Projektarbeit

Entwicklung eines Werkzeugs zur Erfassung und Aufbereitung von Monitoring-Daten

Marc Müller

Betreuer: Joachim Thees

Kaiserslautern, März 1999

Fachbereich Informatik – AG Rechnernetze – Universität Kaiserslautern

Zusammenfassung

In dieser Arbeit wird die Entwicklung und Anwendung einer Implementierung von Software-Monitoring vorgestellt. Dabei wird sowohl konzeptionell beschrieben, wie die Monitoring-Daten erfasst und ausgewertet werden, als auch, woraus die entwickelte Lösung besteht und wie der Benutzer sie anwenden kann, um eine konkrete Auswertung vorzunehmen.

Notation

Im folgenden werden diese Hervorhebungen verwendet:

- *(Eigen-)Namen* werden kursiv geschrieben.
- Klassen- und Methodennamen werden in einer Sans Serif-Schrift gedruckt.
- `Ausgaben, Deklarationen und Dateinamen` werden in einer Typewriter-Schrift gesetzt.

Erstellt auf der Grundlage der neuen amtlichen Rechtschreibregeln mit LaTeX 2_ε.

Inhaltsverzeichnis

Kapitel 1

Monitoring

1.1 Einleitung

Formale Beschreibungstechniken („Formal Description Techniques“, FDT) ermöglichen die genaue Beschreibung von Systemen. *Estelle* ist eine solche FDT. Zur automatischen Implementierung einer Estelle-Spezifikation wurde in der AG Rechnernertze von Joachim Thees der *eXperimental Estelle Compiler (xec)* entwickelt, der aus einer Estelle-Spezifikation durch Generierung von C++-Code ein ablauffähiges Programm erzeugt (siehe auch [ThGo98] und [The98]).

Zur weiteren Verbesserung der eingegebenen Spezifikation sowie des Compilers selbst ist es nötig, zu wissen, was während der Ablaufzeit geschieht und wieviel Zeit für die einzelnen „Ausführungsteile“ (Verarbeitungszyklen, Transitionsauswahl etc.) benötigt wird.

Diese Aufgabe übernimmt das *(Software-)Monitoring*, mit dessen Hilfe man nach Ausführung der Spezifikation ermitteln kann, an welchen Stellen eine Optimierung, also eine Verbesserung der Spezifikation oder der automatischen Implementierung, nötig bzw. möglich ist. Dazu werden sog. *Zeitstempel* erzeugt, die angeben, welche Ereignisse zur Laufzeit aufgetreten sind (s. Kap. 1.3.1).

Ziel der Projektarbeit war es, ein Werkzeug zu entwickeln, das eine Verarbeitung und Auswertung der gewonnenen Zeitstempel zulässt. Zu diesem Zweck wurde eine Klassen-Bibliothek in C++ entwickelt, deren Funktionen im weiteren vorgestellt werden sollen (Kapitel 2). Desweiteren wird erläutert, wie die in der Bibliothek enthaltenen Funktionen genutzt werden können (Kapitel 3), sowie eine Erweiterungsmöglichkeit vorgestellt (Kapitel 4).

1.2 Estelle

Estelle [ISO97, DeBu89] ist eine seit 1989 standardisierte Formale Beschreibungstechnik, die für die Spezifikation von verteilten Systemen entwickelt wurde. *Estelle*-Spezifikationen beschreiben eine Menge von kommunizierenden Erweiterten Endlichen Automaten. Jeder dieser Automaten ist eine Instanz eines *Moduls*.

Module bilden die Grundstruktur einer *Estelle*-Spezifikation. Sie besitzen eine aus Interaktionspunkten und gemeinsamen Variablen bestehende externe Schnittstelle. Intern sind sie aufgebaut aus ihrer lokalen Beschreibung (u.a. bestehend aus Kontextvariablen, einem Kontrollzustand und lokalen *Transitionen*) sowie einer Menge von Kindmodulen, wodurch sich eine (statische) hierarchische Struktur ergibt (s. Abb. 1.1). Während des Ablaufs einer *Estelle*-Spezifikation kann jede Instanz eines Moduls neue Instanzen seiner Kind-Module erzeugen oder beenden, wodurch eine dynamische Hierarchie entsteht. Zusätzlich kontrolliert jede Modulinstanz dynamisch die Kommunikationskanäle, über die sie selbst sowie die jeweiligen Instanzen der Kind-Module durch asynchronen Nachrichtenaustausch über die Interaktionspunkte kommunizieren. Es ergibt sich also eine dynamische Verbindungsstruktur.

Transitionen beschreiben schließlich das Verhalten der Module. Jede Transition besteht aus

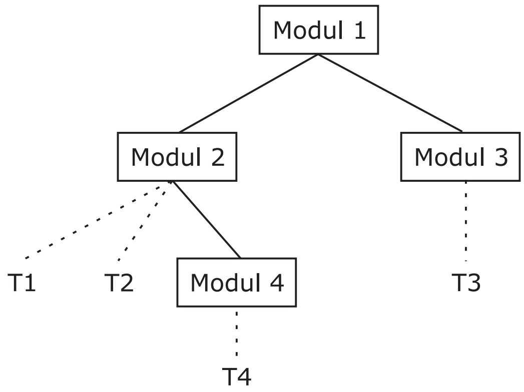

Abbildung 1.1: Beispel einer *Estelle*-Struktur

einer Zustandsübergangs-/Ausgaberelation, die den Anfangszustand, die Eingabe, den Folgezustand und die Ausgabe beschreibt.

Die Ausführung einer *Estelle*-Spezifikation besteht nun im Wesentlichen aus der indeterministischen Auswahl einer Menge von feuerbereiten Transitionen und deren (atomarem) Feuern.

1.3 Monitoring allgemein

Monitoring [Lan92] dient zur Ermittlung der Optimierungsmöglichkeiten einer Implementierung und gliedert sich prinzipiell in zwei Teile: die *Erfassung* von Monitoring-Daten (Zeitstempeln) und deren *Auswertung*.

1.3.1 Die Erfassung von Zeitstempeln

Zeitstempel sind die Grundlage des Monitoring. Sie geben Auskunft darüber, *wann welches* Ereignis *wo* stattgefunden hat. Als Beispiel betrachte man Abbildung 1.2, die einen Kontrollflussgraphen mit drei Messpunkten zeigt. Passiert der Kontrollfluss einen der Messpunkte, so wird dies als ein sog. *Ereignis* festgehalten („was“-Teil), ausserdem wird die aktuelle Zeit in den Zeitstempel eingetragen („wann“-Teil).

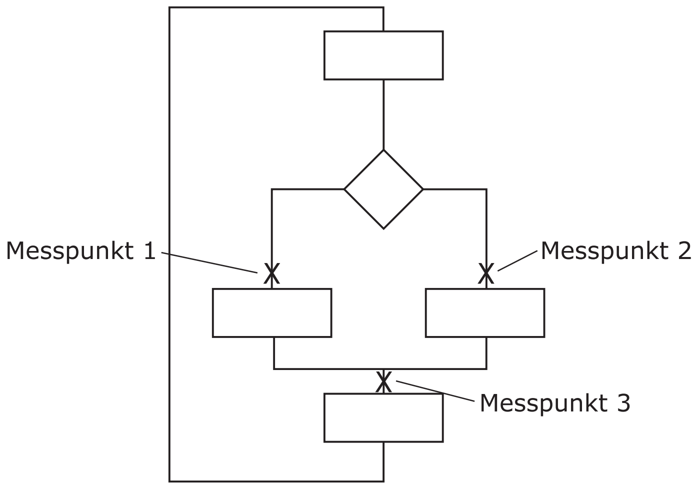

Abbildung 1.2: Messmethode / Ereignisse

Ein Beispiel zur Erfassung von Zeitstempeln zeigt Abbildung 1.3. Zur Laufzeit werden die dargestellten Zeitstempel an den drei Messpunkten erzeugt. Schon anhand des „was"-Teils dieser Tabelle ist im Nachhinein zu erkennen, in welcher Reihenfolge die Messpunkte durchlaufen wurden. Wertet man zusätzlich die „wann"-Spalte aus, so ist die Zeitdifferenz zwischen den einzelnen Ereignissen ermittelbar, und damit auch, ob etwa zum Durchlaufen eines Teils wesentlich mehr Zeit benötigt wurde, als zum Durchlaufen eines anderen Teils. Da der Benutzer (als Autor der zugrunde liegenden Spezifikation) weiß, wie sich die Spezifikation beim Ablauf verhalten *sollte*, kann er bereits aus einer solchen einfachen Tabelle von Zeitstempeln Informationen über Optimierungsmöglichkeiten erkennen.

