung von `P2` nach diesem Schema wäre jedoch aufwändig und fehleranfällig. Eine Alternative bestünde in der Unterteilung in verschiedene Funktionen:

```
void P2_2();
void P2_3();

void P2_1(){
  ::Z1=2;
  ::Scheduler.schedule(P2_2, W3);
}

void P2_2(){
  ::Z1=3;
  ::Scheduler.schedule(P2_3, W4);
}

void P2_3(){
  ::Z1=4;
}
```

Die zweite Variante orientiert sich an der Ereignissimulation. Sie ist ebenfalls aufwändig und wenig intuitiv, da sie den Zusammenhang der Aktionen des Prozesses zerstört. Je komplizierter der Prozess wird, desto negativer sind die Folgen dieser Stückelung.

Ein dritter Lösungsansatz besteht in der Verwendung von *Koroutinen* oder *Threads*. Koroutinen, wie auch Threads, verwalten ihren eigenen Ausführungszustand, wodurch sie an beliebigen Punkten in ihrer Ausführung unterbrochen und später fortgesetzt werden können. Beide eignen sich daher sehr gut für die Realisierung der Prozesse. Eine Umsetzung von `P1` und `P2` könnte wie folgt aussehen:

<pre>void P1() {   ...   ::Z1=1;   ::Scheduler.schedule(P1, W1);   ::Scheduler.activateNext();    ::Z2=87;   ::Scheduler.schedule(P1, W2);   ::Scheduler.activateNext();    ::Z3=0;   ... }</pre>	<pre>void P2() {   ...   ::Z1=2;   ::Scheduler.schedule(P2, W3);   ::Scheduler.activateNext();    ::Z1=3;   ::Scheduler.schedule(P2, W4);   ::Scheduler.activateNext();    ::Z1=4;   ... }</pre>
---	--

Da die Einplanung des Prozesses in den Ereigniskalender mit `schedule` und der Wechsel zum nächsten Prozess immer zusammenhängen, könnte man beide auch in einer gemeinsamen Funktion kapseln.

Im Gegensatz zu Threads werden Koroutinen nicht potentiell parallel, sondern immer sequenziell abgearbeitet. Das Scheduling der Koroutinen obliegt daher auch nicht, wie bei Threads, dem Betriebssystem und seinem Prozessormanagement,

sondern erfolgt kooperativ und explizit durch den Programmierer. Entsprechend unterschiedlich sieht die Verwendung der beiden Techniken im Detail aus.

Der Nachteil der dritten Variante ist die unzureichende Unterstützung durch C++ in seiner derzeit standardisierten Form [C++98]. Sowohl Threads als auch Koroutinen gehören nicht zum standardisierten Sprachumfang und müssen rechner- und betriebssystemabhängig implementiert werden.

### *Objektbasierte und objektorientierte Prozesssimulation*

Die objektbasierte Prozesssimulation modelliert Systeme mit Hilfe von Objekten. Ein System wird dabei strukturge treu nachgebildet, indem man enthaltene Teilsysteme durch Objekte repräsentiert und Wirkungsbeziehungen zwischen Teilsystemen in Relationen zwischen Objekten übersetzt. Man unterscheidet zwischen aktiven und passiven Objekten. Aktive Objekte zeigen ein Verhalten, das über die bloße Reaktion auf Eingaben hinausgeht und werden ihrerseits wiederum mit Prozessen modelliert.

Die objektorientierte Prozesssimulation erweitert die objektbasierte Prozesssimulation um weitere Techniken. Durch die Verwendung von

- polymorphen Zeigern und Referenzen,
- Vererbung sowie
- generischer Programmierung,

wird die Wiederverwendbarkeit von Modellen erhöht und die Bildung von Simulationsframeworks unterstützt.

Der Unterschied zwischen reiner Prozesssimulation, objektbasierter und objektorientierter Prozesssimulation lässt sich am Beispiel des Bediensystems Bankschalter verdeutlichen. In diesem System gibt es Kunden, einen Angestellten und den Bankschalter. Die Kunden betreten die Bank, stellen sich am Bankschalter an und werden durch den Angestellten der Reihe nach bedient. Für eine Prozesssimulation dieses Systems könnte man die beiden Prozesse „Kundenbedienung“ und „Ankunft neuer Kunden“ identifizieren und modellieren. Die objektbasierte Prozesssimulation würde statt dessen Objekte für die Teilsysteme Kunde, Angestellter und Bedienschalter verwenden. Ein objektorientierter Ansatz schließlich könnte eine abstrakte Basisklasse für Kunden vorsehen, in deren Ableitungen man besondere Kundenwünsche definiert.

## 2.2 SimBeans

In diesem Abschnitt wird am Beispiel von **SimBeans** eine Anwendung von Komponenten in der Computersimulation dargestellt, die für den Abschnitt 3.2 Objekt-Objekt-Kopplung als Vorlage dient. **SimBeans** ist ein an der Johannes-Kepler-Universität Linz, unter der Leitung von Herbert Prähofer, entstandenes JavaBeans Komponentenframework für Computersimulation ([PSS98] und [PSS99]).

Einführend wird zunächst die grundlegende Technologie JavaBeans erläutert. Danach folgt die Beschreibung der theoretischen Grundlage von **SimBeans** (DEVS-Formalismus) und der **SimBeans** Architektur. Abschließend wird die Datenerfassung und Auswertung als spezielle Anwendung von Komponenten in **SimBeans** behandelt.



### 2.2.1 JavaBeans

JavaBeans ist ein von Sun entwickeltes Komponentenmodell für Java [<http://java.sun.com>]. Eine Softwarekomponente dieses Modells wird JavaBean genannt. JavaBeans sind darauf ausgelegt, mit der Hilfe von Entwicklungswerkzeugen interaktiv angepasst und zu neuen Softwaresystemen zusammengestellt zu werden. Sie unterstützen die Konzepte:

- *Properties*
- *Events*
- *Introspection*
- *Persistence*
- *Customization*

*Properties* sind öffentliche Attribute einer Komponente, auf die man mit nach der Namenskonvention „*getProperty* *setProperty*“ gebildeten Methoden zugreifen kann. *Bound Properties* unterstützen zusätzlich die Überwachung ihrer Werte durch andere Objekte (*PropertyChangeListener*). Deren Anmeldung erfolgt mit der Methode *addPropertyChangeListener*. Veränderungen an den Attributen werden den *PropertyChangeListener* Objekten durch ihre Funktion *propertyChange* mitgeteilt.

Die Kommunikation zwischen JavaBeans läuft über Ereignisse (*Events*). *Events* werden von einer Komponente (*EventSource*) an registrierte Partner (*EventListener*) geschickt. Für die Verwaltung von *EventListener*-Objekten bietet eine *EventSource* *addEventListener* und eine *removeEventListener* Methoden.

*Introspection* bezeichnet die Möglichkeit zur Laufzeit Informationen über die *Properties*, *Events* und Operationen einer Komponente zu erhalten. *Introspection* wird durch die von Java bereitgestellten Mittel (öffentliche Klassenschnittstelle im Java-ByteCode) sowie durch Namenskonventionen unterstützt. Durch eine zusätzliche *BeanInfo*-Klasse können die Namenskonventionen umgangen und stattdessen *Properties* und *Events* explizit angegeben werden.

Das Konzept *Persistence* umfasst die Möglichkeit, den Zustand einer Komponente auf externe Speichermedien zu sichern, und von diesen zu laden. Auch *Persistence* wird durch Java direkt unterstützt.

*Customization* beinhaltet die Möglichkeit der Anpassung von Komponenten an spezielle Anwendungsfälle. Dies soll in *JavaBeans* durch *Properties* umgesetzt werden, die das Verhalten einer Komponente steuern.

Eine *JavaBean* wird durch eine Java-Klasse, die den Namenskonventionen entspricht, oder durch mehrere Klassen zusammen mit einer *BeanInfo* Klasse implementiert.

### 2.2.2 DEVS-Formalismus

DEVS ist ein systemtheoretischer Formalismus für die modulare, hierarchische Modellierung komplexer dynamischer Systeme. Ein System wird nach diesem Formalismus aus Blöcken (Teilsystemen) mit definierten Ein- Ausgabeschnittstellen (*Ports*) zusammengesetzt (*coupled model*) oder durch Zustände und Zustandsübergänge definiert (*atomic model*). Die Struktur eines zusammengesetzten Sys-

tems ergibt sich aus den enthaltenen Teilsystemen und deren Kopplung (*coupling scheme*).

### *Konzepte von SimBeans*

Für die Realisierung des **SimBeans** Framework wurden die folgenden Konzepte als Richtlinien verwendet:

- Es werden elementare Komponenten für Modellierung (Modellkomponenten), Ausgabe, statistische Auswertung und Visualisierung angeboten.
- Eine Bibliothek mit Hilfsobjekten für die Anpassung der Komponenten an bestimmte Anforderungen wird bereitgestellt.
- Es werden Modellschnittstellen definiert, durch welche die Verwendungsmöglichkeiten von Komponenten spezifiziert sind. Die Modellschnittstellen ordnen sich in eine Klassifizierungshierarchie ein, um die Kompatibilität zwischen Modellkomponenten auszudrücken.
- Durch schnittstellenbasierte Modellierung definiert man generische Modellschablonen (*generic templates*). Dafür werden die Schnittstellen der verwendeten Komponenten sowie ihre Kopplungen spezifiziert. Ein konkretes Modell wird durch Einsetzung von entsprechenden Komponenten erzeugt.
- Das Simulationssystem ist *bottom-up*, durch hierarchische Zusammensetzung und Kopplung von Modellkomponenten zu entwickeln.
- Die Komponentenbibliothek soll für spezielle Anwendungsgebiete erweiterbar sein.
- Simulationsumgebungen für spezielle Anwendungsgebiete werden mit Hilfe elementarer Komponenten realisiert.

Ausgehend von diesen Richtlinien werden vier Methoden der Zusammensetzung von Komponenten unterschieden:

*Selection*: Auswahl konkreter Modellkomponenten für generische Modellschablonen (*top-down*-Modellierung).

*Customization*: Anpassung von Modellkomponenten an spezielle Anwendungsfälle, z.B. durch verschiedene Kontrollstrategien oder unterschiedliche Verteilungen von Zufallszahlen.

*Coupling*: Kopplung von Modellkomponenten (*bottom-up*-Modellierung).

*Attachment*: Anbindung von Komponenten für Ausgabe, statistische Auswertung, Visualisierung und Animation an *Properties* von Modellkomponenten.

### 2.2.3 Architektur von *SimBeans*

5. <i>Application-Specific Simulation Systems and Environments</i>
4. <i>Application-Specific simulation Component Sets</i>
3. <i>Elementary Simulation Component Sets</i>
2. <i>Simulation Kernels</i>
1. <i>Java / JavaBeans / JavaBeans Extensions for Computersimulation</i>

**Tabelle 2**

**SimBeans** setzt sich aus mehreren Schichten zusammen. Die unterste Schicht bilden Java und das JavaBeans-Komponentenmodell. SimBeans fügt auf dieser Ebene Erweiterungen für die Realisierung von Simulationen hinzu. Die beiden wesentlichen Erweiterungen der JavaBeans-Infrastruktur sind:

- synchroner und asynchroner Nachrichtentransport sowie
- *Delegate*-Klassen für flexiblere Nachrichtenverteilung.

Darauf aufbauend bilden *Simulation Kernel* die Grundlage für verschiedene Simulationsarten, z.B. zeitdiskrete, zeitkontinuierliche oder kombinierte Simulation.

Über den unterschiedlichen *Simulation Kernel* bieten verschiedene Mengen elementarer Simulationskomponenten (*Elementary Simulation Component Sets*) die Grundlage für Simulationen in bestimmten Anwendungsbereichen, z.B. zeitdiskrete Prozesssimulation. Diese Mengen beinhalten:

- Elementare Modellkomponenten
- Komponenten für typische (klassische) Kontrollschemata
- Komponenten für die Kopplungsverfahren
- Weitere Hilfskomponenten (z.B. Zufallszahlengeneratoren)
- Spezifische elementare Analyse-, Ausgabe- und Visualisierungskomponenten

Auf der vierten Ebene werden anwendungsspezifische Simulationskomponenten eingeführt. Sie modellieren mit Hilfe der elementaren Modellkomponenten typische Teilsysteme des Anwendungsbereichs. Dies können typische Verarbeitungsanlagen, z.B. Öfen in Stahlwerken, sein.

Ausgehend davon werden schließlich die konkreten Systeme modelliert. Sie bieten eine angepasste Benutzerschnittstelle und/oder Schnittstellen zu externen Systemen.

### 2.2.4 Datenerfassung und Auswertung mit *SimBeans*

Die Datenerfassung und Auswertung in **SimBeans** nutzt die Möglichkeiten der JavaBeans Komponentenarchitektur. Für die Auswertung und Erfassung von Daten werden eigene Komponenten entwickelt, die an *Bound Properties* der Modellkomponenten gebunden werden.

## 2.2 SimBeans

---

Die Realisierung von *Properties* und *Bound Properties* erfolgt durch die JavaBeans Komponentenarchitektur auf der Basis von Java. Das folgende Beispiel zeigt eine von Sun's IDE Forte for Java (CE) generierte JavaBean-Klasse.

```
import java.beans.*;

public class Bean extends Object
    implements java.io.Serializable {

    private static final String PROP_SAMPLE_PROPERTY
        = "SampleProperty";

    private String sampleProperty;
    private PropertyChangeSupport propertySupport;

    /** Creates new Bean */
    public Bean () {
        propertySupport = new PropertyChangeSupport ( this );
    }

    public String getSampleProperty () {
        return sampleProperty;
    }

    public void setSampleProperty (String value) {
        String oldValue = sampleProperty;
        sampleProperty = value;
        propertySupport.firePropertyChange (
            PROP_SAMPLE_PROPERTY,
            oldValue,
            sampleProperty
        );
    }

    public void addPropertyChangeListener (
        PropertyChangeListener listener) {
        propertySupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener (
        PropertyChangeListener listener) {
        propertySupport.removePropertyChangeListener(listener);
    }
}
```

Die PropertyChangeSupport-Klasse verwaltet die PropertyChangeListener. Durch die Kapselung der Attribute kann jede Änderung ihres Wertes überwacht und mit der PropertyChangeSupport-Klasse an die angebundenen Komponenten übermittelt werden. Die Möglichkeit, beliebige Komponenten aneinander zu binden, wird durch Konventionen gewährleistet:

- String *PROP\_Property-Name* (z.B.: PROP\_SAMPLE\_PROPERTY)
- Funktion *GetProperty-Name* (z.B.: GetSampleProperty)
- Funktion *SetProperty-Name* (z.B.: SetSampleProperty)
- Funktion *AddPropertyChangeListener* vorhanden
- Funktion *RemovePropertyChangeListener* vorhanden

Die für die Auswertung und Datenerfassung in SimBeans notwendigen Konzepte sind daher:

- Konsequente Kapselung der Attribute einer Klasse,
- Verwaltung abhängiger Objekte und
- Konventionen.

Weitere für die Realisierung vorteilhafte Konzepte sind:

- *Overloading* von Methoden und
- Objektorientierte Programmierung.

### 2.3 ODEM

In diesem Abschnitt soll eine Übersicht über **ODEM** gegeben werden. **ODEM** ist eine C++-Klassenbibliothek für die Programmierung von Prozesssimulationen. Sie wurde seit 1993 an der Humboldt-Universität zu Berlin entwickelt [FA96]. Die Bibliothek enthält eine Basisbibliothek (`process`), eine Statistik-Bibliothek (`statistics`), eine Bibliothek für die objektorientierte Erstellung von Prozesssimulationen (`discrete`) sowie eine Bibliothek zur Simulation zeitkontinuierlicher Abläufe (`cadsim`).

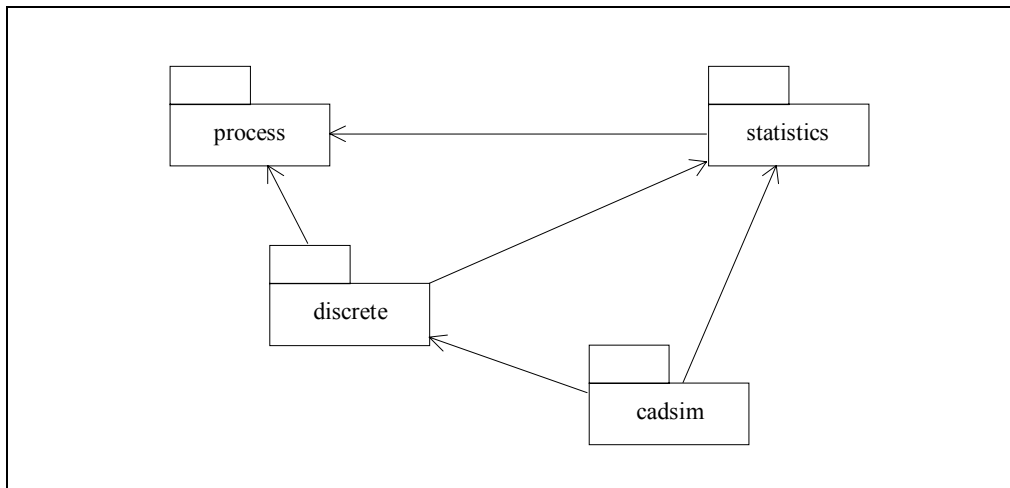


Abbildung 1

Die Bibliothek `process` basiert auf Konzepten von Bjarne Stroustrup (Task-Library, AT&T 1991) und Hansen [Han90]. Sie implementiert portable Koroutinen zur Realisierung einer quasiparallelen Abarbeitung einzelner Prozesse. Darüber hinaus sind Klassen für Prozessverkettung bzw. -synchronisation und ein Konzept für die Prozesskommunikation enthalten.

Die Bibliothek `statistics` bietet verschiedene Generatoren von Pseudozufallszahlen und Hilfsklassen für die statistische Auswertung von Simulationsdaten.

`discrete` bildet die Basis für eine objektorientierte Modellierung. Die an DEMOS (Discrete Event Modelling on Simula) [Bir82] angelehnte Schnittstelle enthält Konzepte wie Prozessobjekte, *Condition*-Warteschlangen, *Master-Slave*-Warteschlangen, Ressourcen und eine gepufferte Kommunikation. Die angebote-

nen Bausteine unterstützen die statistische Auswertung der Ereignisse einer Simulation. `discrete` bietet auch einen Mechanismus an, Mitschriften aus dem Simulationsablauf zu erstellen. `discrete` baut auf `process` auf, versteckt dieses aber mittels privater Vererbung vor dem Nutzer. Damit soll erreicht werden, dass sich der Nutzer auf die objektorientierte Modellierung konzentrieren kann, und sich nicht um die grundlegenden Konzepte von `process` kümmern muss. Darüber hinaus bietet `discrete` ein Beispiel für die Realisierung anwendungsspezifischer Simulationsbibliotheken auf der Basis von `process`.

`cadsim` schließlich bietet eine, auf `discrete` aufbauende, Unterstützung zeitkontinuierlicher Prozesse. Damit ist es möglich, kombinierte oder rein zeitkontinuierliche Modelle zu erstellen. Die Grundlage dafür ist ein fehlersensitiver Integrator mit variablen Zeitschritten, der gewöhnliche Differenzialgleichungen mit wählbarer Genauigkeit berechnet. Die Funktionen werden durch den Benutzer in expliziter Form durch Redefinition einer virtuellen Methode angegeben. `cadsim` unterstützt den Trace Mechanismus aus `discrete` und erweitert ihn um die Ausgabe der Integrationsergebnisse mit einstellbarem Detaillierungsgrad. Darüber hinaus wird eine externe graphische Darstellung der kontinuierlichen Zustandsgrößen durch **Gnuplot** unterstützt. Eine ausführlichere **ODEM**-Dokumentation findet sich in [FA96] und [Ger01].

---

### 3 ODEMx

Im Rahmen dieser Diplomarbeit entstand die Bibliothek **ODEMx**, die auf der Basis der Technologien und Erfahrungen aus **ODEM** neue Lösungen und Ideen umsetzt. In den folgenden Kapiteln werden mehrere Untersuchungen beschrieben, die sich jeweils mit einer neuen Lösung für die Realisierung von C++-Bibliotheken zur Prozesssimulation befassen.

Die erste Untersuchung behandelt eine Designänderung gegenüber **ODEM**, deren Konsequenzen bis zu den Grundlagentechniken für die Realisierung von Prozesssimulationen hinabreichen. Als zweites wird ein Konzept zur Kopplung individueller Objekte mit dem Ziel der Überwachung von Attributen und Operationen sowie die Umsetzung und Anwendung des Konzepts dargelegt. Danach erfolgt die Erläuterung neuer sowie weiterentwickelter Methoden für die Erfassung und Auswertung der Daten einer Simulation. Abschnitt 3.4 schildert, am Beispiel von abstrakten und konkreten Ressourcen, die Umsetzung von Modellbausteinen mit Hilfe von C++-Templates.

#### 3.1 Design

Die Bibliothek **ODEM** hat die Eigenart, eine „Eins-zu-Eins“-Beziehung zwischen Simulation und ausführbarem Programm festzuschreiben. Jedes Programm, das **ODEM** verwendet, kann daher nur genau eine Simulation enthalten. Die Gründe für diese Einschränkung liegen sowohl in der Implementation der Prozesse in **ODEM** als auch in den Vorbildern, an denen sich **ODEM** orientiert. Ein Ergebnis dieser Entscheidung ist, dass Programm und Simulation ineinander verwoben werden und damit eine C++-typische Kapselung nicht stattfindet.

**ODEMx** soll mehrere Simulationen pro Hauptprogramm erlauben, was die Zusammenfassung aller Verwaltungsinformationen einer Simulation in einer Klasse erfordert. Für das Hauptprogramm stellt sich eine Simulation daraufhin im Idealfall als eine abgeschlossene Komponente dar.

Für die Implementierung von Prozessen verwendet **ODEMx** die Techniken aus **ODEM**, die jedoch, aufgrund des veränderten Designs, höheren Anforderungen unterliegen. Als Beispiel für diese Anforderungen, ist die Prozessimplementation mit Hilfe von Stack-Manipulationen zu nennen. Ohne an dieser Stelle auf Details einzugehen, soll der folgende Unterschied genannt werden:

In **ODEM** wird das Hauptprogramm bei der Erzeugung des ersten Prozesses automatisch selbst in einen Urprozess verwandelt und in die Simulation eingefügt. Da in **ODEMx** mehrere Simulationen unterstützt werden, muss es auch möglich sein, diese sequentiell überlappend zu berechnen. Dies würde eine gleichzeitige Zugehörigkeit des Hauptprogramms zu verschiedenen Simulationen erfordern. Da dies nicht möglich ist, gehört das Hauptprogramm in **ODEMx** zu keiner der Simulationen, wodurch jedoch Vorkehrungen zum Schutz seines Stack erforderlich sind.

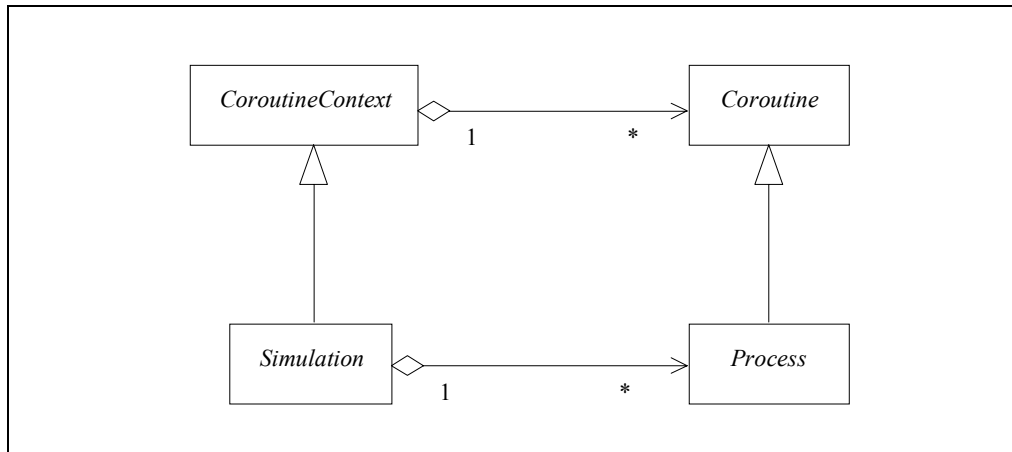
Aufgrund der gestiegenen Komplexität wurde eine Trennung zwischen Prozessen und ihrer Realisierung in C++ eingeführt. Diese erfolgt in **ODEMx** mit Hilfe von Koroutinen, indem man Prozesse durch Ableitung als spezielle Koroutinen umsetzt. Die verschiedenen, unter Umständen nicht portablen, Implementationen von Koroutinen stellen dabei unterschiedliche Realisierungen der Basisklasse dar, wäh-

### 3.1 Design

---

rend für die Programmierung der Prozesse eine einheitliche, vererbte Schnittstelle bereitsteht. Über diese Schnittstelle können Prozesse schließlich ihre Ausführung unterbrechen und andere Prozesse oder das Hauptprogramm aktivieren.

