Erstellung eines Software-Monitors zur Analyse automatisch generierter Protokollimplementierungen

Hartmut Penner

16. Januar 1996

Inhaltsverzeichnis

1 Einführung **3**
- 1.1 Problembeschreibung 3
- 1.2 Vorgehensweise 4

2 Pet-Dingo Entwicklungssystem **5**
- 2.1 Estelle Ausführungsmodell 5
- 2.2 Verteilte Implementierung des Estelle-Ausführungsmodells 6
  - 2.2.1 Pet-Dingo Ausführungsmodell 7
  - 2.2.2 Beispiel 11
- 2.3 Generierung der Implementierung 13
  - 2.3.1 Implementierung des Ausführungsmodells 13
  - 2.3.2 Übersetzung der Spezifikation 14
  - 2.3.3 Erzeugung des ausführbaren Kodes 16

3 Software-Monitor **17**
- 3.1 Meßmodell 18
  - 3.1.1 Meßmodell zur Leistungsanalyse von Protokollimplementierungen 18
  - 3.1.2 Meßmodell zur Leistungsanalyse des Ausführungmodells . 20
- 3.2 Meßverfahren 24
  - 3.2.1 Aufbau eines Software-Monitors 25
  - 3.2.2 Meßmethoden 26
  - 3.2.3 Entwickeltes Meßverfahren 29
- 3.3 Implementierung des Software-Monitors 31
  - 3.3.1 Klassen des Software-Monitors 31
  - 3.3.2 Änderungen des site_server 33
  - 3.3.3 Änderungen der Laufzeitbibliotheken 33
  - 3.3.4 Änderungen des DINGO 34
  - 3.3.5 Meßpunkte des Leistungmodells 34
  - 3.3.6 Meßpunkte des Ausführungmodells 37

4 Entwicklung des Auswertungswerkzeugs **40**
4.1 Eigenschaften der Laufzeitinformationen 40
4.2 Funktionalität des Auswertungswerkzeugs 43
4.3 Implementierung des Auswertungswerkzeugs 46

5 Ergebnisse **52**
5.1 Leistungsanalyse eines AB-Protokolls 52
5.1.1 Analyse mit dem ersten Meßmodell 52
5.1.2 Analyse mit dem zweiten Meßmodell 55
5.2 Leistungsanalyse des Pet-Dingo Entwicklungspaketes 57
5.2.1 Analyse eines Systems ohne Kommunikation 57
5.2.2 Analyse eines Systems mit Kommunikation 58

6 Ausblick **59**

A Analysierte Spezifikation **60**
A.1 AB-Protokolls 60
A.2 System ohne Kommunikation 67
A.3 System mit Kommunikation 67

B Tutorial **69**
B.1 Bedienung des Software-Monitors 69
B.1.1 Formate der Kommentare 70
B.1.2 Optionen von Dingo 73
B.2 Bedienung von PATO 74
B.2.1 Verwaltung der Meßinformationen 74
B.2.2 Ermittlung der Leistungsmaße 75

Kapitel 1

Einführung

Die Handimplementierung ist heute die verbreitete Vorgehensweise, um zu effizienten Implementierungen von Protokollen zu gelangen. Die hohe Komplexität heutiger Protokolle bereitet bei dieser Art der Implementierung immer größere Probleme. Zum einen steigt der Implementierungsaufwand, zum anderen läßt sich die Korrektheit einer Implementierung nicht oder nur bedingt mit hohem Aufwand feststellen. Es wird deshalb zunehmend nach Verfahren gesucht, die sowohl den Prozeß der Implementierung, als auch die Verifikation einer Implementierung unterstützen. Liegt ein Protokoll in einer formalen Beschreibungstechnik spezifiziert vor, so gibt es für einige dieser Techniken Kode-Generatoren, die anhand einer Spezifikation automatisch ausführbaren Kode generieren. Wenn ein solcher Kode-Generator existiert, so verringert sich der Implementierungsaufwand beträchtlich, und eine Verifikation kann entfallen.

1.1 Problembeschreibung

Die funktionale Korrektheit einer Implementierung ist allerdings nur eine notwendige, aber nicht hinreichende Eigenschaft einer Protokollimplementierung. Es treten heute mehr und mehr die nichtfunktionalen Aspekte von Implementierungen in den Vordergrund, unter diesen insbesonders die zeitlichen Aspekte. Und gerade bei den automatisch implementierten Protokollimplementierungen ist der wesentliche Nachteil die geringe Leistung im Vergleich mit einer Handimplementierung.