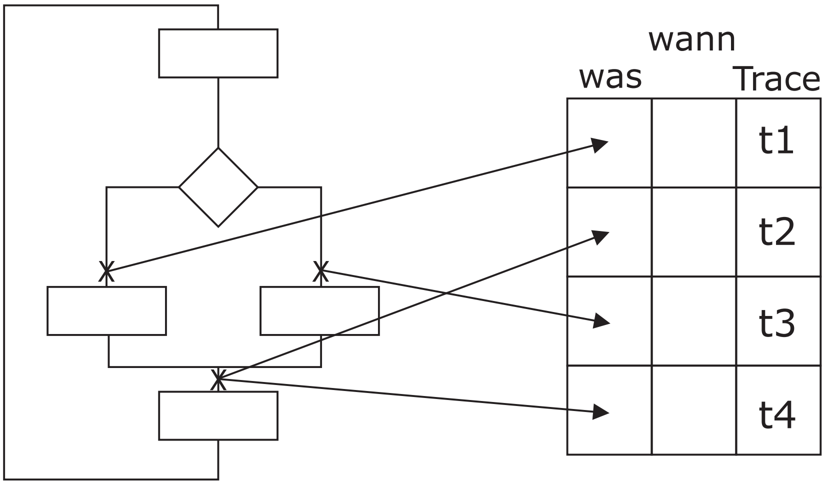

Abbildung 1.3: Erfassung von Zeitstempeln

Angewandt auf *Estelle*-Spezifikationen wird der „was"-Teil um einen „wo"-Teil ergänzt. Die beiden Teile haben dabei folgende Aufgaben: Mit „was" ist der *Typ* des Ereignisses gemeint, also z.B. der Start des Feuerns einer Transistion oder Beginn eines neuen Auswertezyklusses. Dieser Teil beschreibt also das eingetretene Ereignis. Im „wo"-Teil wird angegeben, in welchem Modul bzw. welcher Transistion das Ereignis aufgetreten ist.

Ein Zeitstempel beinhaltet also folgende Informationen:

- „was"-Teil: Typ des Ereignisses
- „wann"-Teil: Zeitpunkt, an dem das Ereignis auftrat
- „wo"-Teil: Ort, an dem das Ereignis auftrat

Die Zeitstempel werden bei der Ausführung der Spezifikation ersteinmal nur erzeugt und gesammelt. Um den zeitlichen Aufwand zu minimieren und die Verfälschung der gemessenen Zeiten so gering wie möglich zu halten, findet eine weitere Verarbeitung zur Laufzeit nicht statt. Die eigentliche Auswertung erfolgt erst nach Ende des Ablaufs.

1.3.2 Die Auswertung von Zeitstempeln

Welche Informationen der Benutzer allein anhand der Zeitstempel erhalten kann, wurde im vorherigen Abschnitt erläutert. Zu beachten ist allerdings die enorme Anzahl von Zeitstempeln, die generiert werden. So können einfache Monitoring-Läufe bereits Hunderttausende oder Millionen von Zeitstempeln umfassen. Diese Menge von (ungeordneten) Zeitstempeln ist für den Benutzer nicht ohne Hilfsmittel überschaubar und damit wenig aussagekräftig.

Daher werden aus den Zeitstempeln ersteinmal Zeitspannen generiert, die einem Ereignis einen Start- und Stoppzeitpunkt zuordnen, woraus sich die zeitliche Länge dieses Ereignisses ergibt. Zeitspannen sind daher für den Benutzer oft aussagekräftiger. Allerdings ist die Problematik der Überschaubarkeit bei der Fülle der Informationen damit nicht gelöst.

Daher ist ein Auswertewerkzeug nötig, das es dem Benutzer erlaubt, die Menge von Zeitspannen nach seinen Bedürfnissen auszuwerten. Wünschenswert ist z.B. eine Funktion zum Herausfiltern und statistischen Auswerten von Zeitspannen nach bestimmten, vom Benutzer festgelegten Kriterien. So ist es denkbar, alle Spannen eines bestimmten Typs (z.B. alle Zykluszeiten) herauszufiltern und alle restlichen Spannen nicht zu beachten. Oder es sind nur alle Spannen eines bestimmten Moduls von Interesse, während alle übrigen Spannen nicht interessieren. Weiterhin ist u.a. die Summierung der Längen aller gefundenen Spannen sowie die gezielte Suche nach einzelnen Spannen wünschenswert.

1.4 Vorhandene Monitoring-Lösungen

1.4.1 PATO

In der AG Rechnernetze wurde bereits mit PATO („Performance Analysis TOol“) [Pen96] eine Monitoring-Lösung realisiert. PATO wurde ursprünglich für DINGO [SiSt93] entwickelt, einen älteren Estelle-Compiler, der wesentlich langsameren Code erzeugt. Teil von PATO ist auch ein grafisches Auswertewerkzeug, das die Auflistung der generierten Zeitspannen sowie deren statistische Auswertung ermöglicht (s. Abb. 1.4).

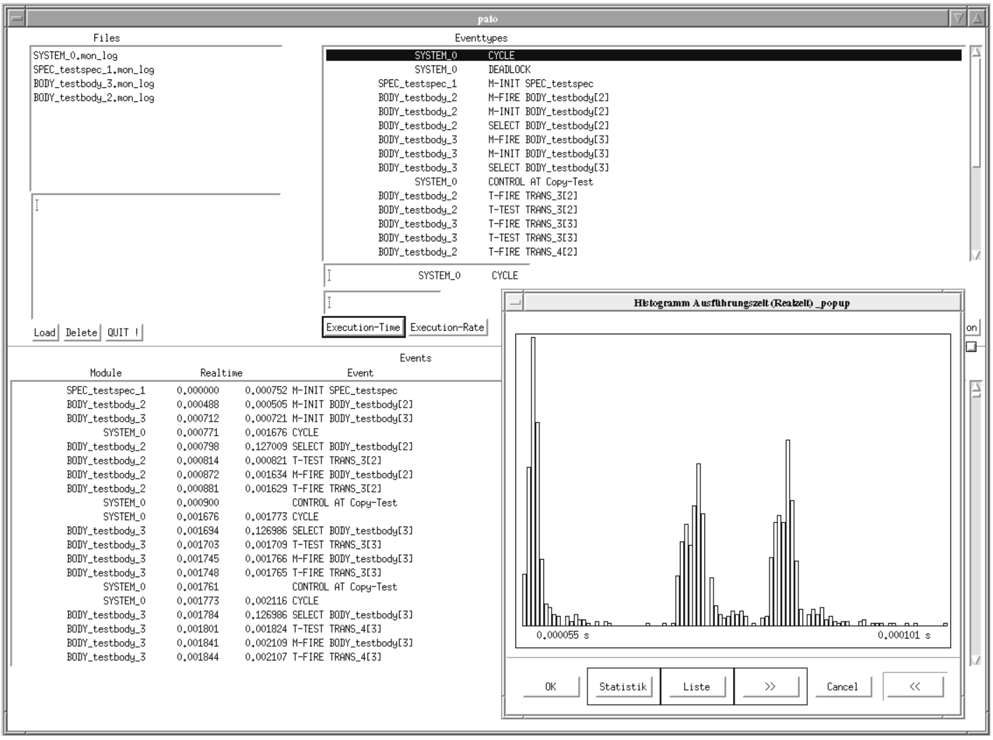

Abbildung 1.4: Das grafische Auswertewerkzeug von PATO

PATO generiert Zeitspannen bereits zur Laufzeit der Implementierung. Durch den so entstehenden Mess-Overhead im Verhältnis zur Schnelligkeit der vom *xec* erzeugten Implementierung werden die gemessenen Zeiten jedoch zunehmend verfälscht. Die Messmethode in PATO ist also nicht effizient genug, um ausreichend genaue Messwerte zu erhalten.

Hinzu kommt, dass vielfach Ereignisse mehrfach gemessen werden (s. Abb. 1.5). Der Messpunkt soll dabei aussagen, dass der Kontrollfluss das Modul A passiert hat und nun in Modul B eintritt. Dazu werden jedoch zwei Messungen vorgenommen, einmal, nachdem Modul A passiert wurde, und einmal, bevor der Kontrollfluss in Modul B eintritt. Es werden also für einen Messpunkt zwei Messungen vorgenommen.