Aus den Designänderungen ergibt sich die in Abbildung 2 dargestellte Struktur. Simulation und Process entsprechen den Simulationen und ihren Prozessen. Coroutine ist die Basisklasse, mit der Koroutinen umgesetzt werden. Die Klasse CoroutineContext fasst mehrere Koroutinen in einem gemeinsamen Koroutinensystem zusammen und verwaltet sie.



**Abbildung 2**

*CoroutineContext* spiegelt die Relation *Process*—*Simulation* auf der Ebene der Koroutinen wider. Da **ODEMx** mehrere voneinander unabhängige Simulationen mit jeweils eigenen Prozessmengen unterstützt, sind auch mehrere voneinander unabhängige Koroutinensysteme notwendig. Jeweils ein solches System bildet die Basis für die Prozesse einer Simulation.

#### 3.1.1 Coroutine / CoroutineContext

Für die nähere Erläuterung der Klassen *Coroutine* und *CoroutineContext* wird zunächst die Schnittstelle von *Coroutine* und ihre Verwendung unabhängig von *Process* und *Simulation* dargestellt. Danach erfolgt eine Beschreibung der verschiedenen Implementationen von *Coroutine* und *CoroutineContext*, die auf Vorbildern aus **ODEM** basieren.

##### 3.1.1.1 Anwendung

Die Schnittstelle von *Coroutine* ist sowohl durch die Anforderungen von *Process* als auch durch die Erfordernisse der verschiedenen Koroutinen-Implementationen geprägt. Die unterschiedlichen Implementationen werden dabei, getrennt durch bedingte Übersetzung unter Verwendung des Präprozessors, in den gleichen Dateien umgesetzt.



```
class Coroutine : public DefLabeledObject, ... {
public:
    enum State {
        CREATED,
        RUNNABLE,
        TERMINATED
    };

    // Öffentliche Schnittstelle
public:
    Coroutine(
        const char* label="",
        CoroutineContext* c=0,
        CoroutineObserver* o=0
    );
    ~Coroutine();

    void switchTo();
    void operator()();

    State getState() const {return state;}
    CoroutineContext* getContext();
    Coroutine* getParent() {return parent;}
    Coroutine* getCaller() {return caller;}

protected:
    // Einsprungpunkt in die Koroutine
    virtual void start() = 0;

    // Implementation
private:
    ...
    void initialize();
    void clear();
    ...

    // Systemabhäng:
#ifdef HAVE_FIBERS
    // Fiber
    // Win32
    LPVOID myFiber;
    LPVOID fiber;

    void saveFiber();
    friend VOID CALLBACK ExecFiber(PVOID);
    friend void FiberSwitch(LPVOID fiber);
#else
    // Stack
    // Linux GCC3.1, Sparc, ...
    static StackAddress adress1;
    static StackAddress adress2;

    StackAddress base;
    StackAddress last;
    StackBuffer oldStack;
    int checksum;
    jmp_buf env;

    bool saveEnvironment();
    void restoreEnvironment();
    ...
#endif
};
```

---

Die Basisklasse `DefLabeledObject` stellt innerhalb eines Kontextes einmalige Bezeichner bereit, die mit `getLabel` abgefragt werden können. Der Kontext wird dabei durch `CoroutineContext` bereitgestellt.

Für die Verwendung von `Coroutine` sind die `public`- und `protected`-Bereiche der Schnittstellendefinition relevant. Der `private`-Abschnitt enthält die, durch den Schalter `HAVE_FIBERS` getrennten, Daten und Funktionen der verschiedenen Implementationen.

Die virtuelle Methode `start` deklariert die Einstiegsfunktion der Koroutine, deren Definition durch den Nutzer, in Ableitungen von `Coroutine` (z.B. `Process`), erfolgt. Aktivierungen von Koroutinen werden durch die Funktion `switchTo` oder den Funktionsoperator ausgelöst.

Koroutinen haben ein zustandsabhängiges Verhalten, in dem sich drei Grundzustände unterscheiden lassen. Der erste Zustand repräsentiert den Zeitraum zwischen der Erzeugung der Koroutine und ihrer ersten Ausführung. Während dieser Phase führt eine Aktivierung zum Sprung an den Anfang der Einstiegsfunktion und zum Wechsel in den zweiten Grundzustand. Der zweite Zustand repräsentiert die Abarbeitungsphase, die bis zum Ende der Einstiegsfunktion oder einer ausdrücklichen Terminierung andauert. Während dieses Zeitraums kann die Koroutine wiederum aktiv oder unterbrochen sein, wobei die Aktivierung einer unterbrochenen Koroutine zum Sprung an den Unterbrechungspunkt führt. Nach Rückkehr der Koroutinenfunktion, oder einer ausdrücklichen Terminierung der Koroutine, beginnt die letzte Phase. Die Koroutine ist nun abgearbeitet und lässt sich nicht mehr aktivieren.<sup>1</sup> Objekte der Klasse `Coroutine` durchlaufen die entsprechenden Zustände `CREATED`, `RUNNABLE` und `TERMINATED`, während ihr aktueller Zustand durch die Funktion `getState` abfragbar ist.

Koroutinen stehen zueinander oder zur Umgebung in einer *Caller*- und einer *Parent*-Beziehung. Der *Caller* einer Koroutine ist der direkte Vorgänger in der Ausführung des Programms. Dies kann sowohl ein `Coroutine`-Objekt als auch die Umgebung des Koroutinensystems sein. Der *Caller* einer Koroutine wechselt während der Ausführung. *Parent* ist der erste Rufer der Koroutine, der den Übergang vom Zustand `CREATED` nach `RUNNABLE` bewirkt hat. In einem System, in dem beispielsweise drei Koroutinen A, B und C interagieren und der erste Aufruf von B aus der Koroutine A heraus erfolgt, wird A zum *Parent* von B, unabhängig davon, ob A das Objekt B erzeugt hat. Wenn B im Verlauf der Interaktion aus C aktiviert wird, nimmt diese Koroutine für einen begrenzten Zeitraum die Position des *Caller* von B ein.

Die Erzeugung einer Koroutine geschieht immer mit Verweis auf einen Kontext. Wenn der Nutzer jedoch keinen eigenen Kontext angibt, wird automatisch ein von der Bibliothek erzeugter einmaliger `DefaultContext` verwendet. Dieser lässt sich in **ODEMx** auch durch die Funktion `getDefaultContext` ermitteln. Dabei ist zu bedenken, dass pro Programm höchstens ein `DefaultContext` existiert.

Innerhalb eines `CoroutineContext` ist maximal eine Koroutine aktiv, die durch `getActiveCoroutine` erfragt werden kann. Der Kontext sichert den Rücksprung-

---

<sup>1</sup> Alternativ wäre auch die Rückkehr in den ersten Zustand möglich. Für den Anwendungsfall Prozesssimulation ist jedoch die Terminierung typisch.

punkt in die Umgebung der Koroutinen, der von allen Koroutinen des Kontextes verwendet werden kann, um die Ausführung an die Umgebung zurückzugeben.

Es können mehrere `CoroutineContext`-Objekte parallel arbeiten, wobei die Ausführung der verschiedenen Koroutinen weiterhin sequentiell abläuft. Der Sprung von einer Koroutine **A** in eine Koroutine eines anderen Kontextes **K** führt dazu, dass Kontext **K** als Rücksprungpunkt **A** sichert und somit die Schachtelung von Koroutinensystemen ermöglicht.

Das folgende Beispiel basiert auf [Hel99] und stellt eine *Mergesort*-Implementierung mit Koroutinen dar. Zwei Instanzen der Koroutine `Traverser` füllen abwechselnd das Zielfeld `c` aus den Feldern `a` und `b`.

```
int a[]={1, 5, 6, 8, 10, 12, 15, 17};
int b[]={2, 4, 7, 9, 11, 13, 14, 18, 20, 30};
int* c;
int m=sizeof(a)/sizeof(int), n=sizeof(b)/sizeof(int), cIndex;

class Traverser : public Coroutine {
public:
    Traverser(int a[], int l) :
        array(a), limit(l), index(0) {};

    int *array, limit, index;
    Traverser* partner;

    virtual void start() {
        while (index < limit) {
            if (partner->array[partner->index]<array[index])
                partner->switchTo();
            c[cIndex++]=array[index++];
        }
        while (cIndex < m+n)
            c[cIndex++]=partner->array[partner->index++];
    }
};

int main(int argc, char* argv[]) {
    // Erzeugung und Initialisierung der Koroutinen
    Traverser* x=new Traverser(a, m);
    Traverser* y=new Traverser(b, n);
    x->partner=y; y->partner=x;

    // Erzeugung und Initialisierung des Ergebnisfeldes
    c=new int[m+n];
    cIndex=0;

    // Aktivierung der ersten Koroutine
    // Alternative: verwendung des Funktionsoperators x()
    x->switchTo();

    // Ausgabe des Ergebnis
    for (int j=0; j<m+n; ++j)
        cout << c[j] << ' ';
    cout << endl;

    return 0;
}
```

Da kein eigener `CoroutineContext` erzeugt wurde, verwenden die Koroutinen den Kontext `DefaultContext`.

Coroutine und CoroutineContext lassen sich, wie im dargestellten Beispiel, unabhängig von Process und Simulation verwenden, was ein separates Testen und Entwickeln ermöglicht. Durch den angebotenen DefaultContext lassen sich einfache Beispiele umsetzen.

#### 3.1.1.2 Stack-basierte Implementation

Der Stack-basierten Implementation von Koroutinen liegen Manipulationen des Laufzeit-Stack eines Simulationsprogramms sowie die Sicherung und Restaurierung des Prozessorzustandes zugrunde. Wenn eine Koroutine unterbrochen wird, muss man ihren Ausführungszustand sichern, der typischerweise Prozessorzustand und Laufzeit-Stack beinhaltet (siehe [FA96]).

Für die Sicherung und Restaurierung des Prozessorzustandes stellt die C-Laufzeitbibliothek die Funktionen `setjmp` und `longjmp` zur Verfügung. Mit `setjmp` werden die wichtigsten Register des Prozessors, insbesondere der Programmzähler `IP`, in einem Puffer gesichert. Danach kehrt `setjmp` mit dem Rückgabewert 0 zurück. Die zweite Funktion (`longjmp`) schreibt die Werte des Puffers in die Register des Prozessors zurück, wobei durch die vorgenommene Veränderung des Programmzählers der Prozessor seine Arbeit in `setjmp` fortsetzt. Aus Sicht eines C++-Programms führt dies zu einer erneuten Rückkehr aus dieser Funktion. Um die zweite Rückkehr der Funktion `setjmp` von der ersten unterscheiden zu können, erlaubt `longjmp` die Angabe eines Rückgabewertes für `setjmp`. Das folgende Beispiel zeigt, wie anhand des Rückgabewertes verschiedene Ausführungszweige durchlaufen werden:

```
...
jmp_buf env;

switch (setjmp(env)) {
case 0:
    // 1.
    longjmp(env, 1);
    break;
case 1:
    // longjmp von 1.
    // 2.
    longjmp(env, 2);
    break;
case 2:
    // longjmp von 2.
    break;
}
...
```

Die Speicherung des Laufzeit-Stack erfordert zunächst die Festlegung des für die Koroutine relevanten Bereichs. Deshalb muss man vor der Ausführung der Koroutinen ihren Basispunkt im Stack ermitteln. Alle lokalen Variablen und Funktionsaufrufe werden ab diesem Punkt im Stack angelegt. Das Ende des zu sichernden Bereiches ergibt sich entsprechend aus der Adresse der zuletzt angelegten lokalen Variable. Der so festgelegte Bereich wird in einen Puffer kopiert und für spätere Konsistenzprüfung mit einer Prüfsumme gesichert.

Der Rücksprung in eine Koroutine beginnt mit der Restauration ihres Laufzeit-Stack. Ein erster Schritt, um dies zu erreichen, ist die Überprüfung der Integrität

des gespeicherten Bereichs anhand der Prüfsumme. Danach muss sichergestellt sein, dass im aktuellen Stack genug Platz für den zu restaurierenden Bereich vorhanden ist. Gegebenenfalls kann man den aktuellen Stack durch rekursive Funktionsaufrufe aufblähen, wobei die Funktion, mit der eine Restauration des Ausführungszustandes erfolgt, ihre lokalen Variablen durch zusätzliche Rekursionen über den zu überschreibenden Bereich retten muss. Nach dem Kopieren des Pufferinhalts in den Stack erfolgt abschließend mit `longjmp` der Rücksprung zum passenden Aufruf der Funktion `setjmp`.

Neben der Handhabung von Koroutinen muss man auch die Umgebung des Koroutinensystems betrachten. Wenn von der Umgebung in eine Koroutine gesprungen wird, kann unter Umständen der momentane Stack der Umgebung gefährdet sein. Der gefährdete Bereich liegt dabei zwischen dem Stack-Anfang der „untersten“ Koroutine und dem aktuellen Ende des Stack. Die Rückkehr zur Umgebung wird schließlich durch die Sicherung des Prozessorzustandes beim Sprung in eine Koroutine ermöglicht.

Der Laufzeit-Stack wird in verschiedenen Rechnerarchitekturen unterschiedlich gehandhabt. In einigen Architekturen wächst der Stack von niedrigen Speicheradressen zu höheren, in anderen in umgekehrter Richtung. Die beiden Varianten lassen sich jedoch während der Programmausführung erkennen und unterschiedlich behandeln. **ODEMx** orientiert sich dabei an der in **ODEM** realisierten Technik.

Für das Funktionieren der Stack-basierten Implementation von Koroutinen müssen einige Voraussetzungen erfüllt sein. Erstens muss die jeweilige Rechnerarchitektur einen zusammenhängenden Laufzeit-Stack verwenden, der nicht vor Manipulationen geschützt ist. Zweitens ist eine Laufzeitbibliothek mit C++-kompatiblen `setjmp`- und `longjmp`-Funktionen, die sich nicht an Stack-Manipulationen stören, erforderlich. Und drittens müssen lokale Variablen und Funktionsrufe auch tatsächlich im Stack gespeichert sein.

#### 3.1.1.3 Anpassung der Stack-Implementation an Sparc-Rechner<sup>1</sup>

Die **Sparc**-Rechnerarchitektur von SUN verletzt die dritte Bedingung für das Funktionieren der Stack-basierten Implementation. **Sparc**-Prozessoren spiegeln Teile des Laufzeit-Stack eines Programms aus Effizienzgründen in Prozessorregistern. Ein Update des Stack-Inhaltes erfolgt dabei nur unter bestimmten Bedingungen oder durch einen speziellen Prozessorbefehl. Deshalb kann es beim einfachen Sichern des Stack dazu kommen, dass man veraltete Daten speichert.

Martin von Löwis hat dieses Problem bereits für **ODEM** behoben. In [vLo92] beschreibt er die **Sparc** Architektur und seine Lösung genauer. Sie besteht im manuellen Update des Stack durch den Prozessorbefehl `asm("t 3")`. Auch **ODEMx** nutzt dieses Verfahren für die Anpassung der Stack-Implementation an **Spar**-Rechner.

#### 3.1.1.4 Win32-Implementation

Microsofts `setjmp`- und `longjmp`-Implementationen sind nicht für das C++-Objekt-Modell ausgelegt und vertragen sich nicht mit den Stack-Manipulationen von **ODEMx**, wodurch die Stack-Implementation unter Windows nicht funktio-

---

<sup>1</sup> Die Ausführungen beziehen sich auf Solaris. Mögliche Unterschiede zu anderen Betriebssystemen, die ebenfalls auf Sparc Rechnern laufen, wurden nicht untersucht.

### 3.1 Design

---

niert. Microsoft bietet jedoch die Alternative *Fiber*, mit deren Hilfe sich Koroutinen realisieren lassen. Die Verwendung von *Fiber* in **ODEMx** stützt sich ebenfalls auf die von Martin von Löwis realisierte *Fiber*-basierte Prozess-Implementation für **ODEM**.

*Fiber* sind Ausführungseinheiten die manuelles Scheduling erfordern und sequentiell innerhalb des Thread abgearbeitet werden, der sie verwaltet. Sie bieten ähnliche Eigenschaften wie Koroutinen und lassen sich leicht für deren Umsetzung nutzen. Die Verwendung von *Fiber* erfordert dabei zunächst die Umwandlung des aktiven Win32-Thread in eine *Fiber*. Das erfolgt mit der Funktion:

```
LPVOID ConvertThreadToFiber(LPVOID lpParameter)
```

Der Funktion kann ein Parameter übergeben werden, der innerhalb der *Fiber* mit der folgenden Funktion abgefragt werden kann:

```
PVOID GetFiberData()
```

Erst nachdem der aktive („*Fiber*-lose“) Thread durch *ConvertThreadToFiber* für das Scheduling der *Fiber* vorbereitet wurde, lassen sich weitere *Fiber* mit der *CreateFiber*-Funktion erzeugen:

```
LPVOID CreateFiber(DWORD dwStackSize, lpStartAddress,  
LPVOID lpParameter)
```

Die Funktion erwartet eine globale Funktion als Einstiegspunkt in die *Fiber* sowie die *Fiber*-Daten. Um von der globalen Funktion wieder zu einer Coroutine-Klasse zu gelangen, wird *lpParameter* für die Übergabe des *this*-Zeigers der Coroutine verwendet. Optional kann die Anfangsgröße des Stack, mit dem die *Fiber* arbeiten, angegeben werden. Eine Übergabe von 0 führt dabei zur Verwendung der Stack-Größe des Hauptthread. Der Stack wird aber in jedem Fall bei Bedarf automatisch vergrößert.

Der Wechsel zu einer *Fiber* erfolgt schließlich mit der Funktion *SwitchToFiber*.

```
VOID SwitchToFiber(LPVOID lpFiber)
```

*Fiber* werden ab Windows 98 unterstützt.

#### 3.1.2 Process / Simulation

In diesem Abschnitt wird zunächst die Anwendung der Klassen *Process* und *Simulation* in mit **ODEMx** entwickelten Simulationen beschrieben.

Anhand von Beispielen wird gezeigt, wie mit **ODEMx**

- einfache Simulationen,
- mehrere Simulationen parallel oder sequentiell und
- rekursive (geschachtelte) Simulationen

durchgeführt werden können.

Danach folgt die Erläuterung des Scheduling von Prozessen und der Berechnung von Simulationen in **ODEMx**.

### 3.1.2.1 Anwendung

Auf der Basis von `Coroutine` und `CoroutineContext` werden die Klassen `Process` und `Simulation` definiert. `Process` ist die Basisklasse für alle nutzerdefinierten Prozesse in einer Simulation.

```
typedef double Priority;
...
class Process : public Coroutine, ... {
public:
    // Funktionstypen für die Kodierung von
    // Bedingungen und Auswahlverfahren:
    typedef bool(*Condition)(Process* owner);
    typedef bool(*Selection)(Process* owner, Process* partner);

    // Prozesszustand
    enum State {CREATED, CURRENT, RUNNABLE, IDLE, TERMINATED};

    // Öffentliche Schnittstelle
public:
    Process(Simulation* s, Label l = "", ProcessObserver* o = 0);
    ~Process();

    void activate();
    void activateIn(SimTime t);
    void activateAt(SimTime t);
    void activateBefore(Process* p);
    void activateAfter(Process* p);

    void hold();
    void holdFor(SimTime t);
    void holdUntil(SimTime t);

    void sleep();

    void cancel();

    State getState() const;

    Priority getPriority() const;
    Priority setPriority(Priority newPriority);

    SimTime getExecutionTime() const;

    Process* getCurrentProcess();
    SimTime getCurrentTime();
    Simulation* getSimulation();

    bool hasReturned() const;
    int getReturnValue() const;

    virtual Trace* getTrace() const;

protected:
    // Schnittstelle für die Definition
    // von Prozessen:
    virtual int main() = 0;
    ...
};
```

`Process` deklariert als Einstiegspunkt die Funktion `main`. Benutzerprozesse implementieren diese Funktion mit ihrem individuellen Verhalten. Um einen Prozess

### 3.1 Design

---

zu aktivieren, muss man diesen innerhalb seiner Simulation mit den Funktionen `activate...` und `hold...` für eine Ausführung einplanen (*Scheduling*). Dabei wird der Prozess in den Ereigniskalender einsortiert. Sind für einen Zeitpunkt mehrere Prozesse gleichzeitig zur Ausführung vorgesehen, muss eine Zwangsserialisierung erfolgen. Die Details des Scheduling werden im nächsten Abschnitt beschrieben.

Auch ein Prozess hat, wie eine Koroutine, mehrere Grundzustände. Nach seiner Erzeugung und vor der ersten Aktivierung ist ein Prozess im Zustand `CREATED`. Wird der Prozess für die Ausführung eingeplant, wechselt er in den Zustand `RUNNABLE`. Der momentan aktuelle Prozess ist als einziger einer Simulation im Zustand `CURRENT`. Wenn ein Prozess auf externe Ereignisse wartet, wird er in den Zustand `IDLE` versetzt. Nachdem ein Prozess seine Aufgaben erfüllt hat oder abgebrochen wurde, ist er im Zustand `TERMINATED`. Der momentane Zustand eines Prozesses kann mit der Funktion `getState` abgefragt werden.

Die Prozesse einer Simulation werden, wie alle anderen Modellkomponenten in **ODEMx**, einer Instanz von `Simulation` zugeordnet. `Simulation` verwaltet die Prozesse und Modellbausteine einer Simulation und bietet Funktionen zur Steuerung der Simulationsberechnung. Zusätzlich werden von `Simulation` Dienste für die Erfassung von Simulationsdaten und die Erzeugung eindeutiger Bezeichner angeboten.

```
class Simulation :
    public ExecutionList,
    public CoroutineContext, ...
{
    // Öffentliche Schnittstelle
public:
    Simulation(Label l="Simulation", SimulationObserver* o=0);
    ~Simulation();

    void run();
    void step();
    void runUntil(SimTime t);

    bool isFinished();

    virtual void exitSimulation();

    Process* getCurrentProcess();
    std::list<Process*>& getCreatedProcesses();
    std::list<Process*>& getRunnableProcesses();
    std::list<Process*>& getIdleProcesses();
    std::list<Process*>& getTerminatedProcesses();

    virtual SimTime getTime() const;

protected:
    virtual void initSimulation() = 0;

    ...
};
```

Die Berechnung einer Simulation wird in den Funktionen `run`, `step` und `runUntil` ausgeführt. Die Funktion `run` kehrt erst nach Abschluss der Simulation zurück,



während `step` bis zum nächsten Prozesswechsel<sup>1</sup> und `runUntil` bis zum Erreichen der angegebenen Zeit rechnet.

Die Verwendung der **ODEMx**-Bibliothek für die Programmierung einer Simulation beinhaltet im Allgemeinen die Definition einer von `Simulation` abgeleiteten und modellspezifischen Simulationsklasse. Diese Klasse definiert die Funktion `initSimulation`, um die Modellkomponenten zu erzeugen und zu initialisieren. Ohne eine benutzerdefinierte Simulation kommt eine vordefinierte `DefaultSimulation` Klasse zum Einsatz. Ein Objekt dieser Klasse wird von **ODEMx** bei Bedarf erzeugt, wobei für jedes Programm maximal ein solches Objekt existiert.

`DefaultSimulation` hat eine leere `initSimulation` Funktion. Bei der Verwendung dieser Klasse müssen deshalb alle Initialisierungen des Modells im Hauptprogramm geschehen. Durch `DefaultSimulation` lassen sich in **ODEMx** Simulationen ungekapselt, wie unter **ODEM**, programmieren.