Um diesen Mangel zu beheben, wird an verschiedenen Verfahren gearbeitet, um das Leistungverhalten zu steigern[GBE95]. Damit man diese Verfahren beurteilen kann, muß die Leistung einer Implementierung quantitativ bewertbar sein. Dafür müssen Leistungmaße definiert werden, die das Leistungsverhalten beurteilbar und vergleichbar machen. Um diese Leistungsmaße bestimmen zu können, benötigt man eine Menge von Informationen, die zur Laufzeit des Systems ermittelt werden müssen.

Es muß demnach ein Verfahren entwickelt werden, welches die Bestimmung der Laufzeitinformation und die Berechnung der Leistungsmaße ermöglicht. Dieses Verfahren soll für einen existierenden Kode-Generator, das Pet-Dingo Entwicklungssystem entwickelt werden. Mit diesem ist es möglich, ausgehend von einer Estelle-Spezifikation eine verteilte Implementierung zu erzeugen. Das Verfahren soll möglichst so entwickelt werden, daß es einfach auf andere Kode-Generatoren übertragen werden kann.

1.2 Vorgehensweise

Für das Erreichen der oben angeführten Ziele wird folgende Vorgehensweise gewählt:

Die Ermittlung der notwendigen Information zur Bestimmung der Leistungsmaße und die Berechnung der Leistungsmaße wird auf zwei Werkzeuge verteilt. Dies erfolgt unter anderem deshalb, da die Ermittelung der Laufzeitinformation direkt von dem verwendeten Kode-Generator abhängig ist, wohingegen die Berechnung der Leistungsmaße weitestgehend unabhängig davon ist. Damit ist sichergestellt, daß der hier verwendete Ansatz leicht auf andere Kode-Generatoren übertragen werden kann.

In Kapitel 2 wird auf das Pet-Dingo Entwicklungspaket eingegangen, insbesonders darauf, wie, ausgehend von einer Spezifikation, die Implementierung generiert wird.

In Kapitel 3 werden die Leistungmaße definiert, die zur Leistungsanalyse benötigt werden. Darauf aufbauend wird darauf eingegangen, wie die dafür notwendigen Laufzeitinformationen ermittelt werden. Für das Pet-Dingo Entwicklungspaket wird das Verfahren vorgestellt, welches die Ermittlung der Laufzeitinformation ermöglicht. Es wurde eine Implementierung dieses Verfahrens erstellt, welche im letzten Abschnitt beschrieben wird.

Das Kapitel 4 beschreibt die Entwicklung und den Aufbau des Auswertungswerkzeugs PATO. Dafür wird insbesonders auf den Prozeß eingegangen, wie aus den Laufzeitinformationen die Leistungsmaße ermittelt werden können.

In Kapitel 5 werden einige Ergebnisse von erfolgten Leistungsanalysen von Protokollimplementierungen vorgestellt.

Kapitel 2

Pet-Dingo Entwicklungssystem

Mit dem Pet-Dingo Entwicklungsssystem ist es möglich, ausgehend von einer formalen Estelle-Spezifikation eine verteilte Implementierung zu generieren. Wegen der Verwendung von Estelle handelt es sich allerdings um geschlossene Systeme. Um die generierten Protokollinstanzen für "sinnvolle" Anwendungen benutzen zu können, müssen Veränderungen an einigen Stellen des erzeugten Kodes vorgenommen werden. Diesen Vorgang zu automatisieren wird in anderen Arbeiten verfolgt. Da das Generieren von Prototypen damit prinzipiell möglich, soll in dieser Arbeit davon ausgegangen werden, daß man ausführbare Protokollinstanzen erhält.

In diesem Kapitel sollen einige Aspekte des Generierungsprozesses beschrieben werden, insbesonders solche, welche einen maßgeblichen Einfluß auf das zeitliche Verhalten der Implementierung haben. Es wird das Ausführungsmodell von Estelle beschrieben, die Umsetzung dieses Ausführungmodells für die verteilte Implementierung, sowie der Generierungsprozess einer Implementierung.

2.1 Estelle Ausführungsmodell

Die formale Beschreibungstechnik Estelle wurde entwickelt, um verteilte Systeme spezifizieren zu können. Ein verteiltes System wird dabei als ein System miteinander kommunizierender erweiterter endlicher Automaten beschrieben. Eine Estelle-Spezifikation, die ein solches verteiltes System darstellt, ist als eine Menge von Modulen und Kanälen definiert, wobei durch die Module die erweiterten endlichen Automaten und durch die Kanäle die Verbindungsstruktur derselben beschrieben werden. Das Verhalten einer solchen Spezifikation wird mit Hilfe einer operationalen Semantik beschrieben. Es wird also eine abstrakte Maschine definiert, auf welcher die Spezifikation ausgeführt wird [RiMaDe83]. Die Ausführung einer Spezifikation auf dieser abstrakten Maschine führt zur Instanziierung von Modulen und Kanälen, in denen Transitionen ausgeführt und über die Interaktion verschickt werden. Die Ausführung beginnt mit der initialen Transition der Spe-