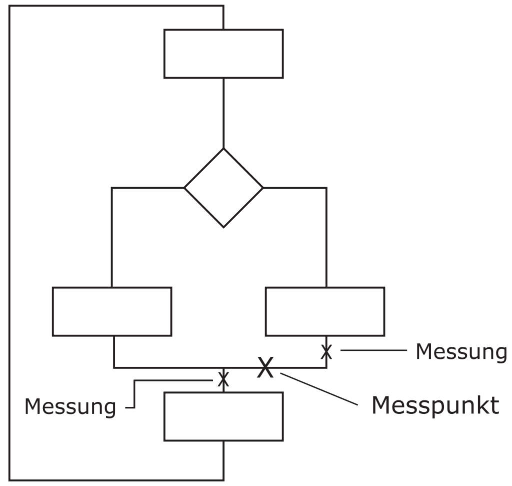

Abbildung 1.5: Messmethode in PATO

Bei der Anwendung auf *Estelle*-Spezifikationen treten solche Messungen zum Beispiel bei Beginn eines neuen Auswertezyklusses auf. Dort wird zuerst die laufende Zeitspanne für den Zyklus beendet (also die aktuelle Zeit als Stoppzeit erfasst), eine neue Zeitspanne erzeugt und dann erneut die aktuelle Zeit als Startzeit der neuen Spanne eingetragen. Obwohl es sich also nur um *einen* Messpunkt handelt, werden *zwei* Messungen vorgenommen. Die neue Monitoring-Lösung generiert beim Start eines neuen Auswertezyklusses nur einen Stempel, der zugleich als Stoppzeit für den alten und als Startzeit für den neuen Zyklus dient. Somit wird der Overhead beim Messen verringert.

Neben der fehlenden Effizienz ist PATO nicht flexibel genug. Soll z.B. ein neuer Messpunkt eingeführt werden, so ist dies nur sehr umständlich durch aufwendige Modifizierungen am PATO-Code möglich. Auch das grafische Auswertewerkzeug von PATO bietet dem Benutzer nicht genügend Flexibilität. Das Hinzufügen einer neuen Auswertemethode ist für den Benutzer nicht oder nur durch umfangreiche Modifizierungen am Code des Werkzeugs möglich.

Um das grafische Auswertewerkzeug jedoch weiter nutzen zu können, unterstützt die entwickelte Bibliothek die Konvertierung der Messdaten in das PATO-Format, wodurch PATO auch weiterhin zur Auswertung der gewonnenen Daten verwendet werden kann. Dazu ist jedoch eine erweiterte Version von PATO nötig, da durch die neue Monitoring-Lösung eine größere Menge unterschiedlicher Typen von Zeitspannen erzeugt wird, als der ursprüngliche PATO unterstützte. Diese Modifizierungen sowie einige Umstrukturierungen im PATO-Code wurden ebenfalls im Rahmen der Projektarbeit durchgeführt.

1.4.2 Generierung der Monitoring-Daten im *xec*

Ebenfalls bereits vorhanden war die Gernerierung der Zeitstempel durch den *xec* beim Ablauf der Spezifikation. Wie bereits oben beschrieben, wird durch die Verwendung von Zeitstempeln statt Zeitspannen sowie der Vermeidung mehrfacher Messungen eines Messpunktes versucht, den Mess-Aufwand zu minimieren.

Damit die generierten Stempel eindeutig dem Modul zugeordnet werden können, bei dem das zugrunde liegende Ereignis aufgetreten ist, werden alle Module und Transitionen beginnend mit Null aufsteigend durchnummeriert. Da sich diese Nummerierung auf die statische Modul-Hierarchie bezieht, wird die jeweilige Nummer als *Static ID* bezeichnet.

Wie bereits oben beschrieben, können Module in *Estelle* jedoch mehrfach instanziiert werden, daher reicht eine solche statische Nummerierung für eine genaue Zuordnung des Ereignisses zur Modulinstanz nicht aus. Aus diesem Grund werden auch alle Modulinstanzen nummeriert. Diese Modulinstanz-Nummer wird als *Dynamic ID* bezeichnet. Beide Identifikationen bilden den „wo“-Teil der generierten Zeitstempel.

Wie oben breits erläutert, beschreibt der „was"-Teil eines Stempels den Typ des Ereignisses genauer, das aufgetreten ist. Tabelle 1.1 zeigt, welche Typen der *xec* generieren kann.

Typ des Zeitstempels	*Erläuterung*
SysCycle	Start eines neuen Auswertezyklusses
SysDeadlock	Deadlock
SysSleepStart	Beginn Wartezustand
SysSleepAbort	Wartezustand abgebrochen
ModSelectLocal	Start lokale Auswahlzeit
ModSelectChild	Start Auswahlzeit bei Kindmodulen
ModSelectEnd	Ende Auswahlzeit
ModFireStart	Start des Feuerns eines Moduls
ModFireEnd	Ende des Feuerns eines Moduls
ModInitStart	Beginn Modulinitialisierung
ModInitEnd	Ende Modulinitialisierung
TraFireStart	Beginn des Feuerns einer Transition
TraFireEnd	Ende des Feuerns einer Transition
TraTestStart	Start Transitionstest
TraTestEnd	Ende Transitionstest
ExplicitTrace	manuell generierter Trace

Tabelle 1.1: Vom *xec* generierbare Zeitstempel-Typen

Generierte Zeitstempel werden zur Laufzeit in einem Ringpuffer gespeichert, so dass also nur eine bestimmte Anzahl von Stempeln gespeichert werden können. Diese Anzahl kann vom Benutzer vorgegeben werden.

Kapitel 2

Die Klassenbibliothek *monitorlib*

2.1 Grundlagen

In Kapitel 1 wurde das Prinzip von Software-Monitoring erläutert, sowie, welche Anforderungen an die Auswertung gestellt werden. Zur Erfüllung dieser Anforderungen wurde die Klassenbibliothek *monitorlib* in C++ entwickelt, die die Auswertung von Monitoringdaten ermöglicht.

Diese neue Monitoring-Lösung ist sowohl effizient als auch flexibel: Effizient, da zur Laufzeit lediglich Zeitstempel erzeugt werden, was nur einen minimalen Aufwand erzeugt. Durch den geringeren Mess-Overhead werden die gemessenen Zeiten weniger verfälscht, ausserdem werden Ereignisse nicht (wie bei PATO) mehrfach gemessen. Die Klassenbibliothek ist flexibel, da der Benutzer sich die gewünschte Auswertung selbst programmiert, also nicht auf vorgegebene, unveränderliche Auswertemuster angewiesen ist. Da der Benutzer jedoch auf einen hohen Abstraktionsebene arbeitet, ist die Programmierung von Auswertetools trotzdem mit geringem Aufwand möglich.

2.2 Struktur

Die *monitorlib* hat grundsätzlich folgende Struktur (s. Abb. 2.1): In der Klasse monitorlib sind all diejenigen Funktionen realisiert, die nur auf den internen Datenstrukturen dieser Klasse arbeiten. Die Funktionen sind somit völlig unabhängig vom *xec* und können auch für die Auswertung von Zeitspannen verwendet werden, die aus einer völlig anderen Quelle stammen. Vorraussetzung ist, dass eine Funktion entwickelt wird, die die Zeitstempel (oder etwas vergleichbares) in Zeitspannen umwandelt und diese in die internen Datenstrukturen der monitorlib einträgt.

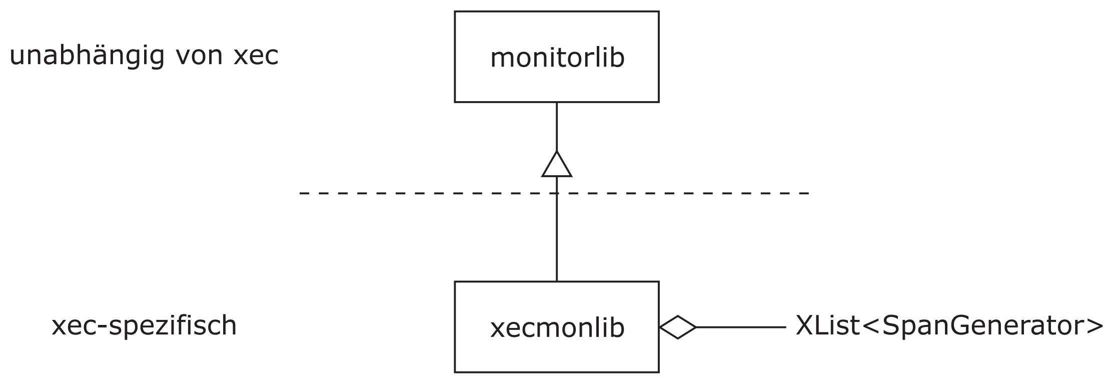

Abbildung 2.1: Struktur der *monitorlib*

In dieser Klasse befinden sich dementsprechend alle Funktionen zur Auswertung der Zeitspannen, also z.B. Funktionen zum Suchen, Auswählen (Select) und Summieren.

Funktionen, die spezifisch für die Strukturen des *xec* sind, sind in der Klasse xecmonlib realisiert. Diese Methoden benötigen Wissen darüber, wie ein *xec*-Monitor-Logfile geöffnet und gelesen wird, wie man auf die einzelnen Datenfelder zugreift und welche Informationen in welchem Format dort gespeichert sind. Die Generierung der Zeitspannen ist z.B. eine solche Funktion, die „wissen muss", wie man einen Zeitstempel einliest und welche Typen von Stempeln existieren. Auch weitere *xec*-spezifische Hilfsfunktionen finden sich hier.