### 3.1.2.2 Anwendungsbeispiele

Im folgenden werden drei Beispiele für die Nutzung der **ODEMx** Bibliothek vorgeführt. Das erste Beispiel zeigt ein einfaches System, welches aus einem einzigen Timer-Prozess besteht. Dieser Prozess gibt regelmäßig ein (Zeit-)Zeichen aus:

```
#include <odemx/base/Process.h>
#include <iostream>

using namespace std;
using namespace odemx;

class TimerA : public Process {
public:
    TimerA(Simulation* sim) : Process(sim, "TimerA") {}

    virtual int main() {
        while (true) {
            holdFor(1.0);

            cout << '.';

        }
        return 0;
    }
};

int main(int argc, char* argv[]) {
    TimerA* myTimer=new TimerA(getDefaultSimulation());

    myTimer->activate();

    cout << "basic simulation example" << endl;
    cout << "===== " << endl;

    for (int i=1; i<5; ++i) {
        cout << i << ". step time="
            << getDefaultSimulation()->getTime() << endl;
    }
}
```

---

<sup>1</sup> Die Funktion `step` veranlasst die Ausführung des ersten Prozesses im Ereigniskalender. Sobald dieser Prozess seine Aktionen unterbricht, z.B. durch Aktivierung eines anderen Prozesses, kehrt die Funktion `step` zurück.

```

        getDefaultSimulation()->step();
        cout << endl;
    }

    cout << endl;
    cout << "continue until SimTime 13.0 is reached or passed:";
    cout << endl;

    // Simulation bis die Modellzeit 13.0 verstrichen ist
    // Die Alternative ...->run() würde in diesem Fall zu einer
    // endlosen Simulation führen.
    getDefaultSimulation()->runUntil(13.0);

    cout << endl << "time=";
    cout << getDefaultSimulation()->getTime() << endl;
    cout << "======" << endl;

    return 0;
}

```

Das Programm verwendet `DefaultSimulation` und muss daher die Erzeugung und Aktivierung in `main` vornehmen (`myTimer->activate()`). Die Berechnung wird zunächst schrittweise durch `getDefaultSimulation()->step()` und danach bis zum Zeitpunkt 13.0 mit `getDefaultSimulation()->runUntil(13.0)` durchgeführt. Der Prozess `TimerA` verwendet die Methode `holdFor`, um sich selbst für jeweils eine Zeiteinheit zu unterbrechen.

Das zweite Beispiel demonstriert die parallele und die sequentielle Durchführung von mehreren Simulationen innerhalb eines Programms. Dafür werden zwei Simulationen in speziellen Simulationsklassen definiert. Die erste Simulation besteht aus zwei Prozessen, die sich mithilfe einer *Master-Slave*-Warteschlange synchronisieren. Die beiden Prozesse werden als innere Klassen realisiert. Der erste Prozess (Master) fordert mit der Methode `coopt` an der Warteschlange einen *Slave*-Prozess an, während sich der zweite Prozess (Slave) mit der Methode `wait` als *Slave* synchronisiert.

```

// Simulation A
class SimA : public Simulation {
    // process types of SimA
    class Master : public Process {
    public:
        Master(SimA* sim, ProcessObserver* o = 0)
            : Process(sim, "Master", o) {}

    protected:
        virtual int main() {
            SimA* sim=dynamic_cast<SimA*>(getSimulation());
            Slave* s=0;

            cout << getLabel() << " coopt" << endl;
            s=dynamic_cast<Slave*>
                (sim->getQueue()->coopt());
            s->activateAfter(this);
            cout << getLabel() << " coopt finished" << endl;

            return 0;
        }
    };

    class Slave : public Process {
    public:

```

---

```

        Slave(SimA* sim, ProcessObserver* o = 0)
            : Process(sim, "Slave", o) {}

protected:
    virtual int main() {
        SimA* sim=dynamic_cast<SimA*>(getSimulation());

        cout << getLabel() << " wait" << endl;
        sim->getQueue()->wait();
        cout << getLabel() << " wait finished" << endl;

        return 0;
    }
};

Master* m;
Slave* s;
Waitq* q;

public:
    SimA(SimulationObserver* o = 0)
        : Simulation("SimA", o), m(0), s(0), q(0) {}

    ~SimA() {
        delete m;
        delete s;
        delete q;
    }

    Waitq* getQueue() {return q;}

protected:
    virtual void initSimulation() {
        m = new Master(this);
        s = new Slave(this);
        q = new Waitq(this, "queue");

        m->activate();
        s->activate();
    }
};

```

Die nachfolgende Simulation enthält lediglich einen Prozess, der sich selbst mehrfach hintereinander mit der Methode `holdFor` verzögert.

```

// Simulation B
class SimB : public Simulation {
    // process types of SimA
    class Loop : public Process {
    public:
        Loop(SimB* sim, ProcessObserver* o = 0)
            : Process(sim, "Loop", o) {}

    protected:
        virtual int main() {
            for (int i= 0; i<10; ++i) {
                cout << getLabel() << i << endl;
                holdFor(1.0);
            }
            return 0;
        }
    };

    public:
        SimB(SimulationObserver* o = 0)

```

---

```
        : Simulation("SimB", o), l(0) {}
~SimB() {
    delete l;
}
Loop* l;

protected:
    virtual void initSimulation() {
        l=new Loop(this);
        l->activate();
    }
};
```

Die main Funktion des Beispiels führt zunächst die erste Simulation in einer Schleife wiederholt aus (Part 1). Danach wird die zweite Simulation durchgeführt (Part 2). Schließlich werden die beide Simulationen abwechselnd schrittweise berechnet (Part 3).

```
int main(int argc, char* argv[]) {
    char c[2];

    cout << "Part 1" << endl;
    for (int i=0; i<5; ++i) {
        SimA s;
        s.run();
    }
    cout << "Press [Enter] to continue";
    cin.getline(c, 1);

    cout << "Part 2" << endl;
    {
        SimB s;
        s.run();
    }
    cout << "Press [Enter] to continue";
    cin.getline(c, 1);

    cout << "Part 3" << endl;
    {
        SimA s1;
        SimB s2;

        s1.step();
        s2.step();

        while (!s1.isFinished() ||
               !s2.isFinished()) {

            if (!s1.isFinished())
                s1.step();

            if (!s2.isFinished())
                s2.step();

        }
    }

    cout << "Finished" << endl;
    return 0;
}
```

Wenn man das Beispiel übersetzt und ausführt erhält man die folgende Ausgabe:

```
Part 1
Slave wait
Master coopt
Master coopt finished
Slave wait finished
Slave wait
Master coopt
Master coopt finished
Slave wait finished
Slave wait
Master coopt
Master coopt finished
Slave wait finished
Slave wait
Master coopt
Master coopt finished
Slave wait finished
Press [Enter] to continue
Part 2
Loop0
Loop1
Loop2
Loop3
Loop4
Loop5
Loop6
Loop7
Loop8
Loop9
Press [Enter] to continue
Part 3
Slave wait
Loop0
Master coopt
Loop1
Master coopt finished
Loop2
Slave wait finished
Loop3
Loop4
Loop5
Loop6
Loop7
Loop8
Loop9
Finished
```

Das abschließende Beispiel soll verschachtelte Simulationen vorführen. Dafür wird eine Simulation `Matroschka` definiert. Diese enthält einen Prozess (`Proc`) der eine neue Simulation `Matroschka` anlegt und durchführt. Für die Einschränkung dieser Rekursion wird ein Parameter für die Simulation eingeführt (`mLevel`).

```
#include <odemx/base/Process.h>
#include <iostream>

using namespace std;
using namespace odemx;
```

```

class Matroschka : public Simulation {
    // process types of Matroschka
    class Proc : public Process {
    public:
        Proc(Matroschka* sim, ProcessObserver* o = 0)
            : Process(sim, "Proc", o) {}

    protected:
        virtual int main() {
            int i;
            Matroschka* sim=dynamic_cast<Matroschka*>(
                getSimulation());
            int level=sim->getLevel();

            for (i=level; i>0; --i)    cout << ' ';
            if (level%2>0) cout << "Di " << level << endl;
            else cout << "Da " << level << endl;

            if (sim->getLevel()>0) {
                Matroschka m(sim->getLevel()-1);
                m.run();
            }

            for (i=level; i>0; --i) cout << ' ';
            if (level%2>0) cout << "Di " << level << endl;
            else cout << "Da " << level << endl;

            return 0;
        }
    };

    Proc* p;
    unsigned int mLevel;
public:
    Matroschka(unsigned int level, SimulationObserver* o=0)
        : Simulation("Matroschka", o), mLevel(level) {}
    ~Matroschka() {
        delete p;
    }
    unsigned int getLevel() {return mLevel;}
protected:
    virtual void initSimulation() {
        p = new Proc(this);
        p->activate();
    }
};

int main(int argc, char* argv[]) {
    Matroschka m(10);
    m.run();

    return 0;
}

```

Durch die Definition spezieller Simulationsklassen erreicht man eine Kapselung der Simulationen. Diese ermöglicht, wie in den letzten beiden Beispielen vorgeführt, Rekursionen und mehrere Simulationen innerhalb eines Programms. Einfache Simulationen oder Umsetzungen aus ODEM werden durch die *Default-Simulation* erleichtert.

### 3.1.2.3 Scheduling und Berechnung einer Simulation

Eine wesentliche Kernkomponente der Prozesssimulation ist der Ereigniskalender. Mit seiner Hilfe wird festgelegt, wann welcher Prozess innerhalb der Modellzeit Änderungen am Systemzustand vornehmen kann.

Anders als die zugrundeliegenden Koroutinen lassen sich Prozesse nicht mehr direkt, durch Aufruf der Startfunktion, aktivieren. Stattdessen wird ein Prozess an einem bestimmten Zeitpunkt im Kalender eingetragen. Wenn im Laufe der Simulation vor diesem Eintrag kein weiterer Prozess auf seine Ausführung wartet, setzt **ODEMx** die Modellzeit auf den Zeitpunkt des Eintrags und aktiviert den Prozess durch einen Sprung in seine Koroutine. Beendet oder unterbricht der Prozess seine Aktionen, geht die Simulation zum nächsten geplanten Prozess, bis entweder kein weiterer Prozess zur Ausführung bereit steht oder die Simulation manuell abgebrochen wurde.

Die Verwendung von Koroutinen führt zu einer Serialisierung der Prozesse. Es kann zu jedem Zeitpunkt immer nur ein Prozess seine Aktionen ausführen. Das hat den Vorteil, dass der aktive Prozess exklusiven Zugang zum System hat, führt aber zu Konflikten, wenn mehrere Prozesse ihre Aktionen gleichzeitig durchführen wollen. In diesem Fall muss eine Zwangsserialisierung stattfinden, für die zunächst die Priorität der Prozesse ausgenutzt wird. Je höher die Priorität eines Prozesses ist, desto eher wird dieser ausgeführt. Bei Prozessen gleicher Priorität ist die Art der Einordnung der Prozesse in den Ereigniskalender entscheidend. Sie kann nach dem Prinzip *FirstInFirstOut* (FIFO) oder *LastInFirstOut* (LIFO) sowie explizit vor bzw. nach einem Partnerprozess erfolgen.

Die Struktur des Ereigniskalenders stellt sich zusammengefasst wie folgt dar. Auf einer Zeitachse werden Ereigniszeitpunkte abgetragen, zu denen Prozesse ihre Aktionen ausführen und damit den Systemzustand verändern. Wenn mehrere Prozesse für den gleichen Zeitpunkt eingeplant sind, werden sie in einer zusätzlichen Liste eingeordnet. Die Reihenfolge in dieser Liste bestimmt die Ausführungsfolge dieser Prozesse.

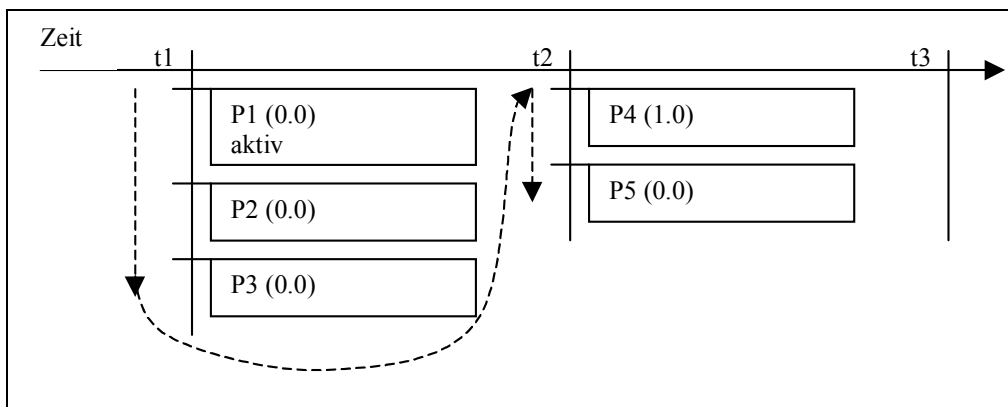


Abbildung 3

Der erste Prozess am ersten Ereigniszeitpunkt im Ereigniskalender (in Abbildung 3 P1 zum Zeitpunkt  $t_1$ ), ist der aktive Prozess einer Simulation. Hat er seine `main`-Funktion abgearbeitet oder unterbrochen, wechselt die Ausführung zum nächsten Prozess (Prozesswechsel).

### 3.1 Design

In **ODEMx** wird das Scheduling durch die Klasse `ExecutionList` realisiert. Diese Klasse implementiert einen Ereigniskalender zusammen mit den benötigten Operationen. `ExecutionList` wird durch den Nutzer jedoch nicht direkt verwendet. Der Anwender der **ODEMx** Bibliothek ruft die `activate...` und `hold...` Funktionen von `Process` auf, um die Ausführung zu steuern. Die Funktion `sleep` kann darüber hinaus einen Prozess in Wartestellung (IDLE) versetzen und dadurch aus dem Ereigniskalender entfernen. Die Klasse `Simulation` steuert mit Hilfe der Klasse `ExecutionList` die Berechnung der Simulation. Abhängig vom Simulationsmodus (`step`, `runUntil`, `run`) kann `Simulation` während eines Prozesswechsels sofort den nächsten Prozess aktivieren oder zum Hauptprogramm zurückkehren.

Die Funktionen `activate`, `activateIn` und `activateAt` fügen den jeweiligen Prozess nach der LIFO-Strategie ein, während `hold`, `holdFor` und `holdUntil` FIFO umsetzen. Dabei beziehen sich `activate` und `hold` auf den aktuellen Zeitpunkt. In Abbildung 4 wird dargestellt, wie Prozess `P1` durch einen Aufruf von `hold` an das Ende der Liste am Zeitpunkt `t1` versetzt wird.

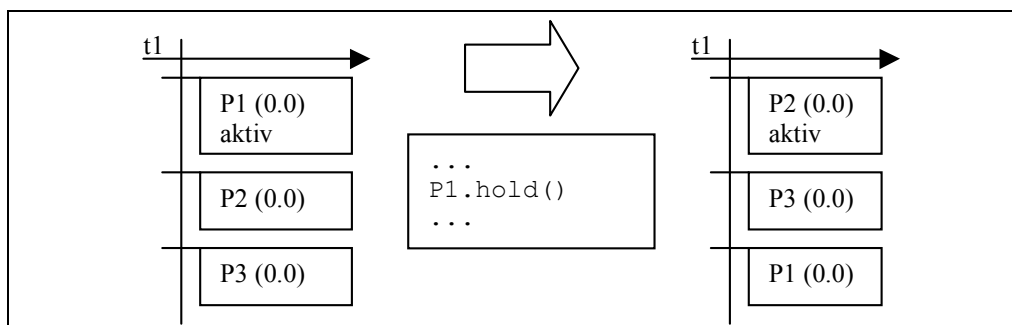


Abbildung 4

#### 3.1.3 Bewertung der Designentscheidungen

Die von den Designänderung erwarteten Ergebnisse wurden erreicht. Im Gegensatz zu **ODEM** können Simulationen in **ODEMx** gekapselt und unabhängig von ihrer Umgebung berechnet werden. Dadurch wird die parallele oder sequentielle Simulation mehrerer Modelle ebenso wie die Schachtelung von Simulationen ermöglicht. Bei Einschränkung auf eine Simulation (`DefaultSimulation`) lässt sich jedoch eine **ODEM**-ähnliche Verwendungsweise nachbilden.

Ein möglicher Nachteil der Änderung sind zusätzliche Funktionsaufrufe bei der Berechnung der Modelle. In Abhängigkeit von den Optimierungsfähigkeiten der C++-Compiler können **ODEMx**-Simulationen daher länger dauern als vergleichbare **ODEM**-Simulationen. Konkrete Messungen wurden nicht durchgeführt.

Von der Kapselung der Simulationen sind nicht nur die Basistechnologien, wie z.B. die Koroutinen-Implementationen, betroffen. Auch andere Dienstleistungen von **ODEM**, beispielsweise die Erzeugung einmaliger Bezeichner für Modellbausteine oder die Mechanismen für die Erfassung der Daten über Simulationen (siehe 3.3 Erfassung und Auswertung der Daten), mussten für ihre Umsetzung in **ODEMx** angepasst werden.



### 3.2 Objekt-Objekt-Kopplung

ODEMX bietet ein Konzept, mit dem die Kopplung von Objekten unterstützt werden soll. Die grundlegende Idee ist, Schnittstellen zur Überwachung der Instanzen bestimmter Klassen bereit zu stellen. Diese Schnittstellen folgen dabei gemeinsamen Konventionen, um ihre Verwendung zu erleichtern. Das umgesetzte Konzept wird *Observation* genannt.

Die Vorlage des Konzepts bildet das in Abschnitt 2.2 SimBeans beschriebene Komponentenframework, dessen Grundlage die JavaBeans-Komponentenarchitektur ist. JavaBeans-Komponenten bieten zwei Möglichkeiten zur Kopplung. Die erste Möglichkeit ist die Kommunikation mit Hilfe von Ereignissen. Die Quellkomponente sendet dabei Ereignisse an registrierte Klienten. Im Programm spiegeln sich solche Ereignisse als Funktionsaufrufe wider. Die zweite Möglichkeit liegt in der Überwachung ausgewählter Attribute einer Quellkomponente. Sie verwendet wiederum spezielle Ereignisse, um Änderungen an registrierte Klienten zu übermitteln.

Im Bereich der Simulation lassen sich solche Kopplungsverfahren auf verschiedene Weise anwenden. Sie können beispielsweise verwendet werden, um Interaktionen zwischen Modellbausteinen zu modellieren. Der für die Entwicklung von *Observation* entscheidende Anwendungsbereich ist jedoch die Erfassung und Auswertung von Daten über die Simulation, der im Abschnitt 3.3.3.3 Anwendung von *Observation* beschrieben wird.

Im Folgenden wird zunächst die Struktur von *Observation* erläutert. Danach werden Details der Realisierung beschrieben. Abschließend wird eine Bewertung des Konzepts versucht.

#### 3.2.1 Struktur von Observation

Im Konzept *Observation* gibt es zwei Rollen, die ein Objekt spielen kann. Die erste Rolle ist die eines *Observable*, eines überwachten Objektes. Die zweite Rolle ist die eines *Observer*. *Observable* senden alle ihre Ereignisse an die registrierten *Observer*.

Damit ein Objekt die Rolle eines *Observable* einnehmen kann, müssen durch die Klasse des Objektes zwei Bedingungen erfüllt sein. Die erste Bedingung ist die Definition einer Überwachungsschnittstelle. Diese Schnittstelle definiert Ereignisprozeduren für alle Geschehnisse, die am *Observable* verfolgbar sind. Die Namensgebung der Überwachungsschnittstelle folgt der Konvention: „Name der überwachten Klasse“ + „Observer“. Eine Klasse, die von einer überwachten Klasse erbt, sollte auch eine abgeleitete Überwachungsschnittstelle definieren.

Die Ereignisprozeduren teilen sich in zwei Gruppen. Die erste Gruppe beschreibt allgemeine Ereignisse und folgt der Namenskonvention: „on“ + „Ereignis“. In der zweiten Gruppe werden alle Ereignisprozeduren zusammengefasst, die Attributänderungen übermitteln. Die Namensgebung dieser Prozeduren richtet sich nach dem Schema: „onChange“ + „Attributname“.

Alle Ereignisprozeduren haben als ersten Parameter einen Zeiger auf das überwachte Objekt. „onChange“ Ereignisse übergeben zusätzlich den alten und den neuen Wert des veränderten Attributes, während allgemeine Ereignisprozeduren beliebige Parameter enthalten können.

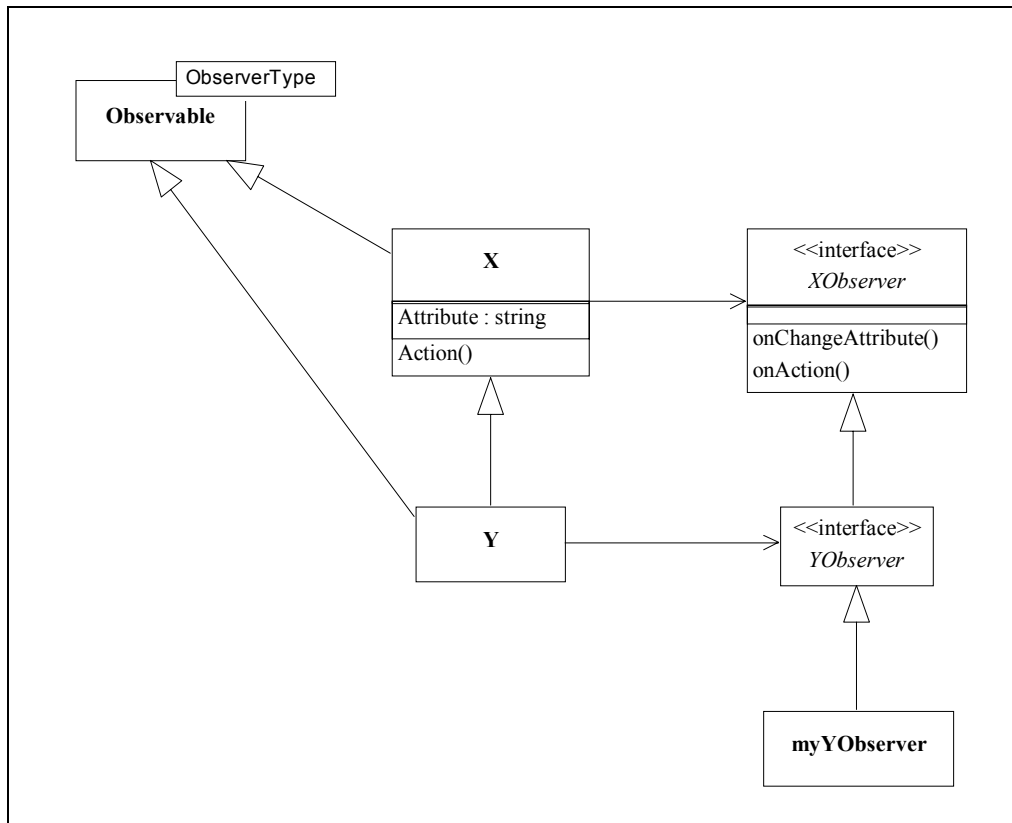


Abbildung 5

Im folgenden Beispiel wird eine Überwachungsschnittstelle für eine gedachte Klasse *Process* definiert. Diese Klasse erlaubt die Überwachung von drei Ereignissen: einfache Aktivierung des Prozesses, Aktivierung durch einen anderen Prozess und Änderung der Priorität. Ihre Überwachungsschnittstelle *ProcessObserver* bietet entsprechend die Ereignisprozeduren *onActivate*, *onActivateBy* und *onChangePriority*.

```

class ProcessObserver {
public:
    // Aktivierung
    virtual void onActivate(Process* sender) = 0;

    // Aktivierung durch einen anderen Prozess
    virtual void onActivateBy(Process* sender, Process* by) = 0;

    // Änderung der Priorität
    virtual void onChangePriority(Process* sender,
        Priority oldPriority, Priority newPriority) = 0;
};
    
```

Die zweite Bedingung, welche die Klasse eines Objekts erfüllen muss damit dieses als *Observable* agieren kann, ist die, dass sie als Basisklasse die Klassenvorlage *Observable<...>* besitzt. Das Template *Observable<...>* dient als Indikator für die Unterstützung des *Observation*-Konzepts und bietet gleichzeitig Funktionen für die Verwaltung der Überwachungsobjekte (*Observer*).

Observable<...> benötigt als Templateparameter die zugehörige Überwachungsschnittstelle. In Bezug auf das obige Beispiel könnte die zu überwachende Klasse beispielsweise wie folgt definiert sein:

```
class Process : public Observable<ProcessObserver> {
    ...
public:
    // Aktivierung
    void activate();
    void activateBy(Process* by);

    // Attributänderung
    void setPriority(Priority n);

private:
    Priority p;
    ...
};
```

Process ist von Observable<...> mit dem Parameter ProcessObserver abgeleitet. Damit erbt es die in Observable<...> definierten Verwaltungsfunktionen. Die Funktionen activate und activateBy erzeugen die Ereignisse, die in der Überwachungsschnittstelle durch die Prozeduren onActivate und onActivateBy repräsentiert sind. Die Funktion setPriority ruft onChangePriority auf.

Die Klasse aus dem Beispiel unterstützt nicht die Überwachung von Objektkonstruktion und -destruktion. Im Allgemeinen kann es jedoch von Interesse sein, bereits die Erzeugung des überwachten Objektes zu verfolgen. Dafür ist es notwendig, dass ein *Observer* zu Beginn der Konstruktion des *Observable* registriert werden kann. Observable<...> bietet deshalb die Möglichkeit, über seinen Konstruktor ein Objekt vom Typ der Überwachungsschnittstelle anzumelden. Eine überwachbare Klasse kann ihrerseits dem Nutzer einen entsprechenden Konstruktor-Parameter anbieten und diesen an Observable<...> weiterleiten.

Das obige Beispiel mit der Klasse Process würde, entsprechend erweitert, wie folgt aussehen:

```
class ProcessObserver {
public:
    // Erzeugung
    virtual void onCreate(Process* sender) = 0;
    ...
};

class Process : public Observable<ProcessObserver> {
    ...
public:
    // Konstruktor
    Process(ProcessObserver* o=0);
    ...
};

Process::Process(ProcessObserver* o=0) :
    Observable<ProcessObserver>(o)
{
    ...
}
```

## 3.2 Objekt-Objekt-Kopplung

---

Die Rolle eines *Observer* kann jedes Objekt einnehmen, dessen Klasse die entsprechende Überwachungsschnittstelle implementiert. Eine Klasse kann dabei problemlos mehrere Überwachungsschnittstellen gleichzeitig implementieren. Da alle Ereignisprozeduren als ersten Parameter einen Zeiger auf ein Objekt vom Typ der überwachten Klasse haben, löst der C++-*Overloading*-Mechanismus Namensgleichheiten zwischen Ereignisprozeduren verschiedener Überwachungsschnittstellen auf. Dennoch ist jeder *Observer* immer an konkrete Überwachungsschnittstellen gebunden.

### 3.2.2 Realisierung

Wie im obigen Abschnitt beschrieben, wird die Verwaltung der *Observer*-Objekte durch das Template `Observable<...>` übernommen. Die Mitteilung der Ereignisse obliegt jedoch der überwachten Klasse. Ihre Funktionen müssen die Ereignisprozeduren der Überwachungsschnittstelle aller registrierten *Observer* an den entsprechenden Punkten aufrufen. Für überwachte Attribute bedeutet dies beispielsweise, dass die überwachte Klasse die entsprechenden Attribute kapseln muss, um alle Veränderungen zu verfolgen.

`Observable<...>` bietet für An- und Abmeldung von Überwachern die Funktionen `addObserver` und `removeObserver`:

```
void addObserver(ObserverType* o)
void removeObserver(ObserverType* o)
```

Diese Funktionen werden an die überwachte Klasse vererbt und sind somit direkt am überwachten Objekt aufrufbar. Der Zugriff auf die Überwacher eines Objektes kann durch die folgende Funktion geschehen:

```
const std::list<ObserverType*>& getObserver();
```

Um das Verteilen der Ereignisse zu erleichtern, werden Präprozessormakros für das *Observation*-Konzept definiert. Das Makro

```
_obsForEach(ObserverType, Event)
```

expandiert zu:

```
{\
    for(std::list<ObserverType*>::const_iterator\
        i=Observable<ObserverType>::getObserver().begin();\
        i!=Observable<ObserverType>::getObserver().end(); ++i)\
        (*i)->on##Event;\
}
```

Im folgenden Beispiel wird ein Ereignis „Activate“ an alle registrierten Überwacher weitergeleitet:

```
_obsForEach(ProcessObserver, Activate(this))
```

Das Makro

```
_obsAForEach(ObserverType, Attribute, oldValue, newValue)
```

dient zur Übermittlung von Attributänderungen und wird umgesetzt in:

```
{\
    for(std::list<ObserverType*>::const_iterator\
        i=Observable<ObserverType>::getObserver().begin();\
        i!=Observable<ObserverType>::getObserver().end(); ++i)\
        (*i)->onChange##Attribute(this, oldValue, newValue);\
}
```

`_obsAForEach()` wird entsprechend dem folgenden Beispiel verwendet:

```
_obsAForEach(ProcessObserver, Priority, 1.0, 1.5);
```

### 3.2.3 Bewertung der Objekt-Objekt-Kopplung

Zu Beginn der Beurteilung des Konzepts *Observation* soll das in 3.2.1 begonnene Beispiel im Ganzen und vervollständigt dargestellt werden. Die zu überwachenden Objekte sind Instanzen der gedachten Klasse `Process`. Zusammen mit dieser Klasse werden im Rahmen einer Überwachungsschnittstelle eine Anzahl von Ereignissen definiert, die Interessenten zugänglich zu machen sind. Die öffentliche Schnittstelle der Klasse `Process` wird gemeinsam mit der Deklaration von `ProcessObserver` zusammengefasst:

```
#include "Observable.h"

class ProcessObserver;

class Process : public Observable<ProcessObserver> {
public:
    // Konstruktor
    Process(ProcessObserver* o=0);

    // Aktivierung
    void activate();
    void activateBy(Process* by);

    // Attributänderung
    void setPriority(Priority n);

private:
    Priority p;
};

class ProcessObserver {
public:
    // Erzeugung
    virtual void onCreate(Process* sender) = 0;

    // Aktivierung
    virtual void onActivate(Process* sender) = 0;

    // Aktivierung durch einen anderen Prozess
    virtual void onActivateBy(Process* sender, Process* by) = 0;

    // Änderung der Priorität
    virtual void onChangePriority(Process* sender,
                                Priority oldPriority, Priority newPriority) = 0;
};
```

In der Implementation der Funktionen der Klasse `Process` wird von den in *Observation* definierten Makros Gebrauch gemacht, um die Ereignisse zu übermitteln. Zunächst muss jedoch im Konstruktor die Initialisierung des Templates

### 3.2 Objekt-Objekt-Kopplung

---

Observable<...> mit dem optionalen, ersten Überwacher erfolgen<sup>1</sup>. Die Implementation stellt sich demnach wie folgt dar:

```
Process::Process(ProcessObserver* o=0) :
    Observable<ProcessObserver>(o), p(0)
{
    _obsForEach(ProcessObserver, Create(this));
}

void Process::activate() {
    _obsForEach(ProcessObserver, Activate(this));
}

void Process::activateBy(Process* by) {
    _obsForEach(ProcessObserver, ActivateBy(this, by))
}

void Process::setPriority(Priority n) {
    _obsAForEach(ProcessObserver, Priority, p, n);
    p=n;
}
```

Nachdem Instanzen der Klasse Process ihre Überwachung unterstützen, lässt sich eine Klasse von Überwachern definieren. Die Klasse myProcessObserver verfolgt alle angebotenen Ereignisse und gibt jeweils eine Meldung an cout aus:

```
#include <iostream>
using namespace std;

class myProcessObserver : public ProcessObserver {
    // Erzeugung
    virtual void onCreate(Process* sender) {
        cout << "onCreate(" << sender << ")" << endl;
    };

    // Aktivierung
    virtual void onActivate(Process* sender) {
        cout << "onActivate(" << sender << ")" << endl;
    }

    // Aktivierung durch einen anderen Prozess
    virtual void onActivateBy(Process* sender, Process* by) {
        cout << "onActivateBy(" << sender << ", " << by << ")"
        << endl;
    }

    // Änderung der Priorität
    virtual void onChangePriority(Process* sender,
        Priority oldPriority, Priority newPriority) {
        cout << "onChangePriority(" << sender << ", ";
        cout <<oldPriority<< ", "<< newPriority << ")" << endl;
    }
};
```

Abschließend sind die einzelnen Bausteine noch im Zusammenhang darzustellen. In der main-Funktion werden zwei Überwacher und ein Objekt der Klasse Process erzeugt und aneinander gekoppelt:

---

<sup>1</sup> Nur ein bei der Konstruktion angegebener *Observer* kann das onCreate-Ereignis erhalten.

```
int main(int argc, char* argv[]) {
    myProcessObserver one, two;
    Process p(&one);

    p.addObserver(two);

    p.activate();
    p.activateBy(0);

    p.removeObserver(two);

    p.setPriority(3);
}
```

Führt man das Programm aus, erhält man eine Ausgabe nach dem folgenden Muster, wobei das **X** für eine Speicheradresse steht:

```
onCreate(X)
onActivate(X)
onActivate(X)
onActivateBy(X, 0)
onActivateBy(X, 0)
onSetPriority(X, 0, 3)
```

Die Meldungen `onCreate` und `onSetPriority` werden von `one` ausgegeben. Die andern beiden Meldungen `onActivate` und `onActivateBy` werden sowohl von `one` als auch von `two` erzeugt und erscheinen daher doppelt.

*Observation* erlaubt es wie im Beispiel, Einblicke in die Abläufe innerhalb der Bibliothek zu nehmen. Der Wert, den das Konzept *Observation* für einen Anwender hat, hängt von der konsequenten Unterstützung innerhalb der Bibliothek ab. Fast alle Klassen von **ODEMx** unterstützen deshalb *Observation*. Dabei kann man die Überwachung sowohl auf einzelne Objekte als auch auf bestimmte Zeitabschnitte beschränken. Diese Eigenschaft lässt sich für den Hauptanwendungsbereich des Konzepts (3.3.3.3 Anwendung von *Observation*) vorteilhaft ausnutzen. Aber es wird darüber hinaus auch die Möglichkeit gegeben, *Observation* in eigenen Klassen zu unterstützen.

Die durchgehende Unterstützung von *Observation* innerhalb von **ODEMx** hat jedoch auch zwei Nachteile. Der erste Nachteil besteht im zusätzlichen Aufwand, der vom Entwickler für die Unterstützung von *Observation* betrieben werden muss. Für jede Klasse ist mindestens eine zusätzliche abstrakte Klasse (bzw. Schnittstelle) zu definieren. In den entsprechenden Funktionen sind die Makros `_obs...` einzufügen. Die zu überwachenden Attribute müssen mit entsprechenden Zugriffsfunktionen gekapselt werden. Der zweite Nachteil liegt in den zusätzlichen Aktionen, die durch die `_obs...` Makros in die Funktionen eingeführt werden. Selbst wenn kein *Observer* angemeldet ist, wird die Bedingung der *For*-Schleifen überprüft (siehe 3.2.2). Dieser „Overhead“ ist um so bedauerlicher, je einfacher die Funktionen sind (`setPriority`).

Eine Verbesserung des Konzepts wäre durch die Verwendung von Templates denkbar. Templates könnten zum Zeitpunkt der Übersetzung die Unterstützung von *Observation* durch einen Parameter aktivieren. Da einem Anwender der Bibliothek bekannt ist, ob er die Überwachung einer bestimmten Klasse von Objekten benötigt, kann er so Optimierungen vornehmen. Dieser Ansatz ist jedoch nur für „reine“ Template-Bibliotheken anwendbar.

#### 3.3 Erfassung und Auswertung der Daten

Aus Erfahrungen mit **ODEM** und aus dem veränderten Design von **ODEMx** ergab sich die Notwendigkeit, die Erfassung und Auswertung von Simulationsdaten zu verbessern. In diesem Abschnitt wird, nach Betrachtung der Ziele einer Simulation, ein Katalog von Anforderungen an die Unterstützung der Datenerfassung aufgestellt. Daraufhin werden drei Konzepte erläutert, die diese Anforderungen erfüllen sollen.

##### 3.3.1 Zielsetzungen einer Simulation

Die Simulation von Systemen kann mit verschiedenen Zielsetzungen erfolgen. Eine typische Klasse von Zielsetzungen der zeitdiskreten Computersimulation ist die Beantwortung von fest vorgegebenen Fragen über bestimmte Zustände eines Systems. Diese Fragen können sowohl quantitativer („Wie lang wird eine Warteschlange?“) als auch qualitativer („Erfüllt das System gestellte Anforderungen?“) Art sein. Die Erfüllung einer solchen Aufgabenstellung ist meist durch die Überwachung weniger Zustandsvariablen möglich. Das Verhalten des Modells während einer Simulation bleibt dann aber weitestgehend im Dunkeln. Deshalb ist es zumindest für die Überprüfung der Simulation notwendig, größere Bereiche eines Modells zu betrachten.

Aus der Beantwortung vorgegebener Fragen entwickeln sich oftmals weitere Fragestellungen. Deren Beantwortung ist nicht immer möglich, da die Modellierung ein Prozess ist, bei dem das Vorbild zielgerichtet vereinfacht wird. Ein erstelltes Modell kann daher grundsätzlich nicht für alle erdenklichen Fragen, die mit seinem Vorbild im Zusammenhang stehen, vorbereitet sein. Das für die objektorientierte Prozesssimulation meist angewandte Verfahren der strukturgetreuen Modellierung bietet aber typischerweise viel mehr Ansatzpunkte für die Informationsgewinnung, als für die ursprünglichen Fragen notwendig wären. Darüber hinaus sind objektorientierte Modelle leicht in Breite (durch die Modellierung weiterer Teile des Vorbildes) und Tiefe (durch detaillierte Modellierung vorhandener Objekte) für die Untersuchung neuer Probleme erweiterbar.

Eine zweite typische Klasse von Zielsetzungen für die Simulation besteht in der Untersuchung bestimmter Abläufe innerhalb eines Systems. Hier sind keine zusammenfassenden Aussagen über die Zustände des Modells gefragt. Stattdessen interessieren die Veränderungen von Teilen oder Strukturen des Modells. Dabei müssen die beteiligten Zustandsgrößen innerhalb der betrachteten Phasen überwacht und an eine Auswertungskomponente übermittelt werden. (Im Zusammenhang mit dieser Zielsetzung lohnt es sich, eine Simulation als Quelle von Daten und Ereignissen zu betrachten.) Die dadurch entstehenden Datenmengen sind im Allgemeinen um ein Vielfaches größer als bei der als erstes beschriebenen Klasse von Zielsetzungen. Auch hier gilt (analog zur ersten Klasse von Zielsetzungen einer Simulation), dass sich die Auswahl der zu untersuchenden Abläufe ändert, und für die Überprüfung des Modells oft zusätzliche Abläufe betrachtet werden müssen.

Eine Erweiterung der als zweites beschriebenen Klasse von Zielsetzungen ist die interaktive Simulation. Dabei wird eine Simulation mit modellexternen Komponenten gekoppelt, die ihrerseits während des Simulationslaufs Einfluss auf Zustandsgrößen der Simulation nehmen können.



### 3.3.2 Anforderungen

Aus den verschiedenen Zielsetzungen einer Simulation ergibt sich eine Reihe von Anforderungen an eine softwaretechnische Unterstützung der Erfassung und Auswertung anfallender Daten.

Eine erste Anforderung ist die Unterstützung für **zusammenfassende Aussagen** über eine Simulation. Zusammenfassende Aussagen werden im Allgemeinen in Form eines Berichtes über eine Simulation bereitgestellt. Dieser Bericht kann sowohl manuell als auch automatisch erzeugt werden. Für eine manuelle Berichterzeugung ist es notwendig, dass alle relevanten Zustandsgrößen zugänglich sind. Eine automatisierte oder teilautomatisierte Berichterstellung kann darin bestehen, dass umfangreiche Statistiken über alle wichtigen Zustandsgrößen geführt und ausgegeben werden.

Als zweites sollte, entsprechend den möglichen Zielsetzungen, Unterstützung für eine **Ablaufverfolgung** bestimmter Zustandsgrößen geboten werden. Die entsprechende Überwachung kann wiederum manuell oder automatisiert erfolgen. Für eine manuelle Überwachung müssen die Zustandsgrößen zusammen mit ihren Veränderungen für den Nutzer zugänglich sein. Eine automatisierte Ablaufverfolgung muss alle Ereignisse, die während einer Simulation auftreten, protokollieren.

Die Datenmengen, die durch Ablaufverfolgung und Simulationsbericht entstehen, werden jedoch schnell sehr groß und unübersichtlich. Eine weitere Anforderung an ein System für Simulationsdatenerfassung ist es daher, Konzepte für die **Reduktion** der entstehenden Datenmengen bereitzustellen.

Eine Variante der Verringerung entstehender Datenmengen ist die Möglichkeit der **Auswahl interessanter Daten**. Die Verarbeitungskette sollte möglichst frühzeitig unwesentliche Daten herausfiltern können.

Demgegenüber steht die Forderung nach **Vollständigkeit**. Es müssen alle anfallenden Informationen über einen Simulationslauf zugänglich sein, denn erst dadurch wird eine komplette Rekonstruktion der Geschehnisse innerhalb eines Modells möglich. Zudem ist nicht voraussagbar, welche Daten in der Zukunft tatsächlich benötigt werden.

Daher ist auch die **Steuerbarkeit** der Datenauswahl notwendig. Diese Steuerung kann durch den Entwickler der Simulation erfolgen. Eine Steuerung durch Parameter des fertigen Simulationsprogramms ist aber wünschenswert und sollte unterstützt werden.

Im Zusammenhang mit der Datenauswahl steht auch die Unterstützung verschiedener **Abstraktionsebenen**. Ein Modell hat meist verschiedene Abstraktionsebenen. Für den Fall der objektorientierten Prozesssimulation in C++ lassen sich beispielsweise eine C++-Ebene, eine Simulationsebene und mehrere Modellebenen unterscheiden. Auf der C++-Ebene betrachtet, besteht eine Simulation aus C++-Programmtext, der, übersetzt in ein ausführbares Programm, Textdaten erzeugt. In der Simulationsebene finden sich Elemente wie Prozesse und Synchronisationsobjekte. Die erzeugten Daten könnten, bezogen auf diese Ebene, zum Beispiel als Statistiken über die Verwendung einer Ressource verstanden werden. Auf den Modellebenen findet man schließlich eine stufenweise Annäherung der verwendeten Konzepte an ihre Vorbilder. Die Simulationsdaten werden auf diesen Ebenen zu Aussagen über das ursprüngliche System.

Die während einer Simulation erzeugten Daten werden erst im Bezug auf die verschiedenen Ebenen zu verständlichen Informationen. Ein Auftraggeber wird oftmals Antworten in der Sprache des Vorbilds oder einer hohen Modellebene verlangen. Gleichzeitig sind während der Entwicklung der Simulation auch die tieferen Ebenen zu betrachten.

Neben den verschiedenen Abstraktionsebenen kann der Wunsch nach verschiedenen **Ausgabeformaten** entstehen. Die Erfassung und Auswertung der Simulationsdaten sollte daher auch Unterstützung für verschiedene Ausgabekomponenten bieten. Dies erfordert die Unterstützung für eine **Austauschbarkeit** von Ausgabe- und Auswertungskomponenten.

Eine mögliche und typische Lösung für viele dieser Anforderungen ist die **Trennung von Modell und Auswertung**. Sie wird jedoch nur durch die Erfüllung der folgenden beiden Bedingungen ermöglicht: Erstens müssen für die Simulationsobjekte definierte Schnittstellen zu ihren Zustandsgrößen gegeben sein. Zweitens muss eine Zuordnung zwischen Simulationsobjekten und Auswertungskomponenten ermöglicht werden.

Abschließend muss auch die **Effizienz** der Datenverarbeitung und -auswertung beachtet werden. Einige der genannten Anforderungen, z.B. Datenauswahl und Zusammenfassungen, wirken sich bereits positiv auf die Effizienz aus, während andere, wie die Trennung von Modell und Auswertung, zwangsläufig Mehraufwand erzeugen.

#### 3.3.3 Lösungen

Für die Erfüllung der beschriebenen Anforderungen werden drei unterschiedliche Konzepte vorgeschlagen. Das erste Konzept –*Trace*– konzentriert sich auf die Ablaufverfolgung. Das zweite Konzept –*Report*– behandelt die Erstellung von Berichten. Das dritte Verfahren –*Observation*– (siehe 3.2 Objekt-Objekt-Kopplung) ermöglicht die Kopplung von Komponenten und wird für eine selektive Datenerfassung verwendet. Während die Umsetzungen von *Trace* und *Report* Weiterentwicklungen ihrer Vorbilder aus **ODEM** darstellen, ist die Realisierung von *Observation* durch **SimBeans** inspiriert.

##### 3.3.3.1 Trace

Das Verfahren „*Trace*“ beinhaltet die Protokollierung aller Ereignisse einer Simulation. Die Ereignisse betreffen die Zustandsänderungen, die während eines Simulationslaufes auftreten. Anhand einer vollständigen Folge von Ereignissen lassen sich alle Geschehnisse im Modell nachvollziehen. Die Umsetzung von *Trace* in **ODEMx** unterstützt den Programmierer mit einer allgemeinen Methode, um Ereignisse weiterzuleiten und zu verarbeiten.

In diesem Abschnitt wird zunächst ein Überblick über das *Trace*-Konzept gegeben. Danach werden die einzelnen Teilkonzepte erläutert. Nach einer Beschreibung der Entwicklungsgeschichte des Konzepts wird abschließend eine Bewertung in Bezug auf den Anforderungskatalog vorgenommen.

## Überblick

Im Rahmen des *Trace*-Konzepts kann ein Objekt zwei Rollen einnehmen. Die erste Rolle ist die eines Ereignisproduzenten (*TraceProducer*). Ereignisproduzenten generieren Folgen von Ereignissen, mit denen sie ihre Zustandsänderungen protokollieren. Die zweite Rolle ist die eines Ereigniskonsumenten (*TraceConsumer*). Ein Ereigniskonsument empfängt alle Ereignisse, die während einer Simulation auftreten und verarbeitet diese weiter. Zwischen den beiden Objektarten steht ein zentrales *Trace*-Objekt. Dieses Objekt empfängt Ereignisse von den Ereignisproduzenten und leitet sie an registrierte Ereigniskonsumenten weiter. Es bietet zusätzlich Methoden an, mit denen der Ereignisfluss zentral gesteuert werden kann.

In Abbildung 6 sind mit *Condq*, *Process* und *Simulation* Klassen von Ereignisproduzenten dargestellt. Sie implementieren die Schnittstelle *TraceProducer*, die eine navigierbare Beziehung zum *Trace*-Objekt garantiert. *XmlTrace* ist eine Klasse von Ereigniskonsumenten. Sie bietet *Trace* die Schnittstelle *TraceConsumer*, mit deren Hilfe *Trace* Ereignisse weiterleiten kann.

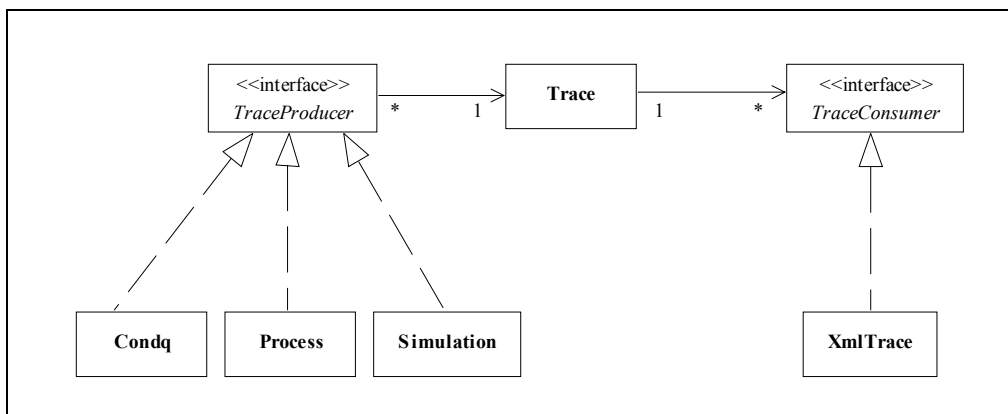


Abbildung 6

Ein Ereignis ist durch einen Ereignistyp spezifiziert. Es kann zusätzliche Daten beinhalten, die in einer vorgegebenen oder selbst definierten Struktur organisiert werden können. Die Übermittlung der Ereignisse erfolgt durch Prozedurrufe. In einfachen Fällen wird ein Ereignis durch genau einen Prozedurruf an das *Trace*-Objekt übermittelt. *Trace* ruft daraufhin seinerseits eine entsprechende Prozedur bei allen registrierten Ereigniskonsumenten auf (siehe Abbildung 7). Komplexere Ereignisse werden durch eine abgeschlossene Folge von Prozedurrufen übermittelt (siehe Zusammengesetzte Ereignisse).