Um die Änderung bzw. Erweiterung des Zeitspannen-Generators einfach nachvollziehbar zu gestalten, wurde der Generator ebenfalls in der eigenen Klasse SpanGenerator realisiert. Diese Klasse besteht aus mehreren Unterklassen, je eine pro generierbarem Zeitspannen-Typ (s. Abb. 2.2). Die Generierung von Zeitspannen läuft prinzipiell so ab: Zuerst wird von jeder dieser Unterklassen ein Objekt angelegt und in eine Liste eingetragen, die in der xecmonlib-Klasse verwaltet wird. Sollen nun Zeitspannen generiert werden, wird der erste Zeitstempel aus dem Monitor-Logfile des *xec* gelesen und dem ersten Objekt in der Liste übergeben. Dieses Objekt entscheidet nun, ob es den Stempel als Startstempel der „eigenen" Zeitspanne erkennt (also der Stempel zu dem Objekt „passt") oder nicht. Wird der Stempel erkannt, so wird der entsprechende Stoppstempel gesucht und die Zeitspanne oder ansonsten ein Nullzeiger zurückgegeben. Danach wird der Stempel an das nächste Objekt der Liste übergeben.

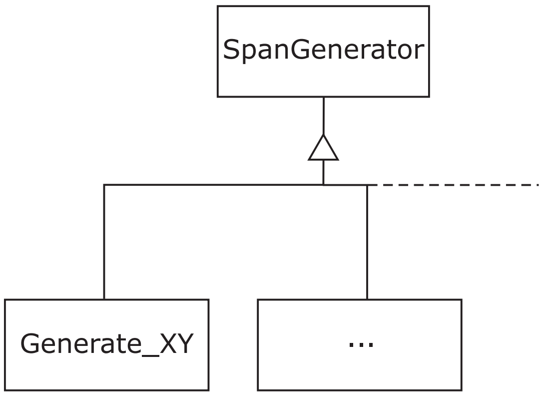

Abbildung 2.2: Struktur der SpanGenerator-Klasse

Durch diese objektorientierte Struktur ist das Hinzufügen von neuen Typen von Zeitspannen einfach realisierbar. Es muss lediglich eine neue Unterklasse von SpanGenerator angelegt, davon ein Objekt erzeugt und dieses in die Liste aller Generatoren eingetragen werden (s. auch Kap 2.5.1).

Welcher Zeitspannen-Typ aus welchen Typen von Zeitstempeln generiert werden, ist aus Tabelle 2.1 zu entnehmen.

2.3 Die Funktionen: **monitorlib**

Im Folgenden werden die Methoden der Klasse monitorlib genau vorgestellt. Zu jeder Methode werden Aufrufe, Rückgabewerte sowie vor allem ihr Zweck erläutert. Das Ziel ist es, einen Benutzer in die Lage zu versetzen, durch Benutzung der monitorlib-Methoden sowie der Methoden der anderen Klassen auf einfache Weise eigene Auswerteprogramme zu erstellen.

Typ der Zeitspanne		*Typen der Zeitstempel*	
Funktion	*Name*	*Start*	*Stopp*
Auswertezyklus	SysCycle	SysCycleStart	SysCycleStart
Feuern eines Moduls	ModFire	ModFireStart	ModFireEnd
Feuern einer Transition	TraFire	TraFireStart	TraFireEnd
Initialisierung eines Moduls	ModInit	ModInitStart	ModInitEnd
lokale Auswahlzeit	ModSelectLocal	ModSelectLocal	ModSelectChild
Auswahlzeit Kindmodule	ModSelectChild	ModSelectChild	ModSelectEnd
Deadlock	SysDeadlock	SysDeadlock	-
Wartezeit	SysSleep	SysSleepStart	SysSleepAbort oder nächster Stamp
Transitions-Test	TraTest	TraTestStart	TraTestEnd
manueller Trace	ExplicitTrace	ExplicitTrace	-

Tabelle 2.1: Zuordnung Zeitspannen- zu Zeitstempel-Typen

2.3.1 Search4Span

Deklaration:

```
MonTimeSpan *Search4Span(int nType,
                         int nStaticId,
                         unsigned long ulDynamicId,
                         unsigned uBegin,
                         bool bForward,
                         XPtrList<MonTimeSpan> *TimeSpanList);
```

Zweck:

Die Methode `Search4Span` sucht nach einer Zeitspanne anhand der übergebenen Kriterien und gibt bei Erfolg die gefundene Spanne, ansonsten `NULL` zurück.

Mit den ersten drei Argumenten werden die Kriterien festgelegt, nach denen die Spanne gesucht werden soll (Typ, statische, dynamische ID – s. Kap. 1.3.1 und 1.4.2). Wird einer oder mehrerer dieser Werte als `ANY` (bibliotheksweit bekannte Konstante) übergeben, so wird das entsprechende Kriterium nicht beachtet (so ist z.B. die Suche nach einer Spanne eines bestimmten Typs unabhängig vom Modul oder die Suche nach der nächsten Spanne eines bestimmten Moduls möglich). Das vierte Argument (`uBegin`) bezeichnet den Startpunkt, bei dem mit der Suche begonnen werden soll (der übergebene Wert entspricht der Nummer eines Spans, s. Kap. 2.5.1), das fünfte Argument (`bForward`) gibt an, ob ab dieser Position vorwärts (true) oder rückwärts (false) gesucht werden soll. Das letzte Argument (`TimeSpanList`) ist ein Zeiger auf die Liste mit Zeitspannen, die durchsucht wird.

Beispiel:

```
MonTimeSpan *Zeitspanne = Search4Span(MonFire, -1 -1, 1, true,
                          &SpanList);
```

Sucht nach der nächsten Zeitspanne, die das Feuern eines Moduls[1] angibt, unabhängig vom Modul, dass diese erzeugt hat.

[1]Mit dem „Feuern eines Moduls“ ist gemeint, dass eine Transition dieses Moduls feuert.

```
MonTimeSpan *Zeitspanne = Search4Span(-1, 5, 2, 42, false,
                          &SpanList);
```

Sucht rückwärts nach der nächsten Zeitspanne des Moduls mit der statischen ID fünf und der dynamischen ID zwei ab Spanne Nummer 42 .

2.3.2 Select

Deklaration:

```
void Select(int nType,
            int nStaticId,
            unsigned long ulDynamicId,
            unsigned uBegin,
            bool bForward,
            XPtrList<MonTimeSpan> *SourceList,
            XPtrList<MonTimeSpan> *DestList);
```

Zweck:

Select filtert aus der Menge der Zeitspannen diejenigen heraus, die die übergebenen Kriterien erfüllen. Die ersten fünf Argumente entsprechen denen der Search4Span-Methode. Die letzten beiden Argumente geben die Quell- und Zielliste an, von der die Zeitspannen gelesen bzw. in die die passenden Spannen geschrieben werden sollen. Die Zeitspannen werden dabei nicht verschoben, sondern kopiert; sie sind in der Quellliste auch dann noch vorhanden, wenn sie in die Zielliste eingetragen wurden.

Sowohl bei der Quell- als auch bei der Zielliste denselben Zeiger auf eine Liste zu übergeben, ist nicht sinnvoll. Falls Zeitspannen gefunden werden, die die übergebenen Kriterien erfüllen, kann dies unter Umständen zu einer Endlosschleife führen.

Beispiel:

```
Select(TraFire, -1, -1, 1, true, &qlist, &zlist);
```

Kopiert alle Spans mit dem Typ „TraFire“ in die Zielliste, unabhängig vom Modul.

```
Select(ModSelectLocal, 2, 1, 1, true, &qlist, &zlist);
```

Kopiert alle Spans, die die lokale Auswahlzeit der ersten Instanz des Moduls zwei angeben.

2.3.3 SelectNeg

Deklaration:

```
void SelectNeg(int nType,
               int nStaticId,
               unsigned long ulDynamicId,
               unsigned uBegin,
               bool bForward,
               XPtrList<MonTimeSpan> *SourceList,
               XPtrList<MonTimeSpan> *DestList);
```

Zweck:

Die SelectNeg-Methode filtert analog zur Select-Methode ebenfalls Zeitspannen, daher sind alle Argumente identisch. Der Unterschied besteht darin, dass SelectNeg diejenigen Zeitspannen kopiert, die die übergebenen Kriterien *nicht* erfüllen. Hierdurch ist es möglich, gezielt bestimmte Zeitspannen zu „verwerfen".[2]

Durch die Select- und SelectNeg-Methoden ist es also möglich, aus der Gesamtzahl der erzeugten Zeitspannen diejenigen herauszunehmen, die für die eigene Auswertung von Interesse sind.