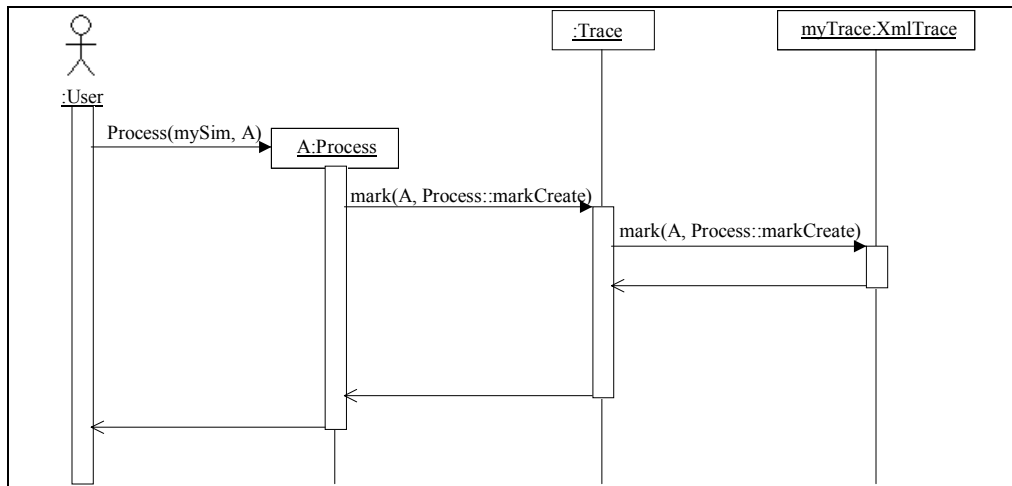


Abbildung 7

#### TraceProducer

TraceProducer ist die abstrakte Basisklasse für alle Ereignisproduzenten. Sie stellt sicher, dass Ereignisproduzenten einen Namen (LabeledObject) besitzen, die Abfrage eines C++-Typs unterstützen (TypedObject) und Zugriff auf das zentrale Trace-Objekt (getTrace) haben.

```

class TraceProducer :
    public virtual LabeledObject, public virtual TypedObject {
    public:
        virtual Trace* getTrace() const = 0;
};
    
```

#### Ereignistyp

Alle Prozeduren für die Übermittlung von Ereignissen an Trace erwarten mindestens die beiden Parameter „Sender des Ereignisses“ und „Ereignistyp“. Der Sender eines Ereignisses ist immer ein Objekt, das die Schnittstelle TraceProducer unterstützt. Der Ereignistyp wird verwendet, um das zu übermittelnde Ereignis zu spezifizieren (siehe Einfache Ereignisse).

Ein Ereignistyp wird durch eine Instanz der Klasse MarkType definiert. MarkType enthält die drei Attribute name, id und scope, die durch Konstruktoren initialisiert werden.

```

typedef unsigned long MarkTypeId;

class MarkType {
    const char* name;
    MarkTypeId id;
    const std::type_info& scope;
public:
    MarkType(const char* n, const std::type_info& s) :
        name(n), id(NOID), scope(s) {assert(name!=NONAME);}

    MarkType(MarkTypeId i, const std::type_info& s) :
        name(NONAME), id(i), scope(s) {assert(id!=NOID);}

    MarkType(const char* n, MarkTypeId i,
    
```

```
        const std::type_info& s) : name(n), id(i), scope(s)
    {assert(name!=NONAME || id!=NOID);}

    const char* getName() const {return name;}
    MarkTypeId getId() const {return id;}
    const std::type_info& getScope() const {return scope;}

    static const MarkTypeId NOID;
    static const char* NONAME;
};
```

Ereignistypen werden immer in Bezug auf einen Geltungsbereich definiert. Dieser wird durch das Attribut `scope` in Form einer konstanten Referenz auf die *RTTI* eines C++-Typs angegeben. Ereignisse, die von einem Prozess erzeugt werden, könnten beispielsweise den Geltungsbereich `typeid(Process)` haben. Bei allen Klassen aus **ODEMx**, die Trace unterstützen, wird nach Konvention `scope` auf `typeid(Klassenname)` gesetzt.

Die Attribute `name` und `id` identifizieren das Ereignis innerhalb ihres Geltungsbereiches. Es genügt bereits, eines der beiden Attribute zu setzen. Eine Angabe der `id` kann jedoch die Weiterverarbeitung erleichtern, indem es Textvergleiche erspart, während der Wert des Attributes `name` durch texterzeugende Ereigniskonsumenten direkt ausgegeben werden kann. Für beide Attribute sind innerhalb des Geltungsbereiches eindeutige Werte auszuwählen. Ein Ereignistyp wird beispielsweise durch die folgende Zeile definiert:

```
const MarkType Process::markCreate("create", 1, typeid(Process));
```

Ereignistypen unterstützen Ereigniskonsumenten bei der Verarbeitung der Ereignisse. Ein Benutzer des *Trace*-Konzepts kann eigene Ereignistypen definieren, die von angepassten Ereigniskonsumenten verarbeitet werden. Allgemeine Ereigniskonsumenten können dennoch die Informationen aus dem Ereignistyp auswerten und für ihre Ausgabe verwenden.

### Einfache Ereignisse

Die Erzeugung einfacher Ereignisse erfolgt durch den Aufruf bestimmter Prozeduren an dem zentralen Trace-Objekt:

```
void mark(const TraceProducer* sender, MarkType m,
         const char* comment = 0)
```

Diese Prozedur zeigt das einfachste Ereignis des *Trace*-Konzepts. Neben den in allen Ereignisprozeduren erforderlichen Parametern Sender (`sender`) und Ereignistyp (`m`) wird lediglich ein optionaler Kommentar (`comment`) erwartet. Kommentare werden für alle einfachen Ereignisprozeduren unterstützt und können beispielsweise verwendet werden, um in Textmitschriften zusätzliche Hinweise zu geben. Der Kommentar-Parameter ist aber nicht für die Weitergabe wesentlicher Informationen gedacht. Für die Übergabe von Daten in Ereignissen werden weitere Prozeduren bereitgestellt.

Das folgende Beispiel zeigt die Erzeugung eines Ereignisses mit Hilfe der `mark`-Prozedur. Das Ereignis soll die Erzeugung eines Prozesses mitteilen. Die Klasse `Process` implementiert dafür `TraceProducer` und definiert den Ereignistyp `markCreate` als Klassenkonstante.

```
class Process : public TraceProducer, ... {  
...  
public:  
    Process(...);  
  
    // Deklaration des Ereignisstyps  
    static const MarkType markCreate;  
...  
};  
...  
Process::Process(...) : ... {  
    ...  
    // Ereigniss wird erzeugt:  
    getTrace()->mark(this, markCreate);  
    ...  
}  
...  
// Definition des Ereignistyps markCreate  
const MarkType Process::markCreate("create", 1, typeid(Process));  
...
```

Für viele Ereignisse wird es wünschenswert sein, zusätzliche Informationen bereitzustellen. Für diesen Zweck existieren Ereignisprozeduren, die weitere Parameter erwarten. Die nächsten beiden Prozeduren erlauben es, Ereignisse mit Verweisen auf weitere TraceProducer zu erzeugen.

```
void mark(const TraceProducer* sender, MarkType m,  
         const TraceProducer* partner, const char* comment = 0)  
  
void mark(const TraceProducer* sender, MarkType m,  
         const TraceProducer* firstPartner,  
         const TraceProducer* secondPartner,  
         const char* comment = 0)
```

Neben den Standardparametern werden die Parameter partner, firstPartner und secondPartner angeboten. So kann beispielsweise ein Ereignis für die Aktivierung eines Prozesses durch einen anderen erzeugt werden:

```
class Process : public TraceProducer, ... {  
...  
public:  
    void activate(Process* by);  
  
    // Deklaration des Ereignisstyps  
    static const MarkType markActivate;  
...  
};  
...  
void Process::activate(Process* by) {  
    ...  
  
    // Ereigniss wird erzeugt:  
    getTrace()->mark(this, markActivate, by);  
  
    ...  
}  
...  
// Definition des Ereignistyps markCreate  
const MarkType Process::markActivate("activate",  
                                     11, typeid(Process));  
...
```

Die folgenden Prozeduren können verwendet werden, um Attributänderungen zu protokollieren. Sie erwarten jeweils den neuen und den alten Wert des Attributes als Parameter. Es wurden Prozeduren für die verschiedenen elementaren C++-Datentypen definiert:

```
void mark(const TraceProducer* sender, MarkType m, bool newValue,
          bool oldValue, const char* comment = 0)

void mark(const TraceProducer* sender, MarkType m, char newValue,
          char oldValue, const char* comment = 0)

void mark(const TraceProducer* sender, MarkType m, short newValue,
          short oldValue, const char* comment = 0)

void mark(const TraceProducer* sender, MarkType m, int newValue,
          int oldValue, const char* comment = 0)

void mark(const TraceProducer* sender, MarkType m, long newValue,
          long oldValue, const char* comment = 0)

void mark(const TraceProducer* sender, MarkType m, float
          newValue, float oldValue, const char* comment = 0)

void mark(const TraceProducer* sender, MarkType m, double
          newValue, double oldValue, const char* comment = 0)

void mark(const TraceProducer* sender, MarkType m, const char*
          newValue, const char* oldValue, const char* comment=0)

void mark(const TraceProducer* sender, MarkType m, unsigned char
          newValue, unsigned char oldValue, const char* comment=0)

void mark(const TraceProducer* sender, MarkType m, unsigned short
          newValue, unsigned short oldValue, const char* comment=0)

void mark(const TraceProducer* sender, MarkType m, unsigned int
          newValue, unsigned int oldValue, const char* comment=0)

void mark(const TraceProducer* sender, MarkType m, unsigned long
          newValue, unsigned long oldValue, const char* comment=0)
```

Wenn die angebotenen Prozeduren nicht ausreichen, um die für ein Ereignis erforderlichen Daten zu übermitteln, können die zusammengesetzten Ereignisse verwendet werden. Die Erzeugung dieser Ereignisse wird im folgenden Abschnitt erläutert.

Für die Erzeugung von Ereignissen eines Ereignistyps sollten immer die gleichen Ereignisprozeduren verwendet werden. Das erleichtert die Weiterverarbeitung der Ereignisse und beugt Konflikten mit zusammengesetzten Ereignissen vor (siehe Zusammengesetzte Ereignisse):

```
class Process : public TraceProducer, ... {
...
public:
    Process(...);
    Process(Process* by, ...);

    // Deklaration des Ereignistyps
    static const MarkType markCreate;
    static const MarkType markCreateBy;
...
}
```

```
};
...
Process::Process(...) : ... {
    ...

    // markCreate Ereigniss wird mit minimalen Parametern erzeugt
    getTrace()->mark(this, markCreate);

    ...
}
...
Process::Process(Process* by, ...) : ... {
    ...

    // Schlecht:
    // markCreate wird mit veränderten Parametern erzeugt
    getTrace()->mark(this, markCreate, by);

    // Besser:
    // es wird der zweite Ereignistyp verwendet
    getTrace()->mark(this, markCreateBy, by);

    ...
}
...
// Definition des Ereignistyps markCreate und markCreateBy
const MarkType Process::markCreate("create", 1, typeid(Process));
const MarkType Process::markCreateBy(
    "createBy", 2, typeid(Process));
...
```

#### Zusammengesetzte Ereignisse

Zusammengesetzte Ereignisse können verwendet werden, um Ereignisse mit komplexeren Parametern zu erzeugen. Dafür wird eine Baumstruktur aus gekennzeichneten Datenfeldern genutzt, die für konkrete Ereignissen mit individuellen Daten gefüllt werden. Die typische Übermittlung eines zusammengesetzten Ereignisses könnte wie folgt ablaufen:

```
class A : public TraceProducer, ... {
public:
    void komplexeInteraktion(Process* a, Process* b, int i);

    // Deklaration des Ereignistyps
    static const MarkType markAction;

    // Deklaration der Datenfeld
    static const Tag tagActionA;
    static const Tag tagActionB;
    static const Tag tagActionI;
};
void A::komplexeInteraktion(Process* a, Process* b, int i) {
    ...
    // Erzeuge Ereignis:
    //-----
    // Eröffne ein zusammengesetztes Ereignis
    getTrace()->beginMark(this, markAction);

    // Erzeuge Baumstruktur
    getTrace()->addTag(tagActionA, a);
    getTrace()->addTag(tagActionB, b);
    getTrace()->addTag(tagActionI, i);
}
```



```
// Schliesse zusammengesetztes Ereignis  
getTrace()->endMark();  
...  
}  
...  
// Definition des Ereignistyps markInteraktion  
const MarkType Process::markAction("interact", 1, typeid(A));  
  
// Definition der Tags  
const Tag Process::tagActionA(1);  
const Tag Process::tagActionB(2);  
const Tag Process::tagActionI("I");  
...
```

Die Übermittlung eines zusammengesetzten Ereignisses beginnt mit der Prozedur `beginMark`. Sie erwartet wieder die Parameter `Sender` und `Typ` des Ereignisses.

```
void beginMark(const TraceProducer* sender, MarkType m)
```

Mit den `addTag`-Prozeduren können danach die Daten des Ereignisses übergeben werden. Das Ende des Ereignisses wird durch `endMark` angezeigt:

```
void endMark()
```

Es existieren `addTag`-Prozeduren für die Übermittlung der grundlegenden C++-Datentypen und der Verweise auf `TraceProducer`:

```
void addTag(Tag t, bool value)  
void addTag(Tag t, char value)  
void addTag(Tag t, short value)  
void addTag(Tag t, int value)  
void addTag(Tag t, long value)  
void addTag(Tag t, float value)  
void addTag(Tag t, double value)  
void addTag(Tag t, const char* value)  
void addTag(Tag t, const TraceProducer* value)  
void addTag(Tag t, unsigned char value)  
void addTag(Tag t, unsigned short value)  
void addTag(Tag t, unsigned int value)  
void addTag(Tag t, unsigned long value)
```

Darüber hinaus ist es möglich, die Konstruktion der Datenstruktur weiter zu schachteln. Dafür kann mit der Prozedur `beginTag` ein Tag eröffnet werden, das mit `endTag` geschlossen wird:

```
void beginTag(Tag t)  
void endTag(Tag t)
```

Zwischen den Aufrufen dieser Prozeduren können alle `addTag`-Prozeduren und weitere `beginTag`-`endTag`-Paare gerufen werden. Zwei zusammenhängende `beginTag`- und `endTag`-Rufe müssen dabei immer gleichwertige Tag-Objekte verwenden.

Die Klasse `Tag` wird für die Kennzeichnung der Datenfelder verwendet. Sie kapselt entweder einen `const char*` oder einen `TagId`-Wert:

### 3.3 Erfassung und Auswertung der Daten

---

```
typedef unsigned long TagId;

class Tag {
    const char* name;
    TagId id;
public:
    Tag(const char* n): name(n), id(NOID) {assert(name!=NONAME);}
    Tag(TagId i) : name(NONAME), id(i) {assert(id!=NOID);}

    const char* getName() const {return name;}
    TagId getId() const {return id;}

    static const TagId NOID;
    static const char* NONAME;

    friend bool operator ==(const Tag& t1, const Tag& t2);
};
```

Die Konstruktoren der Klasse erlauben eine vereinfachte Nutzung der Klasse Tag:

```
...
// Erzeuge Ereignis
getTrace()->beginMark(this,
                    MarkType("simpleComplex",typeid(Beispiel)));

// einfaches gekennzeichnetes Datum hinzufügen
getTrace()->addTag("Erstens", this);

// schachteln
getTrace()->beginTag("Schachtel");
getTrace()->addTag(1, 'a');
getTrace()->addTag(2, 'b');

// Schachtelung und Ereignis beenden
getTrace()->endTag("Schachtel");
getTrace()->endMark();
```

Die im Beispiel erzeugte Struktur wird in Abbildung 8 dargestellt:

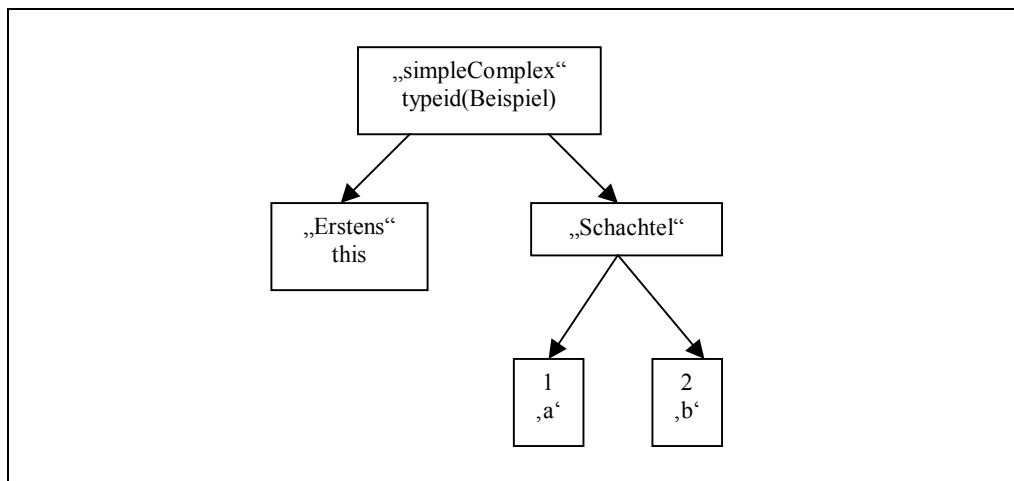


Abbildung 8

Die Erzeugung von zusammengesetzten Ereignissen eines Ereignistyps sollte immer mit der gleichen Folge von Prozedurrufen erfolgen. Dies erleichtert die Weiterverarbeitung der Ereignisse. Andernfalls müssten an bestimmte Ereignistypen

angepasste Ereigniskonsumenten mehrere Folgen von Prozedurrufen für einen Ereignistyp beherrschen und unterscheiden können.

Durch die Verbindung von Daten und Strukturinformationen können auch allgemeine Ereigniskonsumenten spezielle zusammengesetzte Ereignisse auswerten. Der Nachteil dieses Schemas ist der Mehraufwand, der bei der Übertragung anfällt. Die Alternative hierfür wäre die Verwendung spezieller Klassen für zusammengesetzte Ereignisse. Diese Klassen müssten jedoch eine Schnittstelle zur Ermittlung ihrer inneren Struktur besitzen (siehe 4.1 Introspection/ Inspection).

### **Steuerung des *Trace***

Die Ablaufverfolgung kann am zentralen *Trace* mit den folgenden Methoden gesteuert werden:

```
void startTrace()
void stopTrace()
void pauseTrace()
void continueTrace()
```

Die Ablaufverfolgung beginnt mit einem Aufruf von `startTrace()` und endet mit `stopTrace()` oder der Zerstörung des *Trace*-Objektes. Mit `pauseTrace()` und `continueTrace()` kann die Protokollierung zeitweise unterbrochen werden.

Ereignisse werden im Zeitraum zwischen `startTrace` und `endTrace` an die Ereigniskonsumenten verteilt. Die Übermittlung erfolgt auch während der Unterbrechungen. Dadurch können Ereigniskonsumenten ihren internen Zustand mit der Simulation konsistent halten. Während einer Pause sollten Ereigniskonsumenten jedoch keine Ausgaben erzeugen.

### **TraceConsumer**

Ereigniskonsumenten sind Implementationen der abstrakten Basisklasse *TraceConsumer*. *TraceConsumer* deklariert virtuelle Prozeduren, die den Prozeduren zur Übermittlung einfacher und zusammengesetzter Ereignisse und den Prozeduren zur Steuerung der Ablaufverfolgung am *Trace*-Objekt entsprechen. Ein Ereigniskonsument implementiert die virtuellen Methoden, um auf die entsprechenden Ereignisse zu reagieren.

Die Verwaltung der *TraceConsumer*-Objekte erfolgt durch *Trace*. Die folgenden Methoden kommen dabei zum Einsatz:

```
void addConsumer(TraceConsumer* c)
void removeConsumer(TraceConsumer* c)
```

Mit `addConsumer` wird ein neuer Ereigniskonsument registriert. Die Prozedur `removeConsumer` kann dazu verwendet werden, Ereigniskonsumenten wieder zu entfernen. Im folgenden Abschnitt wird eine spezielle Implementation der Schnittstelle *TraceConsumer* näher erläutert.

### **XmlTrace**

Der in **ODEMX** enthaltene *XmlTrace* ist ein Beispiel für einen Ereigniskonsumenten. *XmlTrace* verarbeitet beliebige Ereignisse und erzeugt XML-Text. Dafür werden die Informationen aus *MarkType*, *LabeledObject* und *TypedObject* ausge-

wertet. Zusammengesetzte Ereignisse werden in XML-Strukturen umgesetzt. Die Ausgabe erfolgt in eine durch `XmlTrace` angelegte Datei oder in einen Ausgabestrom.

```
class XmlTrace : public virtual TraceConsumer {
public:
    XmlTrace(Simulation* s, const char* fileName = 0);
    XmlTrace(Simulation* s, std::ostream& os);
    virtual ~XmlTrace();

    // filter
    void setFilter(const char* f);
    const std::string& getFilter() const;

private:
    // Simple trace marks
    virtual void mark(const TraceProducer* sender,
                    const MarkType* m,
                    const char* comment = 0);

    // Marks that include information about
    // producer associations:
    virtual void mark(const TraceProducer* sender,
                    const MarkType* m,
                    const TraceProducer* partner,
                    const char* comment = 0);
    virtual void mark(const TraceProducer* sender,
                    const MarkType* m,
                    const TraceProducer* firstPartner,
                    const TraceProducer* secondPartner,
                    const char* comment = 0);

    // Marks that include information about producer attributes:
    virtual void mark(const TraceProducer* sender,
                    const MarkType* m,
                    bool newValue,
                    bool oldValue,
                    const char* comment = 0);
    ...

    // Complex trace marks
    virtual void beginMark(const TraceProducer* sender, const
MarkType* m);
    virtual void endMark();

    // This methodes are called to describe
    // the construction of the complex mark.
    virtual void beginTag(Tag t);
    virtual void endTag(Tag t);

    virtual void addTag(Tag t, bool value);
    ...

    // Trace Control
    virtual void startTrace(); // XmlTrace initializes a new
trace
    virtual void stopTrace(); // XmlTrace cleans up
    virtual void pauseTrace(); // XmlTrace starts a break
    virtual void continueTrace(); // XmlTrace ends a break
    ...
};
```

XmlTrace bietet die Möglichkeit, einen einfachen Filter auf den Ereignisfluss anzuwenden. Dieser Filter nutzt Ereignistyp und Sender der Ereignisse, um die Verarbeitung ausgewählte Ereignisse zu unterdrücken oder zu erlauben.

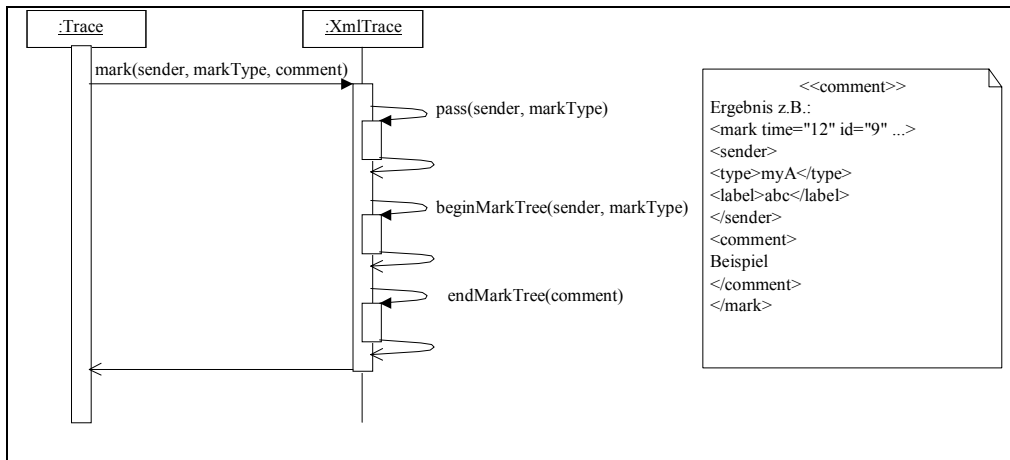


Abbildung 9

In Abbildung 9 wird beispielhaft die Verarbeitung des einfachsten Ereignisses durchgeführt. Die Handhabung erfolgt in der durch XmlTrace implementierten Methode:

```

void XmlTrace::mark(const TraceProducer* sender,
                   const MarkType* m, const char* comment/*= 0*/)
{
    // filter
    if (!pass(m, sender))
        return;

    beginMarkTree(sender, m);
    endMarkTree(comment);
}
  
```

In mark wird zunächst mit der Funktion pass der Filter auf das Ereignis angewendet. Wenn das Ereignis herausgefiltert werden soll, gibt die Funktion false zurück und setzt ein XmlTrace internes filterOut Flag. Dieses Flag wird in anderen Methoden für die Filterung zusammengesetzter Ereignisse verwendet.

Im nächsten Schritt wird mit der Funktion beginMarkTree das XML Tag mark eröffnet und mit den Parametern time, name, id und scope gefüllt. Die Werte für die letzten drei Parameter ermittelt beginMarkTree aus dem Ereignistyp, während der Parameter time auf die Simulationszeit gesetzt wird. Die Informationen, die durch die TraceProducer Schnittstelle zugänglich sind, werden im <sender>...</sender> Block ausgegeben.

Die letzte Funktion endMarkTree fügt, wenn ein Kommentar angegeben wurde, den comment-Block ein und schließt das mark-Tag.

### Entwicklungsgeschichte

Das Trace-Konzept geht auf Erfahrungen mit ODEM zurück. Die erste Version einer Ablaufverfolgung in ODEM bestand in der Erzeugung von Textzeilen an den Stellen im ODEM Programmtext, an denen interessante Ereignisse auftraten. Die

### 3.3 Erfassung und Auswertung der Daten

---

Steuerung erfolgte über eine globale Variable für eine Aktivierung oder Unterbrechung der Mitschrift. Die Ausgabe wurde in einen ebenfalls global zugänglichen Ausgabestrom geleitet.

In der Studienarbeit [Ger01] wurde versucht, aus den Informationen der Ablaufverfolgung automatisch eine 3D-Animation zu erzeugen. Dafür wurden die Erweiterungen zunächst analog zur bestehenden Implementation der Mitschriften an den Stellen, an denen Ereignisse auftreten, vorgenommen.

Durch den Umfang der Änderungen wurde die Übersichtlichkeit des Programmtextes beeinträchtigt. Die Lösung dafür bestand in der Verschiebung der Ereignisverarbeitung in entsprechende Ereignisprozeduren. Diese Funktionen entsprachen den bekannten Ereignissen.

Initiiert durch die Idee einer *XML*-Protokollierung der Ereignisse wurde die verallgemeinerte Struktur Ereigniserzeuger -> Ereignisverteiler -> Ereigniskonsument entwickelt und umgesetzt. Ein zentrales *Trace*-Objekt nimmt dabei alle Ereignisse über Funktionsaufrufe entgegen und leitet diese an alle Konsumenten weiter. Die Funktionen des *Trace*-Objektes spiegeln dabei weiterhin die bekannten Ereignisse wieder.

Während des Simulationsprojektes **Simring** mit der veränderten **ODEM**-Bibliothek zeigte sich, dass eine Unterstützung für nutzerspezifische Ereignisse vorteilhaft wäre. In dem Projekt wird eine Simulation mit einer externen Auswertungskomponente über **CORBA** gekoppelt. Die Simulation sendet dabei Ereignisse aus der Abstraktionsebene Modell an die Auswertungskomponente. Da der *Trace*-Mechanismus benutzerdefinierte Ereignisse nicht unterstützte, wurde ein paralleler Mechanismus umgesetzt.

In dem für **ODEMx** entwickelten *Trace*-Verfahren könnte man dieses Problem durch einen angepassten Ereigniskonsumenten lösen. Dieser würde die speziellen Ereignisse des Modells in **CORBA** Funktionsaufrufe umwandeln und an die Auswertungskomponente weiterleiten.

#### **Bewertung des *Trace*-Konzepts**

Das *Trace*-Konzept setzt die Anforderung nach einer Ablaufverfolgung um. Dabei wird eine Trennung von Modell und Auswertung realisiert. In deren Folge können verschiedene Ausgabeformate unterstützt, und untereinander ausgetauscht werden. Da auch angepasste *TraceConsumer* realisierbar sind, lassen sich verschiedene Abstraktionsebenen umsetzen.

*XmlTrace* bietet mit seinem Filter ein Beispiel, wie auf der Basis des *Trace*-Konzepts, Datenauswahl und Steuerbarkeit umgesetzt werden können, während das Konzept im Kern keine entsprechenden Vorkehrungen trifft. Das Hauptaugenmerk dieses Konzepts liegt jedoch auf der Vollständigkeit der erfassten Informationen. Dabei kommt es auch auf die Unterstützung durch die Modellbausteine an.

Die Trennung von Modell und Auswertung wird durch die Kommunikation mithilfe von Ereignissen erreicht. Dies hat den Nachteil, dass ein Mehraufwand entsteht. Dabei sind insbesondere die zusammengesetzten Ereignisse problematisch. Eine Lösung für dieses Problem wäre in der Einführung weiterer Prozeduren für „typische“ zusammengesetzte Ereignisse.

### 3.3.3.2 Report

Das zweite Verfahren für die Erfassung und Auswertung von Simulationsdaten ist die Erstellung von Berichten. Berichte fassen beispielsweise Informationen über die Benutzung von Ressourcen und Warteschlangen zusammen.

Zunächst wird ein Überblick das *Report*-Konzept gegeben. Danach werden die einzelnen Teile des Konzepts erläutert. Abschließend erfolgt eine Bewertung in Bezug auf den Anforderungskatalog.

#### Übersicht

ODEmx bietet das *Report*-Konzept als Unterstützung für die Erstellung von Berichten an. Dieses Konzept beinhaltet die folgenden Komponenten:

- Report
- ReportProducer
- TableDefinition
- Table

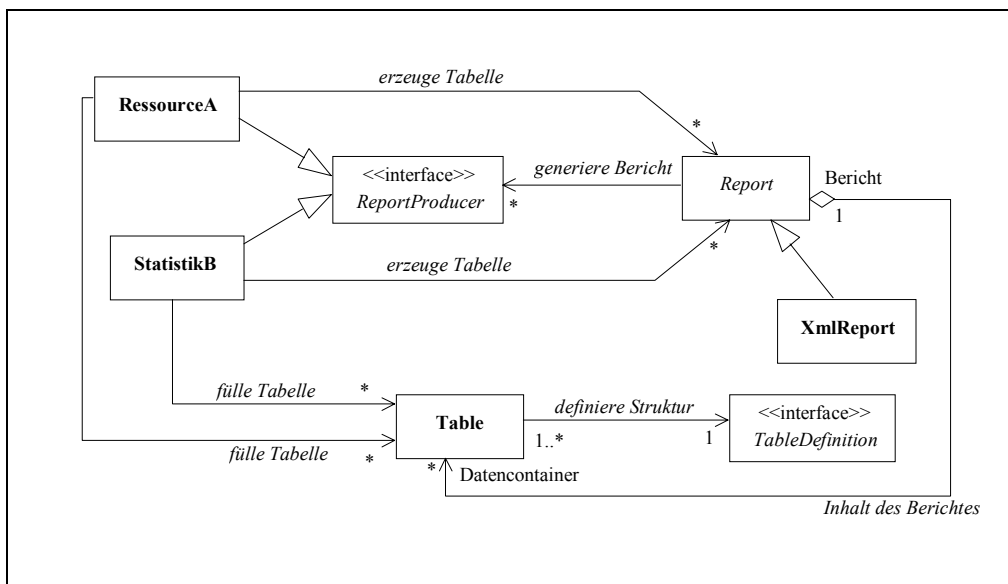


Abbildung 10

Objekte der Klasse *Report* repräsentieren jeweils einen Bericht. Innerhalb einer Simulation können mehrere *Report* Objekte parallel existieren. *Report* ist aufgrund einer rein virtuellen Funktion eine abstrakte Klasse. Durch die Definition dieser Funktion implementieren abgeleitete Klassen ihre individuelle Ausgabe der gesammelten Daten. In Abbildung 10 stellt *XmlReport* eine solche Ableitung dar.

Ein Bericht setzt sich aus Tabellen zusammen. Tabellen stehen dabei immer in einer „Enthaltensein“-Beziehung zu ihrem Bericht und werden durch Objekte der Klasse *Table* umgesetzt. Tabellen haben einen Namen und eine definierte Struktur. Diese Struktur ist im *Report*-Konzept durch die Anzahl ihrer Spalten sowie durch Art und Namen einer jeden Spalte gegeben. Bereitgestellt wird diese Strukturinformation über die Schnittstelle *TableDefinition*. In einem Bericht können meh-

### 3.3 Erfassung und Auswertung der Daten

rere Tabellen mit gleichem Namen vorkommen, solange sich ihre Strukturen unterscheiden. Ebenso sind mehrere Tabellen gleicher Struktur aber unterschiedlicher Namen möglich. Zu jedem Paar Tabellennamen-Tabellenstruktur kann jedoch höchstens eine Tabelle in einem Bericht enthalten sein.

Objekte, die an der Erzeugung eines Berichtes teilnehmen wollen, müssen die Schnittstelle `ReportProducer` unterstützen. `ReportProducer` garantiert die Definition einer Funktion, die durch `Report` während der Berichterzeugung aufgerufen wird. Innerhalb dieser Funktion erzeugen oder erfragen die beitragenden Objekte Tabellen des jeweiligen Berichtes, um diese mit ihren Daten zu füllen. In Abbildung 10 sind die `RessourceA` und `StatistikB` Beispiele für Klassen, die das *Report*-Konzept unterstützen. `ReportProducer` müssen bei den `Report`-Objekten registriert werden, zu denen sie einen Beitrag leisten sollen. Sie können jedoch parallel an mehreren Berichten „mitwirken“. Darüber hinaus haben die `ReportProducer` eines Berichtes die Möglichkeit, Tabellen gemeinsam zu füllen, indem sie sich auf einen gemeinsamen Namen und die gleiche Struktur einigen.

In Abbildung 11 wird die Erzeugung eines Berichtes anhand eines Beispiels schematisch dargestellt. Die Berichterzeugung wird zunächst durch den Nutzer gestartet. Daraufhin werden alle registrierte `ReportProducer` aufgefordert, ihre Daten zu übermitteln. Im Beispiel wird ein Objekt `r` der Klasse `RessourceA` aufgerufen. Der `ReportProducer` `r` verwendet `MyTableDef`, um die Struktur seiner Tabelle zu definieren. Im nächsten Schritt fordert `r` mithilfe von `td` eine Tabelle entsprechender Struktur an. Nachdem `r` diese Tabelle gefüllt hat, beendet er seine Aktionen. Der Bericht `B1` wird, nachdem alle `ReportProducer` aufgerufen wurden, die erzeugten Tabellen in der Reihenfolge ihrer Erzeugung verarbeiten und in entsprechende Ausgaben umsetzen.

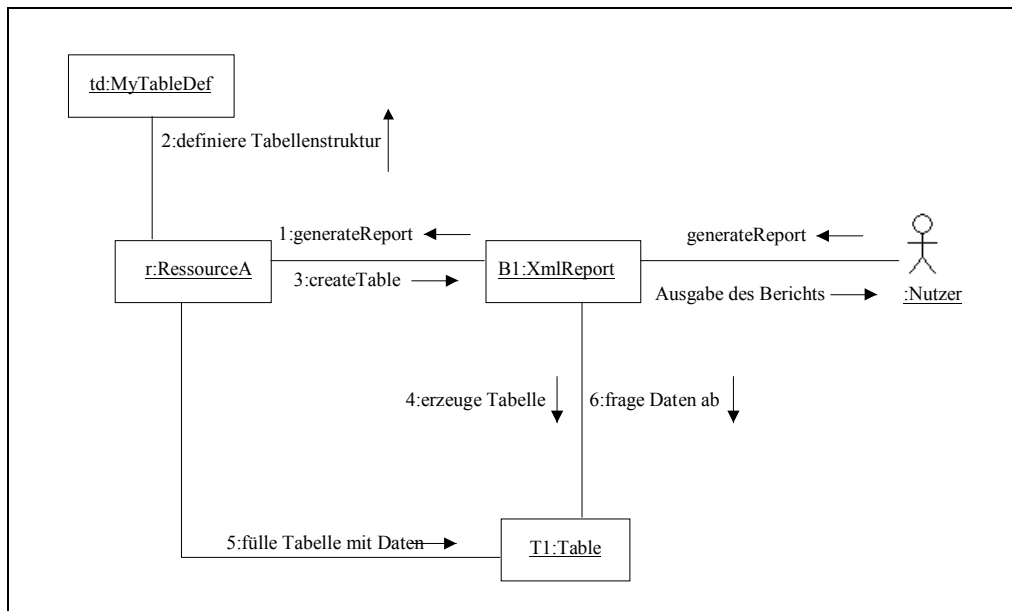


Abbildung 11



### Die Klassen Report und ReportProducer

Die Klasse `Report` erfüllt drei Aufgaben. Die erste Aufgabe besteht in der Verwaltung der registrierten `ReportProducer`. Die zweite Aufgabe ist die Verwaltung der Tabellen eines Berichtes. Die letzte Aufgabe ist die Steuerung der Berichterzeugung. Damit bedient die Klasse `Report` zwei Gruppen von Anwendern. Die erste Gruppe registriert die verschiedenen `ReportProducer` am `Report`-Objekt und initiiert die Erstellung des Berichtes. Diese Gruppe kann darüber hinaus durch Ableitung von der Klasse `Report` eigene Varianten einer Ausgabe definieren.

Die zweite Gruppe besteht aus den `ReportProducer` Objekten, die Tabellen erstellen und füllen. Sie wird durch die Klassen `Report` von der direkten Ausgabe abgekoppelt.

```
class Report {
public:
    Report() {}
    ~Report();

    // Verwaltung der Tabellen
    Table* createTable(const char* name, TableDefinition* def);
    Table* findTable(TableDefinition* def, const char* name);

    // Verwaltung der ReportProducer
    void addProducer(ReportProducer* rp);
    void removeProducer(ReportProducer* rp);

    // Steuerung der Berichterstellung
    void generateReport();

protected:
    // contained Tables
    std::vector<Table*> ts;

    virtual void processTables() = 0;
};
```

Die erste Gruppe von Anwendern verwendet im Wesentlichen die Funktionen `addProducer`, `removeProducer` und `generateReport`. Bei Bedarf implementiert sie die Funktion `processTables` in eigenen Ableitungen von `Report`.

Mit den Funktionen `addProducer` und `removeProducer` können `ReportProducer` an- bzw. abgemeldet werden. Die eigentliche Erzeugung wird durch die Methode `generateReport` angestoßen. Diese Funktion ruft ihrerseits alle registrierten `ReportProducer` auf, damit diese ihren Beitrag zum Bericht hinzufügen können. Abschließend springt `generateReport` zur virtuellen Methode `processTables`.

Die Funktion `processTables` dient als Ansatzpunkt für die Erweiterung und Anpassung des *Report*-Konzeptes. Ableitungen von `Report` implementieren `processTables`, um die Tabellen im Container `ts` auszuwerten.

`ReportProducer` verwenden die Funktionen `createTable` und `findTable`. `createTable` verlangt Namen und Strukturdefinition der Tabelle als Parameter. Mit diesen Parametern wird zunächst untersucht, ob bereits eine Tabelle gleicher Struktur und gleichen Namens innerhalb des Berichts existiert. Nur wenn dies nicht der Fall ist, wird eine neue Tabelle erzeugt, sonst wird die gefundene Tabelle zu-

rückgegeben. Mit `findTable` kann direkt überprüft werden, ob eine entsprechende Tabelle bereits Teil des Berichtes ist.

Durch die Vorkehrungen der Funktion `createTable` und dem Angebot der Funktion `findTable` ist es möglich, dass mehrere `ReportProducer` ihre Daten in eine gemeinsam genutzte Tabelle schreiben können. Wird nur die Funktion `createTable` verwendet, kann ein `ReportProducer` unabhängig davon arbeiten, ob die zurückgegebene Tabelle neu erstellt ist oder bereits von anderen `ReportProducer` Objekten verwendet wurde.

Die Schnittstelle `ReportProducer` besteht nur aus der Funktion `report`:

```
class ReportProducer {  
    virtual void report(Report* r) = 0;  
};
```

Diese Funktion wird von einem `Report`-Objekt innerhalb seiner Funktion `generateReport` aufgerufen. Konkrete `ReportProducer` sind Instanzen von Klassen, in denen diese Funktion durch eine individuelle Berichterstattung implementiert wurde. Die Implementationen müssen das übergebene `Report`-Objekt für die Erzeugung ihrer Tabellen verwenden.

Im folgenden Beispiel wird eine Klasse spezieller Reportobjekte (`XmlReport`) zusammen mit zwei `ReportProducer`-Klassen definiert. Danach werden in funktion Objekte dieser Klassen erzeugt und einander zugeordnet:

```
// XmlReport implementiert auf der Basis von Report eine konkrete  
Ausgabe.  
class XmlReport : public Report {  
public:  
    // Implementierung einer eigenen Ausgabe  
    virtual void processTable() {  
        std::vector<Table*>::iterator i;  
        for(i=ts.begin();i!=ts.end();++i) {  
            // Verarbeitung und Ausgabe der Tabellen  
        }  
    }  
};  
  
class A : public ReportProducer {  
public:  
    virtual void report(Report* r) {  
        // erste Tabelle anfordern  
        Table* t1=r->createTable("Atabelle1", Adefinition1);  
  
        // zweite Tabelle anfordern  
        Table* t2=r->createTable("Atabelle2", Adefinition2);  
  
        // Fülle Tabellen...  
    }  
};  
  
class B : public ReportProducer {  
    virtual void report(Report* r) {  
        // Tabelle anfordern  
        Table* t=r->createTable("Btabelle1", Bdef1);  
  
        // Fülle Tabelle...  
    }  
};
```

```
void funktion() {  
    // Mehrere Berichte sollen erstellt werden  
    XmlReport myReportB;  
    XmlReport myReportAll;  
  
    // ReportProducer  
    A a1, a2;  
    B b1;  
  
    // Ein Bericht über alles  
    myReportAll.addProducer(&a1);  
    myReportAll.addProducer(&a2);  
    myReportAll.addProducer(&b1);  
  
    // Ein Bericht nur über Bs  
    myReportB.addProducer(&b1);  
  
    // Berichte werden erzeugt  
    myReportAll.generateReport();  
    myReportB.generateReport();  
}
```

XmlReport ist eine von **ODEMx** vorgegebene Bericht-Klasse, welche die gespeicherten Tabellen in XML-Text umwandelt und ausgibt. Ein Objekt dieser Klasse kann mit einem Dateinamen oder einem Ausgabestrom erzeugt werden.

### Table und TableDefinition

TableDefinition ist eine Schnittstelle, über die es möglich ist, die Struktur einer Tabelle zu erfragen. Diese Struktur besteht aus der Anzahl der Spalten sowie deren Überschriften und Datentypen. Es werden drei Datentypen für Spalten zugelassen: Ganzzahl, Realwert und Text. Zwei Objekte, die diese Schnittstelle implementieren, können auf Gleichwertigkeit überprüft werden.

Ein Table-Objekt steht immer in Beziehung zu einem Objekt des Typs TableDefinition. Ausgehend von dieser Strukturinformation werden die Daten der Tabelle strukturiert gespeichert. Die Übergabe der Daten an Table erfolgt ähnlich der Datenausgabe an cout über die Operatoren:

```
Table& operator <<(Table& t, int d)  
Table& operator <<(Table& t, long d)  
Table& operator <<(Table& t, float d)  
Table& operator <<(Table& t, double d)  
Table& operator <<(Table& t, std::string d)  
Table& operator <<(Table& t, const char* d)
```

Eine Datenübergabe an eine Tabelle mit der Struktur Ganzzahl-, Realwert- und Textspalte könnte wie folgt aussehen:

```
tab << 578 << 23.0 << "Temperatur";
```

Die Tabelle wird dabei von links nach rechts und von oben nach unten gefüllt. Die verschiedenen Operatoren müssen entsprechend den Datentypen der Spalten verwendet werden, andernfalls wird der übergebene Wert ignoriert.

Mit dem Operator

```
Table& operator <<(Table& t, CtrlCode d)
```

### 3.3 Erfassung und Auswertung der Daten

---

können zusätzlich Steuerbefehle übermittelt werden, die das Auslassen einer Spalte oder den vorzeitigen Übergang zur nächsten Zeile ermöglichen.

Der Zugriff auf die in einem Table-Objekt gespeicherten Daten kann auf zwei Arten erfolgen. Die erste Variante orientiert sich analog zur Datenspeicherung an der Verwendung von `cin`. Die Daten lassen sich über die folgenden Operatoren ermitteln:

```
Table& operator >>(Table& t, long& d)
Table& operator >>(Table& t, double& d)
Table& operator >>(Table& t, std::string& d)
```

Das Auslesen der Daten beginnt in der ersten Zeile und Spalte und schreitet danach von links nach rechts und von oben nach unten fort. Wenn die Daten der zu lesenden Zelle nicht im richtigen Format vorliegen, wird, wenn möglich, eine der Konvertierungen Ganzzahl nach Realzahl, Ganzzahl nach Text und Realzahl nach Text durchgeführt. Das Auslesen der Daten beginnt nach einem Durchlauf wiederum in der oberen linken Ecke der Tabelle.

Die Daten können aber auch gezielt mit den folgenden Funktionen erfragt werden:

```
long getINTEGER(long col, long line)
double getREAL(long col, long line)
std::string getSTRING(long col, long line)
```

Auch diese Funktionen wenden, wenn notwendig, die genannten Konvertierungen an. Falls dies bei `getINTEGER` und `getREAL` nicht möglich ist, wird 0 bzw. 0.0 zurückgegeben.

#### **dynTableDefinition**

Mit `dynTableDefinition` bietet **ODEMx** eine Implementation der `TableDefinition`-Schnittstelle an, die es erlaubt, eine Tabellenstruktur dynamisch zu erstellen. Dafür werden mit der `addColumn`-Funktion nacheinander alle Spalten der Tabelle angegeben:

```
void addColumn(const char* label, ColumnType type)
```

Die Verwendung von `dynTableDefinition` könnte wie folgt aussehen:

```
virtual void report(Report* r) {
    static dynTableDefinition* d = 0;
    if (d==0) {
        // Tabellenstruktur definieren
        d = new dynTableDefinition;

        d->addColumn("Erstens", INTEGER);
        d->addColumn("Zweitens", REAL);
        d->addColumn("Drittens", STRING);
    }

    // Tabelle erzeugen
    Table* t=r->createTable("meineTabelle", d);

    // Tabelle füllen
    *t << 1 << 2.1 << "drei";
}
```

Zunächst wird ein Objekt der Klasse `dynTableDefinition` erzeugt. Danach werden drei Spalten angefügt. Nach der Erzeugung der Tabelle durch das `Report` Objekt wird die Tabelle gefüllt. Das Ergebnis dieses Beispiels wäre die folgende Tabelle:

Erstens	Zweitens	Drittens
1	2.1	drei

`meineTabelle`

### Bewertung des *Report*-Konzepts

Das *Report*-Konzept erfüllt die Anforderung, zusammenfassende Aussagen zu unterstützen. Das Sammeln und Zusammenfassen der einzelnen Informationen erfolgt durch `ReportProducer`. Diese transportieren die Daten mittels der allgemein gehaltene Schnittstelle `Table / TableDefinition`. Die Auswertung und Ausgabe übernehmen Ableitungen der Basisklasse `Report`.

Eine Verringerung der anfallenden Daten liegt zunächst in der Verantwortung der `ReportProducer`. Durch die beliebige Zuordnung von `ReportProducer`- und `Report`-Objekten wird aber die Auswahl von Informationen unterstützt, wodurch sich auch das Datenaufkommen verringern lässt.

Das *Report*-Konzept ist in zwei Richtungen erweiterbar. Zum einen kann man neue `ReportProducer` und zum anderen neue Ableitungen der `Report`-Klasse hinzufügen. Für verschiedene Modelle und Abstraktionsebenen lassen sich daher angepasste Änderungen vornehmen.

Da `ReportProducer` ihre Daten über den allgemein gehaltenen Container `Table` übermitteln, wird eine Trennung erreicht, die zur Austauschbarkeit der `Report`-Objekte führt.

### 3.3.3.3 Anwendung von *Observation*

Das in Abschnitt 3.2 Objekt-Objekt-Kopplung vorgestellte Verfahren lässt sich für eine gezielte Erfassung von Daten ausnutzen. Mit Hilfe der Kopplung können beispielsweise Auswertungsobjekte ausgewählte Modellkomponenten überwachen.

### Anwendungsmöglichkeiten

Für *Observation* lassen sich viele Anwendungsfälle finden. Im Rahmen der Erfassung und Auswertung von Daten sind zwei Fälle von Interesse. Der erste Anwendungsfall ist die Bindung von Auswertungskomponenten an Modellbausteine. Ein Beispiel für diese Anwendung ist die statistische Auswertung der Nutzung einer *Master-Slave*-Synchronisation. In **ODEM** führt die *Master-Slave*-Synchronisation `Waitq` unter anderem Protokoll über die Gesamtzahl der Objekte, die als *Master* oder *Slave* im Laufe der Simulation beteiligt waren. Da es sich dabei um eine häufig benötigte Information handelt, ist es zweckmäßig, sie aus Effizienzgründen innerhalb des `Waitq` Objektes zu erfassen. Darüber hinaus könnte es in einem konkreten Modell aber von Interesse sein, neben der allgemeinen Benutzerzahl zusätzlich *Slave*-Objekte zu zählen, die bestimmte Kriterien erfüllen. Das *Observation* Konzept ermöglicht es, für diese Situation einen speziellen Zähler zu programmieren, und ihn als Observer an `Waitq` zu binden.