Beispiel:

analog zu Select

2.3.4 Summarize

Deklaration:

```
TIME Summarize(XPtrList<MonTimeSpan> *SpanList);
```

Zweck:

Die Summarize-Methode dient zum Summieren der Längen einer Liste von Zeitspannen. Ein Zeiger auf eine solche Liste wird als Argument übergeben.

Diese Methode ist z.B. einsetzbar, um zu berechnen, wieviel Zeit während der Ausführung für eine bestimmte Aktivität aufgewandt wurde. So könnte man mittels der Select-Methode aus der Gesamtmenge der Zeitspannen diejenigen herausfiltern, die die lokale Auswahlzeit (ModSelectLocal) angeben, und diese anschließend mittels der Summarize-Methode summieren. Das Ergebnis wäre die Zeit, die während des gesamten Ablaufs für die lokale Auswahl von Transitionen aufgewandt wurde.

In ähnlicher Art und Weise ist die Summarize-Methode auch bei anderen Auswertungen einsetzbar.

Zur Speicherung von Zeiten (z.B. der Zeiten der Zeitstempel, Start- und Stoppzeit der Zeitspannen etc.) ist der Typ `TIME` definiert, der auch im folgenden Beispiel verwendet wird.

Beispiel:

```
TIME Auswahlzeit = Summarize(&SysCycleSpans);
```

2.3.5 IsChild

Deklaration:

```
bool IsChild(unsigned uChild,
             unsigned uParent,
             bool bDirect=false);
```

Zweck:

Die IsChild-Methode überprüft, ob das Modul mit der statischen ID `uChild` (erstes Argument) ein Kind des Moduls mit der statischen ID `uParent` (zweites Argument) ist. Als drittes Argument kann zusätzlich angegeben werden, ob nur eine Verwandschaft ersten Grades (also ein „direktes Kind") die Prüfung erfüllen kann (true),

[2]Das Löschen von Zeitspannen aus einer Liste ist mit Methoden der *monitorlib* nicht möglich, auch nicht, wenn z.B. bei der SelectNeg-Methode als Quell- und Zielliste Zeiger auf dieselbe Liste übergeben werden. Im Gegenteil: Dies würde zu einer Endlosschleife und einer unendlich langen Liste führen!

oder, ob auch ein Kind eines Kindmoduls usw. als „Kind“ des Elternmoduls gilt (Transitivität; false; standardmäßig).

Mit Hilfe dieser Methode sind Auswertungen anhand von verwandschaftsmäßigen Abhängigkeiten möglich. So sind hierdurch z.B. alle Auswertungen auf die Kindmodule eines bestimmten Moduls einschränkbar.

Voraussetzung für die Ausführung dieser Methode ist die Generierung des Verwandschaftsbaumes der Spezifikation. Dies geschieht durch den einmaligen Aufruf der Methode CreateParentChildStructure der xecmonlib-Klasse (s. Kap. 2.4.5).

Beispiel:

```
bool bKind = IsChild(6, 2, true);
```

Ist Modul sechs ein direktes Kind von Modul zwei?

2.4 Die Funktionen: xecmonlib

Als nächstes folgen die Methoden der Klasse xecmonlib. Diese Methoden benutzen Wissen über den Aufbau von *xec*-Monitor-Logfiles und den Zugriff darauf und sind dementsprechend *xec*-spezifisch.

2.4.1 ReadLog

Deklaration:

```
void ReadLog(char *pszFilename,
             bool bCalibrate=true);
```

Zweck:

Mittels dieser Methode wird ein Monitor-Logfile geöffnet und eingelesen. Auf die in dem File enthaltenen Zeit*stempel* kann dann im Folgenden zugegriffen werden.

Das erste Argument ist ein Zeiger auf den Filenamen der zu lesenden Datei. Das zweite Argument ist optional und gibt an, ob die Zeitstempel kalibriert werden sollen (standardmäßig: ja) oder nicht. Kalibrieren heisst, dass versucht wird, die Zeit, die nötig ist, die Zeitstempel zu generieren, aus den gemessenen Zeiten „herauszurechnen“. Im Idealfall erhält man so die Zeiten, die sich ergeben hätten, wenn keine Stempel erzeugt worden wären, also der zusätzliche Aufwand zum Erzeugen der Stempel nicht existiert hätte. Normalerweise ist eine Kalibrierung notwendig, um die Zeiten so genau wie möglich zu halten; falls jedoch einmal mit den „Originalzeiten“ der Datei gerechnet werden soll, ist dies durch dieses Argument auch möglich.

Die ReadLog-Methode ist dient nur dazu, die Zeitstempel der *xec*-Datei für die weitere Verarbeitung zugänglich zu machen. Durch diesen Methoden-Aufruf wurden noch keine Zeitspannen erzeugt, die die Grundlage für viele der weiteren Auswertefunktionen sind, insbesondere solche, die in der monitorlib-Klasse realisiert sind.

2.4.2 CreateSpans

Deklaration:

```
void CreateSpans(XPtrList<MonTimeSpan> *TimeSpanList);
```

Zweck:

CreateSpans ist eine zentrale Methode der *monitorlib*. Durch sie werden aus den Zeitstempeln Zeitspannen generiert, auf denen dann die meisten anderen Methoden aufbauen. Als Argument wird ihr ein Zeiger auf eine Liste von Zeitspannen übergeben (die Liste muss also bereits angelegt sein und sollte normalerweise keine Elemente enthalten).

Die Generierung geschieht im Wesentlichen dadurch, dass eine Liste von Objekten von Zeitspannen-Generatoren erzeugt wird (dies erfolgt nicht hier, sondern im Konstruktor der SpanGenerator-Klasse) und dann die Zeitstempel nacheinander jedem Element dieser Liste übergeben werden, wobei die Objekte prüfen, ob der Stempel Startpunkt der eigenen Zeitspanne ist. Durch die Verwendung dieser Objekt-Liste ist das Hinzufügen neuer Typen von Zeitspannen bzw. deren Generierung leicht möglich. Ausserdem bleibt jede der betroffenen Methoden klein und überschaubar.

Um später mit der Methode SavePato Daten im PATO-Format speichern zu können, muss vor dem CreateSpans-Aufruf die CreateDB-Methode aufgerufen werden. Insofern dies geschehen ist, werden zusätzlich zu den Zeitspannen auch die PATO-Keys erzeugt, die zur Generierung eines Logfiles im PATO-Format notwendig sind.

Beispiel:

```
CreateSpans(&SpannenListe);
```

2.4.3 CreateDB

Deklaration:

```
void CreateDB();
```

Zweck:

CreateDB erzeugt die zur Generierung eines PATO-Logfiles mittels SavePato notwendigen PATO-Keys aus den Zeitstempeln. Die PATO-Keys geben bei PATO den Typ einer Zeitspanne an. Zusätzlich wird einem Key jeweils ein Name (z.B. der Modulname) zugeordnet, der dann im grafischen Auswertewerkzeug angezeigt wird. Die Keys werden in einer separaten Datei aufgelistet (wird beim SavePato-Aufruf erzeugt), aus der PATO die Keys dann wieder liest.

Es werden höchstwahrscheinlich mehr Keys erzeugt als später im PATO-Logfile tatsächlich benötigt werden. Dies kommt daher, dass Keys für sämtliche möglichen Span-Typen erzeugt werden. So werden beispielsweise für jedes Modul drei Keys generiert: einer für das Feuern, einer für die Initialisierung und einer für die Auswahl des Moduls. Die geschieht unabhängig davon, ob die entsprechenden Typen im Logfile überhaupt vorkommen. Da jedoch in PATO nur diejenigen Typen angezeigt werden, die im Logfile auch verwendet werden, ist eine genauere Überprüfung hier nicht notwendig.

Der Aufruf von CreateDB ist vor dem Aufruf von CreateSpans nötig, da dort die Keys bereits den generierten Zeitspannen zugeordnet werden.

Beispiel:

```
CreateDB();
```

2.4.4 SavePato

Deklaration:

```
void SavePato(char *pszPatolog,
              XPtrList<MonTimeSpan> *TimeSpanlist);
```

Zweck:

SavePato dient zur Erzeugung eines Logfiles im PATO-Format. Übergeben werden als erstes Argument ein Filenamen-Präfix sowie, als zweites Argument, die Liste von Zeitspannen, aus denen das PATO-Logfile erstellt werden soll.

Vor Aufruf dieser Methode ist ein Aufruf von CreateDB und CreateSpans mit *derselben Zeitspannen-Liste* nötig!

Da die PATO-Keys bereits erzeugt (CreateDB) und den erzeugten Zeitspannen zugeordnet (CreateSpans) wurden, brauchen hier nur noch die Zeitspannen aus der Liste in das PATO-Format gebracht und in eine Datei gespeichert, sowie das PATO-Datenbank-File für die Keys erstellt zu werden.