### 3.3 Erfassung und Auswertung der Daten

---

Im folgenden wird eine vereinfachte Schnittstelle der Klasse `Waitq` dargestellt. Die Funktion `coopt` wird von Prozessen verwendet, um sich als *Master* anzumelden. Mit der Funktion `wait` synchronisieren sich Prozesse als *Slave*:

```
class Waitq : public Observable<WaitqObserver> {
public:
    Waitq(WaitqObserver* o = 0);
    ~Waitq();

    // Anmeldung als Slave-Partner
    void wait();
    // Synchronisierung als Master mit Rückgabe des Slave
    Process* coopt();
};
```

Die passende, ebenfalls vereinfachte, Überwachungsschnittstelle `WaitqObserver` bietet Ereignisse für *Slave*- und *Master*-Synchronisation:

```
class WaitqObserver {
public:
    // Synchronisation als Slave
    virtual void onWait(Waitq* sender, Process* slave) = 0;

    // blockierende Synchronisation als Master
    virtual void onCooptFail(Waitq* sender, Process* master) = 0;

    // erfolgreiche Synchronisation
    virtual void onCooptSucceed(Waitq* sender,
                                Process* master, Process* slave) = 0;
};
```

Den im Beispiel beschriebenen *Slave*-Zähler würde man wie folgt definieren:

```
class MySlaveCounter : public virtual WaitqObserver {
public:
    int n;

    MySlaveCounter() n(0) {};

    // Synchronisation als Slave
    virtual void onWait(Waitq* sender, Process* slave) {
        if (/*Soll slave gezählt werden?*/)
            n++;
    }

    // blockierende Synchronisation als Master
    virtual void onCooptFail(Waitq* sender, Process* master){};

    // erfolgreiche Synchronisation
    virtual void onCooptSucceed(Waitq* sender,
                                Process* master, Process* slave){};
};
```

Der zweite Anwendungsfall ist die selektive Datenerfassung. Dabei werden Aufzeichnungs- oder Informationsfilterobjekte an ausgewählte Modellkomponenten gebunden. Diese Bindung kann verändert werden, um abwechselnd verschiedene Informationen zu erfassen. Durch die frühzeitige Auswahl der überwachten Modellkomponenten wird dabei das Datenaufkommen verringert.

### **Bewertung der Anwendung von *Observation***

*Observation* ist sowohl für die Erstellung von Zusammenfassungen als auch für die Ablaufverfolgung anwendbar. Dabei wird durch Auswahlmechanismen ein intuitiver und effizienter Weg für die Reduzierung des Datenaufkommens beschrieben. Demgegenüber steht der Mangel an Vollständigkeit. Es ist zwar möglich, mit den Mitteln des Konzepts ein vollständiges Bild über den Ablauf einer Simulation zu erhalten, aber praktisch würde das sowohl einen großen zusätzlichen Entwicklungsaufwand, als auch einen unübersichtlichen Initialisierungsaufwand innerhalb des Simulationsprogramms nach sich ziehen.

Die Anforderungen „Unterstützung von Abstraktionsebenen“, „Unterstützung verschiedener Ausgabeformate“ sowie „Austauschbarkeit“ werden zusammen mit einer eingeschränkten Trennung von Modell und Auswertung nur im begrenzten Umfang erreicht. Da die Objekt-Objekt-Kopplung über spezielle und individuelle Überwachungsschnittstellen erfolgt, lässt sich nicht jeder Überwacher an jedes Objekt binden.

### **3.4 Verwendung von Templates am Beispiel Ressourcen**

**ODEM** bietet zwei vorgefertigte Modellbausteine für die Modellierung von Ressourcen. Von diesen ausgehend wurden für **ODEMX** weiterentwickelte Bausteine entworfen. Dabei kamen C++-Templates zur Anwendung. Ziel dieser Untersuchungen war sowohl die Bereitstellung weiterer Modellbausteine als auch die Darstellung der Verwendung von Templates im Bereich der Prozesssimulation.

In den folgenden Abschnitten wird zunächst die Umsetzung der Ressourcenbausteine von **ODEM** vorgeführt. Darauf aufbauend werden die beiden Bausteine `BinT` und `ResT` vorgestellt, welche die Verwaltung konkreter Ressourcenelemente mit Hilfe von C++-Templates realisieren.

#### **3.4.1 Bin und Res**

**ODEM** stellt zwei Varianten von Ressourcen bereit: `Res` und `Bin`. Beide werden für die Synchronisation von Prozessen mithilfe von Mengen verwendet. Sie unterscheiden sich darin, dass `Bin` beliebig viele Elemente in der verwalteten Menge erlaubt, während `Res` die maximale Anzahl beschränkt. Beide Bausteine abstrahieren jedoch von den tatsächlichen Elementen und arbeiten lediglich mit der Größe der verwalteten Menge sowie den Größen der angeforderten bzw. übergebenen Mengen.

Die Synchronisation von Prozessen mithilfe der `Bin`-Ressource kann beispielsweise wie folgt ablaufen:

- Die von `Bin` verwaltete Menge hat eine Größe von fünf.
- Ein Prozess nimmt vier Elemente aus `Bin`. `Bin` enthält nur noch ein Element.
- Ein weiterer Prozess will zwei Elemente aus `Bin` entnehmen. Da nicht mehr genügend Elemente vorhanden sind, wird dieser Prozess blockiert. Erst wenn genügend Elemente bereitstehen, wird er reaktiviert.
- Während der zweite Prozess blockiert ist, übergibt ein dritter (Erzeuger-)Prozess drei neue Elemente an `Bin`. Daraufhin erfolgt die Reaktivierung des wartenden Prozesses.

- Der reaktivierte Prozess erhält zwei Elemente.

Res und Bin lassen sich in den verschiedensten Fällen einsetzen. So kann man Res beispielsweise für die Modellierung eines Fahrzeugparks verwenden. Mit Bin können Puffer zwischen Produktionsanlagen und Konsumenten nachgebildet werden. Da jedoch beide keine echten Elemente verwalten, sind bei der Verwendung von Res und Bin unter Umständen weitere Hilfskonstrukte notwendig.

**ODEMx** enthält Portierungen von Res und Bin aus **ODEM**. Die Umsetzung erfolgt mithilfe einer sortierten Liste von Prozessen: `ProcessQueue`. `ProcessQueue` unterstützt benutzerdefinierte (Halb-)Ordnungen. Es werden jedoch bereits zwei typische Halbordnungen vordefiniert. `DefaultOrder` sortiert nach Ausführungszeitpunkt und, bei Gleichzeitigkeit, nach Prozesspriorität. `PriorityOrder` sortiert nur nach der Priorität der Prozesse. Wenn mehrere Prozesse entsprechend der verwendeten Halbordnung gleich sind, entscheidet die Angabe eines FIFO-LIFO Auswahlparameters, ob ein Prozess am Anfang oder am Ende der Liste „gleicher“ Prozesse eingeordnet wird.

**ODEMx** verwendet `ProcessQueue` für die Verwaltung schlafender bzw. blockierter Prozesse (Prozesszustand ist `IDLE`). Deshalb sind für `ProcessQueue` drei Funktionen definiert, mit denen sich Prozesse aus einer Warteliste aktivieren lassen. Mit `awakeAll` werden alle Prozesse aktiviert, während `awakeFirst` den ersten Prozess in der Schlange aufweckt und `awakeNext` den ersten Prozess nach einem angegebenen Referenzprozess reaktiviert.

Ein Prozess kann stets nur in einer `ProcessQueue` einsortiert sein. Wird versucht, einen Prozess in eine zweite `ProcessQueue` einzufügen, bricht das Programm mit einer Fehlermeldung ab. Wenn man die Priorität eines Prozesses verändert und dieser in einer `ProcessQueue` enthalten ist, ändert sich seine Position innerhalb der Schlange entsprechend der neuen Priorität.

```
struct ProcessOrder {
    virtual bool operator()(const Process& f, const Process& s)=0;
};

extern struct DefaultOrder : public ProcessOrder {
    virtual bool operator()(const Process& f, const Process& s);
} defOrder;

extern struct PriorityOrder : public ProcessOrder {
    virtual bool operator()(const Process& f, const Process& s);
} priOrder;

class ProcessQueue {
    // Interface
public:
    ProcessQueue(ProcessOrder* pred = &defOrder);

    // Access methodes
    Process* getTop() const;
    const std::list<Process*>& getList() const;

    bool isEmpty() const;
```



```
// Manipulator methodes
void popQueue();
void remove(Process* p);
void inSort(Process* p, bool fifo = true);

// Implementation
private:
    std::list<Process*> l;
    ProcessOrder* order;
};

void awakeAll(ProcessQueue* q);
void awakeFirst(ProcessQueue* q);
void awakeNext(ProcessQueue* q, Process* p);
```

Die Schnittstelle der Klasse Bin aus ODEMx beinhaltet die Funktionen take und get, mit denen Elemente aus der verwalteten Menge entnommen bzw. der verwalteten Menge zugeführt werden können. Die Anzahl der verfügbaren Elemente kann man mit getTokenNumber erfragen. Eine Liste der momentan wartenden Prozesse wird durch die Methode getWaitingProcessList zurückgegeben. Die ProcessQueue takeWait unterstützt die Verwaltung blockierter Prozesse.

```
class Bin : public Observable<BinObserver>,
            public DefLabeledObject,
            public virtual TraceProducer {
public:
    Bin(Simulation* s, Label l,
        unsigned int startTokenNumber,
        BinObserver* o = 0);
    ~Bin();

    void take(unsigned int n);
    void give(unsigned int n);

    unsigned getTokenNumber() const;
    const std::list<Process*>& getWaitingProcessList() const;
    ...

    // Implementation
private:
    // simulation
    Simulation* env;
    // available number of token
    unsigned int tokenNumber;
    // process management
    ProcessQueue takeWait;
    ...
};
```

Die Funktionen take und give sind wie folgt implementiert:

```
void Bin::take(unsigned int n) {
    // compute order of service
    takeWait.inSort(getCurrentProcess());

    if (n>tokenNumber ||
        takeWait.getTop()!=getCurrentProcess()) {
        // not enough tokens or not the
        // first process to serve
        ...

        // block execution
```

```
        while (n>tokenNumber ||
               takeWait.getTop()!=getCurrentProcess())
            getCurrentProcess()->sleep();

        // block released
    }
    ...

    // take token
    tokenNumber -= n;
    ...

    // remove from list
    takeWait.remove(getCurrentProcess());

    // awake next process
    awakeFirst(&takeWait);
}

void Bin::give(unsigned int n) {
    ...

    // releas token
    tokenNumber += n;
    ...

    // awake process waiting for give
    awakeFirst(&takeWait);
}
```

Die Funktion `give` fügt die Anzahl der übergebenen Elemente zur Anzahl der verfügbaren Elemente hinzu (`tokenNumber += n;`) und aktiviert den ersten wartenden Prozess (`awakeFirst(&takeWait);`). In der Funktion `take` wird zuerst der Prozess in die Liste wartender Prozesse einsortiert. Solange der Prozess nicht am Anfang dieser Liste steht (`takeWait.getTop() != getCurrentProcess()`) erfolgt keine Bedienung. Wenn genügend Elemente bereitstehen, wird die Zahl der verfügbaren Elemente angepasst (`tokenNumber -= n;`), der Prozess aus der Liste wartender Prozesse entfernt (`takeWait.remove(getCurrentProcess());`) und die Bedienung weiterer Prozesse angestoßen (`awakeFirst(&takeWait);`). Andernfalls wird der Prozess bis genügend Elemente bereitstehen (`while (n>tokenNumber) || ...`), schlafen gelegt (`getCurrentProcess()->sleep();`).

Die Schnittstelle der Klasse `Res` bietet die Funktionen `acquire` und `release`, um Elemente von der verwalteten Menge anzufordern, bzw. an die verwaltete Menge zurückzugeben. Dem Konstruktor der Klasse wird zusätzlich zur Anfangsgröße der verwalteten Menge deren maximal Größe übergeben. Für die Veränderung der maximalen Größe der Menge nach der Konstruktion des Objektes, stehen die Methoden `control` und `unControl` bereit. Die maximale Größe lässt sich dabei mit `control` erhöhen und mit `unControl` verringern. Die Funktionen `getTokenNumber` und `getMaxTokenNumber` geben die Anzahl der aktuell verfügbaren Elemente bzw. die momentane maximale Größe zurück. Mit `getWaitingForAcquire` kann man die wartenden Prozesse erfragen. Die `ProcessQueue` `acquireWait` realisiert die Verwaltung der wartenden Prozesse.

```
class Res : public Observable<ResObserver>,
           public DefLabeledObject,
           public virtual TraceProducer {
public:
    Res(Simulation* s, Label l,
        unsigned int startTokenNumber,
        unsigned int maxTokenNumber,
        ResObserver* o = 0);
    ~Res();

    void acquire(unsigned int n);
    void release(unsigned int n);

    void control(unsigned int n);
    unsigned int unControl(unsigned int n);

    unsigned int getTokenNumber() const;
    unsigned int getMaxToken() const ;

    const std::list<Process*>& getWaitingForAcquire() const;
    ...

private:
    // simulation
    Simulation* env;
    // current number of tokens
    unsigned int tokenNumber;
    // max number of tokens
    unsigned int maxToken;
    // process management
    ProcessQueue acquireWait;
    ...
};
```

Die Funktionen `acquire` und `release` wurden analog zu den Funktionen `take` und `give` aus `Bin` implementiert. Im Unterschied zu `give` überprüft `release` jedoch ob die maximale Anzahl der Elemente überschritten wird. Falls dies der Fall ist, wird eine Fehlermeldung ausgegeben. Die überschüssigen Elemente werden ignoriert.

```
void Res::acquire(unsigned int n) {
    // compute order of service
    acquireWait.inSort(getCurrentProcess());

    if (n>tokenNumber ||
        acquireWait.getTop()!=getCurrentProcess()) {
        // not enough tokens or not the
        // first process to serve
        ...

        // block execution
        while (n>tokenNumber ||
            acquireWait.getTop()!=getCurrentProcess())
            getCurrentProcess()->sleep();

        // block released
    }
    ...

    // acquire token
    tokenNumber -= n;
    ...
}
```

```
// remove from list
acquireWait.remove(getCurrentProcess());

// awake next process
awakeFirst(&acquireWait);
}

void Res::release(unsigned int n) {
    if ((tokenNumber+n)>maxToken) {
        // Not enough room left.
        ...

        // Error:
        error("Res: could not release all tokens.");

        // change token number
        n=maxToken-tokenNumber;
    }
    ...

    // release tokens
    tokenNumber += n;
    ...

    // awake process waiting for release
    awakeFirst(&acquireWait);
}
```

Die Methoden `control` und `unControl` wurden wie folgt implementiert:

```
void Res::control(unsigned int n) {
    // add token
    tokenNumber+=n;
    maxToken+=n;

    // awake process waiting for release
    awakeFirst(&acquireWait);
}

unsigned int Res::unControl(unsigned int n) {
    // secure enough token
    acquire(n);

    // remove token
    maxToken-=n;

    return n;
}
```

In der Funktion `control` erhöht sich die Anzahl der verfügbaren Elemente zusammen mit der Anzahl der maximal erlaubten Elemente. Dadurch werden unter Umständen die Forderungen wartender Prozesse erfüllbar. Entsprechend wird die Überprüfung der Forderungen angestoßen. Die Methode `unControl` kann wiederum nur dann Elemente aus der verwalteten Menge entfernen, wenn diese verfügbar sind. Deshalb wird in `unControl` zunächst mit `acquire` eine entsprechend große Menge angefordert. Ein Aufruf von `unControl` führt daher gegebenenfalls zu einer Blockierung des aktiven Prozesses.

Res und Bin verwenden für die Bedienung der Anforderungen die gleiche Strategie. Die Prozesse in der `ProcessQueue` werden streng der Reihe nach bedient. Erst wenn der erste Prozess die angeforderten Elemente erhalten hat, wird der nächste Prozess betrachtet. Zu dieser Strategie gibt es mehrere Alternativen, die jedoch nicht unterstützt werden. Eine solche Alternative könnte z.B. darin bestehen, dass trotz Blockierung des ersten Prozesses kleinere Anforderungen anderer Prozesse erfüllt werden.

### 3.4.2 BinT

Das neu entwickelte Template `BinT` bietet die Funktionalität von `Bin` im Zusammenhang mit der Verwaltung konkreter Elemente. Anders als in `Bin` und `Res` wird nicht nur die Anzahl der verfügbaren Elemente betrachtet. `BinT` ist zusätzlich ein Container, in dem freie Objekte gespeichert sind. Um diesen Container vielseitig einsetzen zu können, wird er als Template realisiert.

Der Typ der verwalteten Elemente wird `BinT` als Templateparameter `Token` übergeben. Die Speicherung verfügbarer Objekte erfolgt aus Effizienzgründen in einem `std::vector<Token>` (`tokenStore`). Analog zu `Bin` bietet `BinT` die Funktionen `take` und `give` für die Entnahme bzw. Übergabe von Elementen. Da im Allgemeinen mehrere Objekte transferiert werden, kommt auch hier `std::vector<Token>` zum Einsatz. Wartende Prozesse verwaltet `BinT` mit einer `ProcessQueue` (`takeWait`).

```
template <typename Token>
class BinT : public Observable<BinTObserver<Token> >,
             public DefLabeledObject,
             public virtual TraceProducer {
public:
    BinT(Simulation* s, Label l, BinTObserver<Token>* o=0);
    ~BinT();

    std::vector<Token>* take(unsigned int n);
    void give(std::vector<Token>*& t);

    const std::list<Process*>& getWaitingProcessList() const;
    ...

    // Implementation
private:
    // simulation
    Simulation* env;
    // token
    std::vector<Token> tokenStore;
    // process management
    ProcessQueue takeWait;
    ...
};
```

Die Funktion `take` ist wie folgt implementiert:

```
template <typename Token>
std::vector<Token>* BinT<Token>::take(unsigned int n) {
    // compute order of service
    takeWait.inSort(getCurrentProcess());

    if (tokenStore.size()<n ||
        takeWait.getTop()!=getCurrentProcess()) {
        // not enough token or
        // not first process
        ...

        // block execution
        while (tokenStore.size()<n ||
               takeWait.getTop()!=getCurrentProcess())
            getCurrentProcess()->sleep();

        // block released
    }

    // create return vector
    std::vector<Token>* tokenReturn = new std::vector<Token>(n);

    // fill return vector
    tokenReturn->insert(tokenReturn->end(),
                       tokenStore.begin(),
                       (tokenStore.begin()+n));
    ...

    // remove token from store
    tokenStore.erase(tokenStore.begin(), (tokenStore.begin()+n));
    ...

    // remove process from list
    takeWait.remove(getCurrentProcess());

    // awake next
    awakeFirst(&takeWait);

    return tokenReturn;
}
```

take erhält die Anzahl der angefordert Elemente als Parameter *n*. Anhand dieses Parameters wird, im Zusammenhang mit der Größe des Vektors *tokenStore*, überprüft, ob genügend Elemente verfügbar sind (*tokenStore.size()<n*). Zunächst erfolgt jedoch die Einsortierung des Prozesses in die Warteliste (*takeWait.inSort(getCurrentProcess())*). Nur wenn der Prozess am Anfang dieser Liste steht, wird er bedient.

Solange ein Prozess nicht an der Reihe ist oder nicht genügend Elemente verfügbar sind, wird er blockiert. Für die Übergabe der angeforderten Elemente legt take einen neuen *std::vector<Token>* an (*tokenReturn*) und füllt diesen mit Kopien der ersten *n* Elementen aus *tokenStore*. Die Originale der angeforderten Elemente werden aus *tokenStore* entfernt (*tokenStore.erase(...)*). Abschließend folgt die Rückgabe des neuen Vektors.

Die Funktion *give* erwartet einen Zeiger auf *std::vector<Token>*, der als Referenz übergeben wird (*t*). *\*t* enthält neue Elemente für den Container *BinT*. Nach der Überprüfung des Parameters (*t==0*) kopiert *give* den Inhalt von *\*t* am Ende

von `tokenStore` ein (`tokenStore.insert(...);`) und löscht `*t`. Abschließend reaktiviert `give` den ersten wartenden Prozess (`awakeFirst(&takeWait);`).

```
template <typename Token>
void BinT<Token>::give(std::vector<Token>*& t) {
    if (t==0) {
        // Error: no tokens provided in give()
        error("not tokens provided in give()");
        return;
    } ...
    // transfer token to tokenStore
    tokenStore.insert(tokenStore.end(), t->begin(), t->end());
    delete t; t=0;

    // wake up waiting process objects
    awakeFirst(&takeWait);
}
```

`BinT` ersetzt nicht den Modellbaustein `Bin`. Während `Bin` eine effiziente Synchronisation von Prozessen über die Anzahl verfügbarer Element bietet, unterstützt `BinT` den Anwender durch eine integrierte Verwaltung konkreter Elemente. Beide Bausteine stellen somit Lösungen für zwei verschiedene Anwendungsfälle dar.

Die Realisierung einer integrierten Verwaltung konkreter Elemente ist auch ohne die Verwendung von Templates möglich. Durch deren Nutzung wird jedoch eine typsichere Übertragung und Speicherung der verwalteten Elemente erreicht. Darüber hinaus überträgt sich die Flexibilität von STL-Containern an den Modellbaustein.

Zur dargestellten Realisierung gibt es mehrere Alternativen und weitergehende Ideen. Beispielsweise könnte man eine gezielte Entnahme von Elementen aus der Ressource erlauben, indem man Such-Funktionen anbietet. Im Zusammenhang dazu könnte man einen `std::list` Container anstelle des `std::vector` verwenden. Weiterhin ließe sich die Verwaltung verfügbarer Elemente verändern. Zur momentanen Queue-Variante (Entnahme am Anfang und Hinzufügen am Ende) bietet sich eine Stack-Variante (Entnahme und Hinzufügen am Anfang) als Alternative an. Schließlich könnte man durch zusätzliche Templateparameter weitere Konfigurationsmöglichkeiten für `BinT` bereitstellen. Eine solche Einstellung könnte beispielsweise die Sortierstrategie der `ProcessQueue` verändern.

### 3.4.3 ResT

`ResT` ist ein Template, mit dessen Hilfe man die Zuteilung konkreter Elemente zu Klienten (Prozessen) verwalten kann. Der Typ der Elemente wird als Templateparameter übergeben (`Token`). Die Speicherung der Zuteilung erfolgt in einer `std::multimap<...>` in der die Elemente als Schlüssel dient. Durch die Verwendung von `multimap` anstelle von `map` kann ein Anwender auch mehrere bezüglich des Schlüsselvergleichs gleiche Elemente in der Ressource verwalten.

Ähnlich wie in `BinT` können Prozesse konkrete Elemente bei `ResT` anfordern. `ResT` führt jedoch genau Buch darüber, welcher Prozess welche Elemente erhalten hat. Ein Prozess kann daraufhin nur Elemente freigeben, die ihm vorher zugeteilt wurden.

### 3.4 Verwendung von Templates am Beispiel Ressourcen

---

```
template <typename Token>
class ResT : public Observable<ResTObserver<Token> >,
            public DefLabeledObject,
            public virtual TraceProducer {
public:
    // function type for coding selections
    typedef bool(*Selection)(Process* owner, Token& t);

    ResT(Simulation* s, Label l, std::multiset<Token>& initSet,
        unsigned int maxTokenNumber=0, ResTObserver<Token>*o=0);
    ~ResT();

    Token& acquire();
    std::multiset<Token&> acquire(unsigned int n);

    void release(Token& e);
    void release(std::multiset<Token&>& s);

    Token& find(ResT<Token>::Selection select);
    std::multiset<Token&> find(ResT<Token>::Selection select,
        unsigned int n);

    void transfer(Token& e, Process* oldOwner, Process* newOwner);
    void transfer(std::multiset<Token&>& s, Process* oldOwner,
        Process* newOwner);

    void control(Token e);
    void control(std::multiset<Token>& s);

    Token unControl();
    std::multiset<Token> unControl(unsigned int n);

    unsigned int getTokenNumber() const {return tokenNumber;}
    unsigned int getMaxToken() const {return tokenStore.size();}
    const std::multiset<const Token&> getTokenSet() const;

    // get list of blocked process objects
    const std::list<Process*>& getWaitingProcessList() const;
    ...

    // Implementation
private:
    // simulation
    Simulation* env;
    // current number of tokens
    unsigned int tokenNumber;
    // token
    std::multimap<Token, Process*> tokenStore;
    // process management
    ProcessQueue acquireWait;
    ...

    bool _find(ResT<Token>::Selection select, unsigned int n,
        std::vector<std::multimap<Token,
        Process*>::iterator>& result);
};
```

Die Zuteilung der Elemente zu Prozessen verwaltet ResT in der `std::multimap tokenStore`. Für jedes Element der verwalteten Menge existiert in `tokenStore` ein Eintrag [Element -> Prozess]. Freie Elemente verweisen auf den 0-Zeiger. Die Verwendung von `std::multimap` anstelle von `std::map` ist notwendig, da ge-



benenfalls mehrere in Bezug auf die map-Sortierung gleiche Elemente in einer ResT enthalten sein können. Diese Möglichkeit ist abhängig vom Typ der Elemente (Token). Für eine vereinfachte Umsetzung wird die Anzahl der verfügbaren Elemente zusätzlich in tokenNumber gespeichert. Schlafende Prozesse verwaltet ResT in der ProcessQueue acquireWait.