Der übergebene Filenamen-Präfix hat dabei folgende Bedeutung: Erstellt wird die Datenbank-Datei mit dem Namen `<Präfix>pato.mon_db`, sowie jeweils eine Datei pro Modul mit dem Namen `<Präfix><Modulname>_<Instanznummer>.mon_log`. Der Präfix kann also auch ein leerer String sein.

Beispiel:

```
SavePato("Bsp1", &SpanList);
```

2.4.5 CreateParentChildStructure

Deklaration:

```
void CreateParentChildStructure();
```

Zweck:

Diese Methode generiert den Verwandschaftsbaum, der aussagt, welche Module Kinder welcher Module sind. Dieser Baum wird für die Methode IsChild der Klasse monitorlib benötigt. Demzufolge muss vor der ersten Benutzung von IsChild einmal diese Methode aufgerufen werden.

Der (interne) Baum besteht aus Knoten, die jeweils die eigene statische ID, einen Zeiger auf ein Feld von Zeigern auf die eigenen Kinder sowie deren Anzahl enthalten (s. Abb. 2.3). Die Erstellung des Baumes erfolgt rekursiv. Dabei wird zuerst überprüft, welche Module das SYSTEM-Modul (also das Modul, von dem alle anderen abstammen) als Vatermodul angeben. Für jedes der gefundenen Kinder wiederholt sich diese Suche mit dem jeweiligen Kind als Ausgangspunkt. Durch das Eintragen der jeweils gefundenen Informationen (Anzahl der Kinder und deren statische ID) in neu angelegte Baumknoten, sowie das Einbinden dieser neuen Knoten in den Baum (durch Anpassen der entsprechenden Verzeigerung), entsteht der Verwandschaftsbaum.

Durch die Bereitstellung der Methode IsChild sollte ein Zugriff auf diese interne Datenstruktur für den Benutzer normalerweise nicht notwendig sein. Da die Struktur jedoch recht einfach ist, sollte ihre Nutzung aber auch mit geringem Aufwand realisierbar sein, falls dies für spezielle Auswertungen benötigt wird.

Beispiel:

```
CreateParentChildStructure();
```

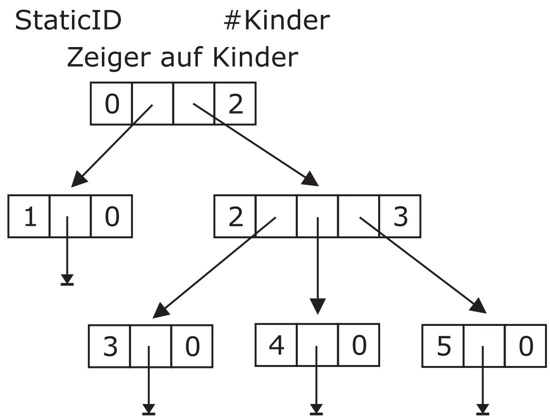

Abbildung 2.3: Struktur des Verwandschaftsbaumes

2.5 Die Funktionen: SpanGenerator

Nun folgen die Funktionen der SpanGenerator-Klasse, die für die Erzeugung von Zeitspannen aus Zeitstempeln verantwortlich ist.

Zu beachten ist, dass im Standardkonstruktor der Klasse die Objekt-Liste der Klasse xecmonlib gefüllt wird. Werden also weitere Unterklassen erstellt, um neue Typen von Zeitspannen zu erzeugen, so muss für jede solche Unterklasse ein Objekt erzeugt und dieses in die Liste eingetragen werden. Zur eigentlichen Erzeugung einer Zeitspanne dient die Methode For.

2.5.1 For

Deklaration:

```
virtual MonTimeSpan *For(int nPos)=0;
```

Zweck:

Dies ist die Methode, in der aus Zeitstempeln Zeitspannen gewonnen werden. Die Methode ist dabei in der Klasse SpanGenerator als `pure virtual` deklariert, da für jeden Typ von Zeitspanne, der generiert werden kann, jeweils eine andere Funktion tatsächlich ausgeführt werden muss. Daher existiert für jeden Zeitspannen-Typ eine Unterklasse, die die For-Methode implementiert (s. auch Abb. 2.2).

Beispielhaft soll hier an einer solchen Unterklasse deren Funktionsweise demonstriert werden. Betrachtet wird die Unterklasse Generate_ModInitSpan, die die Zeitspannen der Modulinitialisierung generiert. Die vollständigen Deklarationen lauten

```
class Generate_ModInitSpan : public SpanGenerator{...};
MonTimeSpan *Generate_ModInitSpan::For(int nPos);
```

Das übergebene Argument bei For ist die Position des aktuell betrachteten Zeitstempels innerhalb der Monitor-Log-Datei. Als erstes wird nun der Stempel anhand dieser Positionsnummer aus der Datei gelesen. Danach wird überprüft, ob der Typ dieses Stempels zur eigenen Klasse passt, also in diesem Fall, ob es ein Stempel des Typs „ModInitStart“ ist, der den Startpunkt der zu generierenden Spanne angibt. Ist dem nicht so, wird ein Nullzeiger zurückgegeben.

Wird der Typ als der eigene erkannt, so wird eine neue Zeitspanne angelegt und mit den entsprechenden Informationen versorgt. So wird z.B. die Startzeit, der Typ, die statische und dynamische ID eingetragen. Durch den Aufruf der beiden Hilfsmethoden Search4Stop und Search4Id werden die Stoppzeit sowie der entsprechende

PATO-Key (falls erzeugt) ermittelt und ebenfalls in die neue Zeitspanne eingetragen.

Hinzu kommt noch die aktuelle laufende Nummer der Spanne. Alle Spannen werden fortlaufend durchnummeriert, wodurch später Such- oder Filteroperationen auch ab einer bestimmten Position möglich sind. Da die Nummern nur bei der Generierung erzeugt werden, können durch den Aufruf z.B. der Select-Methode einige Nummern herausgefiltert werden, wodurch die Nummerierung nicht mehr fortlaufend ist. Die innerhalb dieser Klassenbibliothek implementierten Operationen, die mit den „laufenden“ Nummern der Spannen arbeiten, erkennen jedoch solche Fälle und beginnen dann mit der jeweils nächsten, noch vorhandenen Nummer.

Zum Schluss der Generierung wird ein Zeiger auf die so erstellte Zeitspanne zurückgegeben.

Soll ein neuer Zeitspannengenerator für einen neuen Spannentyp angelegt werden, ist also Folgendes zu tun:

- Deklaration der neuen Klasse als Unterklasse zu SpanGenerator („`: public SpanGenerator`“). Enthaltene Methode: For.
- Implementierung der For-Methode nach oben beschriebenem Muster.
- Modifizierung des Konstruktors der SpanGenerator-Klasse. Ein Objekt der neuen Klasse muss erzeugt und in die Liste mit Objekten von Spannengeneratoren eingetragen werden.

2.5.2 Search4Stop

Deklaration:

```
TIME Search4Stop(const MonTimeStamp *pTimeStamp,
                 unsigned uType,
                 int nStart);
```

Zweck:

Mittels Search4Stop wird bei der Erzeugung von Zeitspannen der Stempel gesucht, der das Stoppereignis der jeweiligen Spanne anzeigt. Als Argument wird ein Zeiger auf den Startstempel der Spanne übergeben, woraus die statische und dynamische ID des gesuchten Stempels ersichtlich werden. Zur Suche werden weiterhin der gesuchte Stempeltyp (zweites Argument) und die Position, ab der innerhalb der Datei gesucht werden soll (drittes Argument) benötigt (da der Stoppstempel zeitlich nach dem Startstempel generiert worden sein sollte, muss der Stempel nur ab der Position des Startstempels in der Datei gesucht werden).

Wird der gesuchte Stempel gefunden, wird seine Zeit zurückgegeben (und dort als Stoppzeit der Spanne eingetragen). Wird kein passender Stempel gefunden, so wird der Benutzer informiert und die Zeit des im ersten Argument übergebenen Stempels zurückgegeben (die generierte Zeitspanne hätte dann die Länge null, was die weitere Auswertung aber nicht verhindert, sondern nur verfälscht).

2.5.3 Search4Id

Deklaration:

```
unsigned Search4Id(const MonTimeStamp *pTimeStamp,
                   int nClass);
```

Zweck:

Diese Methode dient zur Suche des richtigen PATO-Keys für eine Zeitspanne, falls PATO-Keys vorher erzeugt wurden (ansonsten wird 0 zurückgegeben). Sie wird verwendet, um bei der Generierung der Zeitspannen gleich den entsprechenden PATO-Key mit zuzuordnen, so dass, wenn ein PATO-Logfile erzeugt werden soll, diese Informationen schon vorliegen, und nicht erst aufwendig erzeugt werden müssen (man müsste dann jede Zeitspanne erneut bearbeiten und um den PATO-Key ergänzen). In dem Fall, dass kein PATO-Logfile erstellt werden soll, ist dies ein geringer Mehraufwand.

Die Methode sucht in der internen Datenstruktur nach dem richtigen PATO-Key anhand der übergebenen Informationen. Das erste Argument ist ein Zeiger auf den Zeitstempel, zu dem der Key gesucht werden soll; aus ihm werden z.B. die statische und dynamische ID ausgelsen. Das zweite Argument ist die „Klasse“ von Key, die gesucht wird. Da ein Modul bzw. eine Transistion mehreren Keys zugeordnet ist (ein Modul hat, wie oben schon erwähnt, drei zugeordnete Keys) wird mit Hilfe dieser „Klasse“ genau eingeschränkt, welcher dieser Keys gemeint ist.

Für den Fall, dass kein passender Key gefunden wird, wird 0 zurückgegeben.

Kapitel 3

Anwendungen

Nachdem nun die Funktionen der *monitorlib* erläutert wurden, sollen in diesem Kapitel zwei Beispiel-Anwendungen vorgestellt werden: Zuerst eine einfache Auswertung und dann ein Tool zur Konvertierung eines Monitor-Logfiles in das PATO-Format.

3.1 Ein Auswerte-Tool

Für die Beschreibung des Beispiel-Auswertetools gelte folgende Ausgangssituation: Es wurde eine *Estelle*-Spezifikation erstellt, die die in Abbildung 3.1 gezeigte Struktur besitzt, also aus zwei Modulen besteht, die ihrerseits je zwei Transitionen besitzen. Es soll dabei angenommen werden, dass die Ausführungsdauer beider Module ungefähr identisch sein soll. Die Spezifikation ist mit dem *xec* compiliert und ausgeführt worden, und nun soll untersucht werden, ob die Spezifikation (oder der Compiler) optimiert werden können.

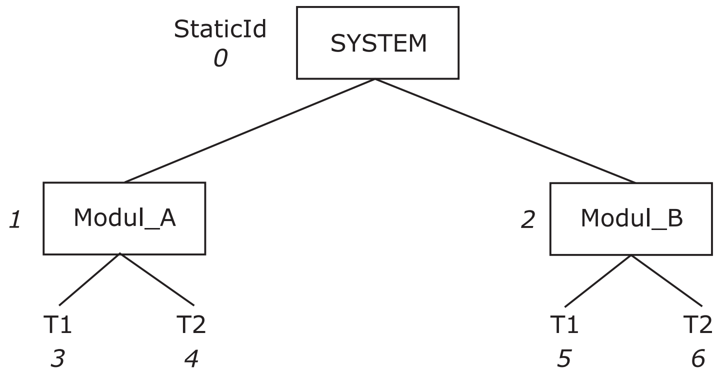

Abbildung 3.1: *Estelle*-Struktur der Beispiel-Anwendung

Um einen allgemeinen Überblick darüber zu erhalten wird nun die Auswahl- und Feuerzeit ausgewertet. Man betrachtet dabei zuerst die gesamte Zeit dieser Spannen im System und danach die Summe dieser Zeiten der Module A und B. Verbraucht eines der beiden Module deutlich mehr Zeit als das andere, wäre das in den Ergebnissen sichtbar.

Um diese Frage mit diesem Schema zu beantworten wird ein Auswerteprogramm geschrieben, das in Abbildung 3.2 zu sehen ist. Dieses wird nun detailliert besprochen:

- Zeile 3: ein Objekt der Klassenbiliothek *monitorlib* wird angelegt. Da auch die *xec*-spezifischen Methoden benötigt werden, wird also ein Objekt der Klasse `xecmonlib` erzeugt.
- Zeile 5: Die Datei `monitor.log` im aktuellen Verzeichnis wird eingelesen. Die beim Ablauf generierten Zeitstempel stehen jetzt zur Verfügung.

```
 1  main()
 2  {
 3    xecmonlib monitorlib;
 4
 5    monitorlib.ReadLog("monitor.log");
 6
 7    XPtrList<MonTimeSpan> alleSpannen, Auswahl;
 8
 9    monitorlib.CreateSpans(&alleSpannen);
10
11    for(int i = 0; i <= 2; i++)
12    {
13      monitorlib.Select(ModSelectLocal, i, -1, 1, true, &alleSpannen,
                          &Auswahl);
14      monitorlib.Select(ModSelectChild, i, -1, 1, true, &alleSpannen,
                          &Auswahl);
15      monitorlib.Select(ModFire, i, -1, 1, true, &alleSpannen, &Auswahl);
16
17      TIME summe = monitorlib.Summarize(&Auswahl);
18      printf("Auswählen und Feuern Modul %d: %lld\n", i, summe);
19
20      Auswahl.clear();
21    }
22  }
```

Abbildung 3.2: Listing des Auswertungs-Beispiels

- Zeile 7: Zwei Listen für Zeitspannen werden angelegt, eine, die alle erzeugten Zeitspannen enthalten soll, eine andere für die jeweils herausgefilterten Spannen.
- Zeile 9: Die Zeitspannen werden erzeugt und in die übergebene Liste eingetragen.
- Zeile 11: Für jedes der drei zu betrachtenden Module (das System, Modul A und Modul B) geschieht Folgendes:
- Zeile 13: Die lokale Auswahlzeit des jeweiligen Moduls wird aus der Gesamtmenge herausgefiltert und in die Auswahl-Liste eingetragen.
- Zeile 14+15: Genauso wird mit der Auswahlzeit der jeweiligen Kindmodule und dem Feuern der Module verfahren. Die hier herausgefilterten Spannen werden jeweils zur Liste „Auswahl“ hinzugefügt (da die Liste ja zwischendurch nicht geleert wird). Nach Ausführung von Zeile 15 enthält diese Liste also alle Spannen, die die Auswahl- und Feuerzeit des jeweiligen Moduls angeben.
- Zeile 17+18: Nun werden die einzelnen Längen der Zeitspannen addiert und das Ergebnis ausgegeben.
- Zeile 20: Damit beim nächsten Schleifendurchlauf die Auswahl-Liste wieder leer ist, wird dies mit dem Methoden-Aufruf clear der XPtrList-Klasse erledigt.

Bei der Ausführung dieses Auswertetools erhält der Benutzer also drei Ausgaben, die die Summe der betrachteten Zeitspannen für das System (also die zum Auswählen und Feuern benötigte Gesamtzeit) und die beiden Module angeben. Sollte eins der Module deutlich mehr Zeit beanspruchen, so wird dies an den Ergebnissen sichtbar sein.

An diesem Beispiel sollte die grundsätzliche Struktur eines Auswertetools erkennbar werden:

- Einlesen des Logfiles
- Erzeugung der Zeitspannen

- Herausfiltern derjenigen Spannen, die von Interesse sind
- weitere Bearbeitung/Auswertung dieser Spannen (z.B. Summieren oder Konvertieren in PATO-Format, s. Kap. 3.2).

Andere in der *monitorlib* vorhandene Methoden können ebenfalls nützlich sein, so z.B. die IsChild-Methode, mit der Auswertungen ja nach Verwandschaftsbeziehungen möglich sind (sollten im obigen Beispiel nur die Auswahl- und Feuerzeit der Module ohne die entsprechenden Zeiten der Transitionen gemessen werden, hätte man mittels IsChild die Verwandschaft erkennen und die entsprechenden Zeiten wieder abziehen können).

3.2 Konvertierung: *mon2pato*

mon2pato ist ein kleines Tool zur Konvertierung eines Monitor-Logfiles im neuen Format in das alte PATO-Format. Auf diese Weise kann das grafische Auswertewerkzeug von PATO auch weiterhin zur Auswertung der Daten genutzt werden, die mit der neuen Monitoring-Lösung generiert wurden. Wie man in Abbildung 3.3 sieht, ist auch eine Konvertierung ohne großen Aufwand realisierbar.

```
1   main(int argc, char **argv)
2   {
3    xecMonLib monitorlib;
4    MonitorLog monlog;
5
6    if ((argc < 2) || (argc > 3))
7    {
8      printf("\nUsage: %s <SourceFile> [<DestPrefix>]\n\n", argv[0]);
9      exit(1);
10   }
11
12   XPtrList<MonTimeSpan> TimeSpanList;
13
14   monitorlib.ReadLog(argv[1], &monlog);
15   monitorlib.CreateDB();
16   monitorlib.CreateSpans(&TimeSpanList);
17   monitorlib.SavePato(argv[2] ? argv[2]  : "", &TimeSpanList);
18  }
```

Abbildung 3.3: *mon2pato*-Listing

Die Programmzeilen im einzelnen:

- Zeile 3+4: Instanziierung eines Objektes der *monitorlib* (*xec*-spezifisch).
- Zeile 6-10: Überprüfung der Anzahl übergebener Parameter.
- Zeile 12: Anlegen einer Liste für die Zeitspannen.
- Zeile 14: Einlesen der Zeitstempel. Übergeben wird der Filename (erster *mon2pato*-Parameter).
- Zeile 15: Erstellen der (vorerst internen) Datenbank mit PATO-Keys. Dies muss vor dem CreateSpans-Aufruf geschehen, da die Zuordnung zwischen Zeitspannen und PATO-Keys direkt bei der Spannen-Generierung erfolgt.
- Zeile 16: Jetzt können die Zeitspannen generiert werden. Übergeben werden muss dieser Methode der Filenamenpräfix für die PATO-Dateien. Dies ist das zweite Argument von *mon2pato* oder ein leerer String, falls dieses nicht existiert.