ResT stellt die Funktionen acquire und release für die Anforderung bzw. Freigabe von Elementen bereit. Mit find kann ein Prozess ausgesuchte Elemente anfordern. Die Funktionen control und unControl stehen dem Anwender für die nachträgliche Veränderung der Menge verwalteter Elemente zur Verfügung. Weiterhin kann man die Übergabe zugeteilter Elemente an einen anderen Prozess mit der Funktion transfer dem ResT Objekt mitteilen.

Die Funktion acquire ist wie folgt realisiert:

```
template <typename Token>
std::multiset<Token&> ResT<Token>::acquire(unsigned int n) {
    if (n>tokenStore.size()) {
        // Error: will always block
        ...
    }
    // compute order of service
    acquireWait.inSort(getCurrentProcess());

    if (n>tokenNumber ||
        acquireWait.getTop()!=getCurrentProcess()) {
        // not enough token or not
        // first process
        ...

        // block execution
        while (n>tokenNumber ||
            acquireWait.getTop()!=getCurrentProcess())
            getCurrentProcess()->sleep();

        // block released
    }
    ...

    // acquire token
    tokenNumber -= n;

    multimap<Token,Process*>::iterator i;
    multiset<Token&> rval;
    for (i=tokenStore.begin(); i!=tokenStore.end(); i++)
        if ((*i).second==0) {
            (*i).second=getCurrentProcess();
            rval.insert((*i).first);
        }
    ...

    // remove process from list
    acquireWait.remove(getCurrentProcess());

    // awake next
    awakeFirst(&acquireWait);

    // return token
    return rval;
}
```

```
template <typename Token>
Token& ResT<Token>::acquire() {
    std::multiset<Token&> s=acquire(1);
    return (*(s.begin())).first;
}
```

acquire überprüft zunächst anhand der vorhandenen Elemente (tokenStore.size()), ob eine Bedienung überhaupt möglich ist. Wenn nicht genügend Elemente in der verwalteten Menge enthalten sind, kommt es immer zu einer Blockierung. Im Zusammenhang mit control, das ein späteres Hinzufügen von Elementen erlaubt, kann diese Situation jedoch beabsichtigt sein.

Danach sortiert acquire, analog zu Res::acquire, den Prozess in die Warteliste ein. Nur wenn der Prozess an erster Stelle in dieser Liste steht, wird er bedient. Anhand von tokenNumber überprüft die Funktion, ob genügend Elemente verfügbar sind. Dies erspart die Durchsuchung von tokenStore nach freien Elementen [Element -> Nullzeiger]. Solange nicht ausreichend freie Elemente bereit stehen oder der Prozess nicht an der ersten Stelle in der Warteliste steht, wird er blockiert.

Die Übergrabe der Elemente beginnt mit der Anpassung von tokenNumber (tokenNumber -= n;). Danach werden die freien Elemente in tokenStore gesucht (for (...) if ((\*i).second==0) {...}). acquire übergibt keine Kopien der Objekte (vgl. BinT) sondern Referenzen, welche die Funktion im std::multiset rval zusammenfasst. Abschließend wird der Prozess aus der Warteschlange entfernt, die Überprüfung weiterer Anfragen angestoßen und die Ergebnismenge rval zurückgegeben.

Für die Entnahme ausgewählter Elemente stehen die find-Funktionen zur Verfügung. Diese sind wie folgt implementiert:

```
template <typename Token>
Token& ResT<Token>::find(ResT<Token>::Selection select) {
    std::multiset<Token&> s=find(select, 1);
    return (*(s.begin())).first;
}

template <typename Token>
std::multiset<Token&> ResT<Token>::find(
    ResT<Token>::Selection select, unsigned int n)
{
    using namespace std;
    multimap<Token, Process*>::iterator i;
    vector<multimap<Token, Process*>::iterator> work;
    vector<multimap<Token, Process*>::iterator>::iterator j;

    // compute order of service
    acquireWait.inSort(getCurrentProcess());

    if (!_find(select, n, work) ||
        acquireWait.getTop()!=getCurrentProcess()) {
        // not enough tokens or not
        // first process

        while (!_find(select, n, work))
            getCurrentProcess()->sleep();

        // block released
    }
}
```

```

...

// acquire token
tokenNumber-=n;

std::multiset<Token&> rset;
for (j=work.begin(); j!=work.end(); ++j)
    rset.insert( (*j).first );

// remove from list
acquireWait.remove(getCurrentProcess());

// awake next
awakeFirst(&acquireWait);

return rset;
}

```

`find` erwartet als Parameter neben der Anzahl von Elementen (`n`) eine Auswahl-funktion (`select`). Mit der übergebenen Funktion lassen sich gezielt bestimmte Objekte aus der verwalteten Menge anfordern. Dadurch wird jedoch die Überprüfung, ob genügend Elemente verfügbar sind und die Zusammenstellung der Ergebnismenge komplizierter. Diese Aufgaben wurden zur besseren Übersichtlichkeit in die Hilfsfunktion `_find` ausgelagert. `_find` erwartet als Parameter die Auswahl-funktion (`select`), die Anzahl der angeforderten Elemente (`n`) und eine *by reference* übergebene Rückgabemenge (`work`). In der Rückgabemenge werden durch `_find` Verweise auf passende Einträge in `tokenStore` gespeichert. Die Hilfsfunktion gibt `true` zurück, falls `n` freie Elemente, die der Auswahl `select` entsprechen, gefunden wurden. Eine Implementation von `_find` vorausgesetzt, stellt sich `find` ähnlich wie `acquire` dar.

Die Funktion `_find` wurde wie folgt realisiert:

```

template <typename Token>
bool _find(ResT<Token>::Selection select, unsigned int n,
           std::vector<std::multimap<Token,
Process*>::iterator>& result)
{
    unsigned int count=0;
    std::multimap<Token, Process*>::iterator i;
    result.resize(tokenStore.size());

    // find token
    for (i=tokenStore.begin(); i!=tokenStore.end(); ++i) {
        if (select(getCurrentProcess(), (*i).first)) {
            count++;

            if ((*i).second==0) {
                result.insert(i);

                if (result.size()==n)
                    break;
            }
        }
    }

    if (count<n) {
        // Error: will always block
        ...
    }
}

```

```
        return result.size()==n;
    }
```

In der `for`-Schleife werden alle passenden Einträge gesucht. Wenn genügend freie Elemente gefunden wurden, bricht die Schleife ab. Anhand des Zählers `count` wird überprüft, ob genügend passende Elemente in der verwalteten Menge vorhanden sind. Ist dies nicht der Fall gibt `_find`, analog zu `acquire`, eine Warnung aus.

Mit den `release`-Funktionen können Prozesse zuvor mit `find` oder `acquire` angeforderte Elemente wieder freigeben. Die Funktionen erwarten dafür die freizugebenden Elemente als Parameter (`Token& e` bzw. `std::multiset<Token&>& s`).

`release` geht davon aus, dass der aktive Prozess (`current`) der eingetragene Besitzer dieser Objekte ist [`Element -> current`]. Zunächst werden für alle Elemente aus `s` die passenden Einträge in `tokenStore` ermittelt. Innerhalb dieser Einträge werden diejenigen gesucht, welche `current` zugeteilt sind. Die Freigabe erfolgt durch Änderung des Eintrages von [`Element -> current`] nach [`Element -> Nullzeiger`]. Anhand des Zählers `count` wird überprüft, ob alle übergebenen Elemente freigegeben werden konnten. Abschließend wird `tokenNumber` angepasst und die Bedienung schlafender Prozesse angestoßen.

```
template <typename Token>
void ResT<Token>::release(Token& e) {
    std::multiset<Token&> s;
    s.insert(e);
    return release(s);
}

template <typename Token>
void ResT<Token>::release(std::multiset<Token&>& s) {
    using namespace std;

    map<Token,Process*>::iterator i;
    multiset<Token&>::iterator i;
    multimap<Token,Process*>::iterator j;
    Process* current=getCurrentProcess();
    unsigned int count=0;

    for (i=s.begin(); i!=s.end(); ++i) {
        for (j=tokenStore.lower_bound(*i);
            j!=tokenStore.upper_bound(*i); ++j) {
            if ((*j).second==current) {
                (*j).second=static_cast<Process*>(0);
                count++;
            }
        }
    }

    if (count<s.size()) {
        // Error:
        error("ResT: could not release all token.");
        return;
    }
    ...

    // release token
    tokenNumber+=count;
    ...
}
```

```
        // wake up waiting process objects
        awakeFirst(&acquireWait);
    }
```

Mit den Funktionen `control` und `unControl` können nachträglich weitere Elemente der verwalteten Menge hinzugefügt bzw. enthaltene Elemente entfernt werden. Die Funktionen wurden wie folgt umgesetzt:

```
void ResT<Token>::control(Token e) {
    using namespace std;

    // Add token
    tokenStore.insert(make_pair(e, static_cast<Process*>(0)));
    tokenNumber=tokenStore.size();

    // wake up waiting process objects
    awakeFirst(&acquireWait);
}

template <typename Token>
void ResT<Token>::control(std::multiset<Token>& s) {
    using namespace std;

    // Add token
    multiset<Token>::iterator i;
    for (i=s.begin(); i!=s.end(); i++)
        tokenStore.insert(
            make_pair(*i, static_cast<Process*>(0)));

    tokenNumber=tokenStore.size();

    // wake up waiting process objects
    awakeFirst(&acquireWait);
}

template <typename Token>
Token ResT<Token>::unControl() {
    std::multiset<Token> s=unControl(1);
    return (*(s.begin())).first;
}

template <typename Token>
std::multiset<Token> ResT<Token>::unControl(unsigned int n) {
    using namespace std;

    // acquire enough token
    // compute order of service
    acquireWait.inSort(getCurrentProcess());

    if (n>tokenNumber ||
        acquireWait.getTop()!=getCurrentProcess()) {
        // not enough token or not
        // first process

        // block execution
        while (n>tokenNumber ||
            acquireWait.getTop()!=getCurrentProcess())
            getCurrentProcess()->sleep();

        // block released
    }
}
```

### 3.4 Verwendung von Templates am Beispiel Ressourcen

---

```
// remove token
tokenNumber-=n;

multimap<Token,Process*>::iterator i;
for (i=tokenStore.begin(); i!=tokenStore.end(); i++)
    if ((*i).second==0)
        tokenStore.erase(i);

// remove process from list
acquireWait.remove(getCurrentProcess());
}
```

Die control-Funktionen fügen neue Einträge [Element -> Nullzeiger] in tokenStore ein (tokenStore.insert(make\_pair(...))) und aktivieren den ersten schlafenden Prozess (awakeFirst(&acquireWait;)). unControl sichert zunächst, dass genügend freie Elemente in ResT bereitstehen. Dabei kann es zur Blockierung des aktiven Prozesses kommen. Danach werden n Einträge mit freien Elementen aus tokenStore entfernt.

Mit den transfer Funktionen können Prozesse dem ResT-Objekt den Austausch angeforderter Elemente mitteilen. Die Implementierung der beiden Funktionen ist nachfolgend aufgeführt:

```
template <typename Token>
void ResT<Token>::transfer(Token& e, Process* oldOwner,
                          Process* newOwner) {
    std::multiset<Token&> s;
    s.insert(e);
    transfer(s, newOwner);
}

template <typename Token>
void ResT<Token>::transfer(std::multiset<Token&>& s,
                          Process* oldOwner, Process* newOwner)
{
    using namespace std;
    multiset<Token&>::iterator i;
    map<Token,Process*>::iterator j;

    for (i=s.begin(); i!=s.end(); ++i)
        for (j=tokenStore.lower_bound(*i);
             j!=tokenStore.upper_bound(*i); ++j)
        {
            if ((*j).second==oldOwner) (*j).second=newOwner;
        }
}
```

transfer sucht die Elemente aus s in tokenStore, für die oldOwner als Besitzer eingetragen ist ((\*j).second==oldOwner) und weist ihnen den Prozess newOwner als neuen Besitzer zu.

ResT hilft dem Anwender Modellierungsfehler zu vermeiden, in dem es die Zuordnung von Prozessen zu Elementen kontrolliert. Im Zusammenhang mit den transfer-Funktionen wird eine überlegtere Modellierung veranlasst. Durch die Bereitstellung der find-Funktionen werden darüber hinaus die Möglichkeiten, die durch die Verwaltung konkreter Elemente entstehen, im Vergleich zu BinT besser ausgenutzt.

Die Verwendung von Templates ist auch im Fall von `ResT` nicht zwingend erforderlich. Ähnliche Funktionalität ließe sich durch die Einführung einer Basisklasse für in `ResT` verwaltete Objekte erreichen. Für die Verwaltung elementarer Typen müssten man dann jedoch *Wrapper*-Klassen oder spezielle `ResT`-Varianten bereitstellen. Die Verwendung von Templates erlaubt stattdessen eine flexible und dennoch typsichere Speicherung, Übertragung und Verwaltung von Objekten. Wie für `BinT` konnte für die Umsetzung von `ResT` auf Templates der STL zurückgegriffen werden.

`ResT` ist aufgrund der komplizierteren Verwaltung der Elemente langsamer als `BinT` und benötigt gleichzeitig mehr Speicher. In der vorliegenden Implementierung sind insbesondere die `find`-Funktionen rechenaufwändig, da jede Überprüfung der Anforderungen eine Durchsuchung der verwalteten Menge veranlasst.





---

## 4 Ausblicke

### 4.1 Introspection/ Inspection

*Introspection* bzw. *Inspection* bezeichnet die Möglichkeit, an einem übersetzten Objekt unbekannten Typs während der Laufzeit eines Programms die Attribute und Funktionen erfragen zu können. Dies ist insbesondere im Zusammenhang mit auf Komponenten basierender Softwareentwicklung von Interesse.

**ODEMx** würde in verschiedener Weise von *Inspection* profitieren. Allgemein steht für **ODEMx** vor allem die Ermittlung der Attribute eines Objektes im Vordergrund. Dieses könnten beispielsweise die Attribute eines Ereignisses oder eines überwachten Objektes sein.

Leider bietet C++, im Gegensatz zu Java, *Inspection* nicht von sich aus an. Deshalb sind eigene Erweiterungen und Konventionen für die Realisierung dieser Eigenschaft notwendig.

Es gibt solche Erweiterungen beispielsweise in CORBA. CORBA verwendet für die Ermittlung der inneren Struktur eines Objektes jedoch die Informationen aus einer zusätzlichen Schnittstellenbeschreibung mit IDL. Diese wird durch spezielle Übersetzer ausgewertet.

Für eine reine C++-Lösung sind die folgenden Konzepte denkbar:

- Verwendung manuell erstellter Informationsklassen oder Funktionen für alle Ereignisse
- Automatische Erstellung von Informationsklassen oder Funktionen durch einen speziellen Übersetzer
- Verwendung von C++-Templates für eine automatisierte Erzeugung von Strukturinformationen

Die erste Variante würde einen beträchtlichen Mehraufwand für die Entwicklung von Ereignissen bedeuten. Es müssten für jedes Ereignis zusätzliche Klassen oder Funktionen erzeugt werden. Dabei käme es zu einer Wiederholung von Programmtext, denn alle Attribute wären, zusätzlich zu ihren Definitionen in den Klassen, auch in den Funktionen für die *Inspection* anzugeben.

Die zweite Variante könnte diesen Schritt automatisieren und dadurch Fehler vermeiden. Sie erfordert aber die Entwicklung eines zusätzlichen Übersetzers, der den vollen C++-Sprachumfang beinhalten muss. Für diese Variante wäre es sinnvoll, auf eine standardisierte Lösung innerhalb der Programmiersprache C++ hinzuarbeiten.

Die dritte Variante ist vielversprechend und auch von wissenschaftlichem Interesse, sie könnte daher als Ansatzpunkt für weitere Entwicklungen dienen. Ideen und Lösungen bietet dafür z.B. [Ale01].

### 4.2 Ein generischer Ressourcenbaustein

Ausgehend von ResT und BinT aus Abschnitt 3.4 sowie [Ale01] wurden Überlegungen angestellt, ob es möglich wäre, einen generischen Ressourcenbaustein zu entwickeln. Dieser Baustein (ResourceT) könnte die Gemeinsamkeiten der ein-

zelenen Klassen `Res`, `ResT`, `Bin` und `BinT` zusammenfassen und ihre Unterschiede über Templateparameter einbinden.

Die Technologie für diese Entwicklung basiert auf den von Alexandrescu in [Ale01] beschriebenen *Policies*. *Policies* stellen Möglichkeiten zur Konfiguration und Erweiterung bestehender Klassen durch Einfügen oder Austauschen von Basisklassen bereit. Diese Technik funktioniert für Templates, da diese ihre Parameter als Basisklasse verwenden können. Im Folgenden wird ein einfaches Template zusammen mit zwei *Policies* definiert:

```
class PolicyNothing {
public:
    void doSomething() {
    }
};

class PolicySomething {
public:
    void doSomething() {
        cout << 1234;
    }
};

template <class Policy=PolicyNothing>
class Host : public Policy {
    void do() {
        doSomething();
    }
};

typedef Host Nothing;
typedef Host<PolicySomething> Something;
```

Die durch `typedef` definierten Typen basieren beide auf der Klasse `Host`, deren Verhalten jedoch durch ihren Templateparameter `Policy` gesteuert wird. Alexandrescu führt diese Techniken am Beispiel von Smartpointern vor, deren Eigenschaften (z.B. Multithreadingsicherheit) über *Policies* bestimmt sind.

Ressourcenbausteine haben zwei Hauptaspekte, die kombiniert werden. Der erste Aspekt ist die Verwaltung der Elemente einer Ressource. `Res` und `Bin` abstrahieren von den Elementen, während `ResT` und `BinT` konkrete Objekte verwalten. Im Fall von `ResT` und `BinT` wäre zudem interessant, nach welchem Verfahren man die Elemente sortiert (z.B. LIFO oder FIFO). Der zweite Aspekt ist die Verwaltung der blockierten Prozesse. Die Reihenfolge nach der diese bedient werden, könnte ebenfalls durch LIFO und FIFO bestimmt sein. Darüber hinaus ist von Bedeutung, ob eine Ressource die wartenden Prozesse streng der Reihe nach abfertigt oder Prozesse, die schneller bedient werden können, vorzieht.

Obwohl bereits erste Versuche unternommen wurden, ein `RessourceT` zu implementieren, fand sich noch keine zufriedenstellende Realisierung. Die ersten Erfahrungen lassen jedoch eine Machbarkeit vermuten. Das Ergebnis des Versuchs finden sich in der Datei `RessourceT.h` der **ODEMx**-Bibliothek auf <http://www.odemx.de>.

---

## Anhang A: Dokumentation von ODEMX

Ein wesentlicher Punkt bei der Erstellung eines Softwaresystems wie **ODEMX** ist die Dokumentation des Systems. Inzwischen werden Softwareentwickler bei der Erstellung der Dokumentationen durch Werkzeuge unterstützt.

Für die Dokumentation von **ODEMX** sind zwei Zielgruppen zu beachten. Die erste Gruppe sind die Anwender der Bibliothek. Sie benötigen eine Beschreibung der Möglichkeiten der Bibliothek und ihrer Verwendung. Das beinhaltet z.B. die Dokumentation der öffentlichen Schnittstelle der Bibliothek.

Die zweite Gruppe besteht aus denjenigen, die an **ODEMX** Veränderungen vornehmen wollen. Sie benötigen zusätzlich Informationen über die Interna der Bibliothek. Dazu gehören:

- Die interne Schnittstelle der Bibliothek.
- Die verwendeten Konzepte und Techniken.
- Kommentare zum Quelltext.
- Übersicht über die Struktur der Bibliothek.

**Doxygen**<sup>1</sup> ist ein Programm, das die Dokumentation eines Softwaresystems innerhalb des Quellcodes unterstützt. Dabei wird die Dokumentation in speziellen Kommentaren im Quellcode an den entsprechenden Stellen vorgenommen. **Doxygen** generiert aus der Struktur des Softwaresystems und den gefundenen Kommentaren eine fertige Beschreibung in verschiedenen Ausgabeformaten. Darüber hinaus bietet **Doxygen** eine Reihe von Kommandos, mit denen spezifische Informationen bereitgestellt werden können. Diese Informationen werden dann von **Doxygen** in die Dokumentation eingearbeitet.

**Doxygen** bietet einen umfangreichen Satz an Optionen und Einstellungen, mit denen die Erzeugung der Dokumentation gesteuert werden kann. Diese werden in einer Konfigurationsdatei zusammengefasst. Für die Dokumentation von **ODEMX** wurden zwei Konfigurationsdateien erstellt. Eine Konfiguration ist für die Bedürfnisse der **ODEMX**-Anwender zugeschnitten, während die zweite zusätzliche Informationen für die Entwicklung bereitstellt.

Durch die Verwendung von **Doxygen** ließ sich schnell eine Dokumentation der Bibliothek erstellen. Die sukzessiv in den Programmtext eingearbeiteten Informationen wurden durch das Programm in einer einheitlichen Form in die Dokumentation eingefügt. Dabei hat sich die Verwendung von Dokumentationsvorlagen für Dateien, Klassen und Schnittstellen als vorteilhaft erwiesen.

Es zeigten sich jedoch auch Probleme in der verwendeten Version 1.2.17. In der erzeugten Dokumentation fehlten beispielsweise einige Referenzen zwischen Programmbeispielen und demonstrierten Klassen und Funktionen. Diese Referenzen werden im Normalfall durch **Doxygen** automatisch erzeugt und lassen sich nicht manuell anlegen. An anderen Stellen scheint **Doxygen** Probleme mit einigen Programmstrukturen zu haben, wodurch die Dokumentation unvollständig sein kann. Durch Umstellungen im Programmtext konnte man dieses Problem beheben.

---

<sup>1</sup> Dimitri van Heesch, <http://www.doxygen.org>

## 4.2 Ein generischer Ressourcenbaustein

---

Zusammenfassend hat sich die Verwendung von **Doxygen** jedoch positiv ausgewirkt. Die Dokumentation liegt im HTML-Format auf <http://www.odemx.de> vor. **ODEMx** kann an gleicher Stelle abgerufen werden.

---

## Literaturverzeichnis

- [Ale01] Alexandrescu, Andrei. **Modern C++ Design**. Addison-Wesley, 2001
- [Bie96] Bielig, Frank. **Prozess-orientierte Simulationssysteme in C++**. Humboldt-Universität zu Berlin, Februar 1996
- [Bir82] Birtwistle G., Luker P. **Discrete Event Simulation with DEMOS**. Proceedings of the 1982 Winter Simulation Conference, Highland, Chao, Madrigal, S. 682-691, 1982
- [C++98] ISO/IEC: 14882: 1998(E). **Programming Languages-C++**. ISO und ANSI C++ Standard, 1998
- [FA96] Fischer, Joachim; Ahrens, Klaus. **Objektorientierte Prozesssimulation in C++**. Addison-Wesley, 1996
- [Ger01] Gerstenberger, Ralf. **Visualisierung von Simulationsabläufen in ODEM**. Humboldt-Universität zu Berlin, 2001
- [Ger02] Gerstenberger, Ralf. <http://www.odemx.de>
- [Han90] Hansen. **The C++ Answer Book**. Addison-Wesley, 1990
- [Hel99] Helsgaun, Keld. **A Portable C++ Library for Coroutine Sequencing**. Roskilde University, 1999
- [Lec99] Lechler, Tim. **Entwurf und Implementierung eines Frameworks für diskrete Simulatoren in Java**. Universität Hamburg, 1999
- [Lie95] Liebl, Franz. **Simulation, Problemorientierte Einführung**. R. Oldenburg Verlag, 1995.
- [Pag91] Page Bernd. **Diskrete Simulation, Eine Einführung mit Modula-2**. Springer-Verlag, 1991
- [PSS98] Praehofer, Herbert; Sameting, Johannes; Stritzinger, Alois. **Using JavaBeans to teach simulation and using simulation to teach JavaBeans**. European Simulation Multiconference'98 Manchester UK, 1998
- [PSS99] Praehofer, Herbert; Sameting, Johannes; Stritzinger, Alois. **Discrete event simulation using the JavaBeans component model**. Proceedings of WEBSIM99, 1999
- [vLo92] von Löwis, Martin. **Porting the Coroutine Process Library**. Humboldt-Universität zu Berlin, 1992



## **Erklärung**

Hiermit erkläre ich, dass die vorliegende Arbeit von mir selbst und ohne jede unerlaubte Hilfe nur unter Verwendung der angegebenen Quellen angefertigt wurde.

Berlin, den 19. Februar 2003

Ralf Gerstenberger

## **Einverständniserklärung**

Hiermit erkläre ich mein Einverständnis, dass diese Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin öffentlich ausgelegt wird.

Berlin, den 19. Februar 2003

Ralf Gerstenberger