- Zeile 17: Zum Schluss: Speichern der generierten Zeitspannen im PATO-Format in die Dateien (je eine pro Modul) und Erzeugung des Datenbank-Files für die Keys.

Man sieht, dass auch die PATO-Konvertierung mit wenigen Methoden-Aufrufen möglich ist. Selbstverständlich ist auch die Kombination beider hier vorgestellter Anwendungen sinnvoll. So könnte man z.B. zuerst aus der Gesamtzahl der Zeitspannen diejenigen herausfiltern, die von Interesse sind, und nur diese dann zur weiteren Analyse oder zum Erstellen von statistischen Auswertungen in das PATO-Format konvertieren.

Kapitel 4

Ausblick

In den vorherigen Kapiteln wurde erläutert, welche Möglichkeiten die entwickelte Klassenbibliothek *monitorlib* besitzt, und wie man diese nutzt, um flexibel eigene Auswertungen von Monitoring-Daten zu erstellen. Nun soll gezeigt werden, wie man dieses Konzept erweitern kann.

Ein Nachteil der entwickelten Lösung besteht darin, dass Auswertungen vom Benutzer in C++ programmiert werden müssen, falls keine vorgefertigten Tools für diese Anwendung existieren. Obwohl gerade dies die hohe Flexibilität der Lösung begründet, setzt es doch entsprechende Programmierkenntnisse und -möglichkeiten (Compiler etc.) voraus. Wünschenswert wäre also eine Lösung, mit der der Benutzer bei gleichbleibend hoher Flexibilität einfacher seine gewünschte Auswertung spezifizieren kann, also eine höhere Abstraktionsebene.

Zu diesem Zweck wäre die Entwicklung einer eigenen Sprache möglich, durch die angegeben werden kann, welche Auswertung der Benutzer wünscht. Die Sprache müsste dabei einerseits die vorhandene Funktionalität der Bibliothek abbilden (also Sprachkonstrukte zum Filtern, Suchen oder Konvertieren von Zeitspannen besitzen) und andererseits dies möglichst flexibel und mächtig halten.

Eine solche Sprache würde intern weiterhin die Methoden der Klassenbiliothek aufrufen, deren Funktionalität also erhalten bliebe. Der Benutzer könnte nun ohne C++-Kenntnisse die Bibliothek bequem nutzen und mit solchen Sprachelementen komplexe Auswertungen einfacher formulieren, als das bisher möglich ist.

Eine weitere interessante Möglichkeit wäre die einfache Spezifikation von virtuellen Messpunkten durch den Benutzer mittels einer solchen Sprache (also ohne durch Modifizierungen am Code einen neuen Spannen-Typ einzuführen). So könnten neue Zeitspannen zur Auswertezeit durch die Angabe eines Start- und Stoppereignisses angegeben werden, und der Benutzer wäre nicht mehr auf die Änderung der vorgegebene Zuordnung von Zeitstempel-Typen zu Zeitspannen-Typen (s. Tab. 2.1) durch Modifizierungen des Codes angewiesen. Damit solche Zeitspannen getrennt nach Modulinstanzen erzeugbar sind, aber nicht für jede Instanz definiert werden müssen, sind auch Variablenbindungen innerhalb der Sprache wünschenswert. Eine Definition einer virtuellen Zeitspanne „ModSelect“, die für jede Modulinstanz eine neue Spanne definiert, deren jeweiliger Start- bzw. Stoppzeitpunkt durch den Stamp-Typ „ModSelectLocalStart“ bzw. „ModSelectEnd“ angegeben wird, könnte dann so aussehen:

```
ALL modInstance: SPAN ModSelect[modInstance]
                 ( TYPE ModSelectLocalStart INST modInstance
                   TO TYPE ModSelectEnd INST modInstance )
```

Die Implementierung einer solchen Sprache wurde bereits für diese Projektarbeit erwogen. Die Realisierung zusätzlich zur Entwicklung einer Klassenbibliothek würde durch die Anforderungen (hohe Abstraktion bei gleichzeitig großer Ausdrucksfähigkeit) jedoch sehr aufwendig und über den Rahmen der Projektarbeit hinausgehen. Daher wurde auf eine Entwicklung und Implementierung einer solchen Sprache verzichtet.

Kapitel 5

Zusammenfassung

In den vorherigen Kapiteln wurde das Prinzip des *Software-Monitorings* und die entwickelte Klassenbibliothek *monitorlib* vorgestellt.

Beschrieben wurde das Prinzip von Software-Monitoring, wobei zuerst auf die Formale Beschreibungssprache *Estelle* eingegangen wurde, auf die die Monitoring-Verfahren angewandt werden. Desweiteren wurde erläutert, was Zeitstempel sind und wie sie erfasst und ausgewertet werden. Ausserdem wurde beschrieben, in welchen Punkten sich die neu entwickelte Monitoring-Lösung mit Anwendung auf den *xec* von einer bereits existierenden Lösung (PATO) unterscheidet.

Weiterhin wurde die entwickelte Klassenbibliothek *monitorlib* vorgestellt (Kapitel 2). Dabei wurder zuerst die Struktur der Lösung und das Prinzip der Generierung von Zeitspannen beschrieben, anschließend wurden die Methoden der drei Klassen monitorlib, xecmonlib und SpanGenerator mit ihrer Deklaration, ihrem Zweck und einem Beispiel vorgestellt. Ausserdem wurde erläutert, wie die Klassen ergänzt werden müssen, wenn neue Typen von Zeitspannen generiert werden sollen.

Behandelt wurden ebenfalls zwei Beispiel-Anwendungen (Kapitel 3), ein einfaches Auswerte-Tool und das Programm *mon2pato* zur Konvertierung von mit der neuen Lösung gewonnenen Monitoring-Daten in das PATO-Format. Jedes der beiden Beispiel-Listings wurde detailliert besprochen. Ziel war es, dem Benutzer den grundsätzlichen Aufbau eines Auswerte-Tools und das Zusammenspiel der einzelnen Methoden zu erläutern.

Schließlich wurde dann eine Erweiterungsmöglichkeit der entwickelten Lösung vorgestellt. Denkbar ist danach eine höhere Abstraktionsebene durch die Entwicklung einer Sprache. Der Benutzer könnte dann auch ohne C++-Programmierkenntnisse komplexe Auswertungen formulieren.

Literaturverzeichnis

[DeBu89] Dembinski, P., Budowski, S., *Specification Language Estelle*, in: M. Diaz et al. (eds.), *The Formal Description Technique Estelle*, North-Holland, 1989

[Lan92] Langendörfer, Horst, *Leistungsanalyse von Rechensystemen: Messen, Modellieren, Simulation*, Hanser Verlag, München, 1992

[ISO97] ISO/TC97/SC21, *Estelle - A Formal Description Technique Based on an Extended State Transition Model*, ISO/TC97/SC21, IS 9074, 1997

[Pen96] Penner, H., *Erstellung eines Software-Monitors zur Analyse automatisch generierter Protokollimplementierungen*, Diplomarbeit, Universität Kaiserslautern, Fachbereich Informatik, AG Rechnernetze, 1996

[ThGo98] Thees, J., Gotzhein, R., *The eXperimental Estelle Compiler - Automatic Generation of Implementations from Formal Specifications*, in: M. Ardis (edt.), Proceedings of The 2nd Workshop on Formal Methods in Software Practice (FMSP '98), Clearwater Beach, Florida, USA, March 1998

[The98] Thees, J., *Protocol Implementation with Estelle - from Prototypes to Efficient Implementations*, in: S. Budkowski, S. Fischer, R. Gotzhein (Edts.): Proceedings of the 1st International Workshop on the Formal Description Technique Estelle (ESTELLE '98), Evry, France, November 1998

[SiSt93] Sijelmassi, R., Strausser, B., *The PET and DINGO tools for deriving distributed implementations from Estelle*, Computer Networks and ISDN Systems 25, 1993

Abbildungsverzeichnis

Tabellenverzeichnis