zifikation. Zu jedem Zeitpunkt befindet sich die Spezifikation bei der Ausführung auf der abstrakten Maschine in einer globalen Situation sit_{SP}, die definiert ist als ein Tupel $(gid_{SP}, A_1, ..., A_n)$. Dabei ist gid_{SP} die globale Momentbeschreibung der Spezifikation, die als der erweiterte Zustand aller Modulinstanzen, der Verbindungsstruktur und dem Inhalt der Warteschlangen definiert ist. Bei den A_k handelt es sich um die Menge zur Ausführung ausgewählten Transitionen der Subsysteme S_k. Befindet sich das System in einer globalen Situation sit_i, so ist eine mögliche globale Situation sit_{i+1} definiert als $(t(gid_{SP}), A_1, .., A_k \backslash \{t\}, .., A_n)$ oder $(gid_{SP}, A_1, .., AS(gid_{SP} \backslash S_k), .., A_n)$, falls $A_k = \emptyset$ für ein beliebiges $k \in (1...n)$. $t(gid_{SP})$ ist definiert als Anwendung der Transition t auf die globale Momentbeschreibung, mit anderen Worten die Veränderung des globalen Zustands durch das Ausführen der Transition t. $AS(gid_{SP} \backslash S_i)$ bestimmt die Menge schaltbarer Transitionen des Subsystems S_k, die in Folge ausgeführt werden. Sowohl $AS(gid_{SP} \backslash S_k)$, als auch $t(gid_{SP})$ sind Relationen wegen möglicher Indeterminismen bei der Auswahl der schaltbaren Transition und der Ausführung einer Transition. Eine *Berechnung* ist definiert als eine Folge von globalen Situationen $< sit_0, ...sit_n.. >$, wobei sit_0 die initiale globale Situation des Systems ist und die folgenden globalen Situationen sich nach dem oben angeführten Verfahren ergeben. Die Semantik einer Spezifikation ist durch die Menge der möglichen Berechnungsfolgen definiert.

Bei den zu spezifizierenden verteilten Systemen handelt es sich i. allg. um nebenläufige Systeme. Das bedeutet, daß die Berechnung möglicherweise parallel ausgeführt werden kann. Die Nebenläufigkeit wird in Estelle durch die Interleaving-Semantik ausgedrückt. Synchrone Nebenläufigkeit wird dadurch beschrieben, daß die ausgewählten Transitionen eines Subsystems in beliebiger Reihenfolge ausgeführt werden können. Asynchrone Nebenläufigkeit wird durch die beliebige Reihenfolge der Auswahl und Ausführung der Transitionen verschiedener Subsysteme beschrieben.

2.2 Verteilte Implementierung des Estelle-Ausführungsmodells

Liegt eine Spezifikation eines verteilten Systems in Estelle vor, so gibt es neben der Handimplementierung zwei Möglichkeiten, ablauffähigen Kode aus dieser Spezifikation zu erhalten. Zum einen kann man auf der Grundlage der abstrakten Maschine der Estelle-Semantik eine Implementierung generieren, also zu jedem Zeitpunkt exakt eine Aktion ausführen. In diesem Fall wird die Nebenläufigkeit in keiner Weise ausgenutzt. Außerdem ist es nicht möglich, einzelne Modulinstanzen aus dem erzeugten Kode herauszulösen und einen Prototypen daraus zu erstellen. Diese Vorgehensweise wird deshalb i. allg. nur für die Simulation von Estelle-Spezifikationen angewandt.

Eine andere Vorgehensweise ist, daß die vorhandene Nebenläufigkeit einer Spezifikation ausgenutzt wird für eine parallele, verteilte Ausführung. Dafür muß ausgehend von der abstrakten Maschine der Estelle-Semantik eine andere Maschine definiert werden, welche einen parallelen Kontrollfluß ermöglicht. Eine Forderung an diese Maschine ist, daß alle bei der Abarbeitung einer Spezifikation möglichen Berechnungsfolgen auch auf der abstrakten Maschine der Estelle-Semantik möglich sind. Wenn diese Forderung erfüllt ist, handelt es sich um eine, laut Estelle-Semantik korrekte Implementierung.

Im folgenden soll diese Maschine als *Estelle-Maschine* bezeichnet werden. Diese *Estelle-Maschine* kann man sich derart vorstellen, daß für jeden der parallelen Kontrollflüsse eine Instanz vorhanden ist. Jede Instanz arbeitet einen bestimmten Teil der Spezifikation ab. Die zur Einhaltung der Estelle-Semantik notwendige Synchronisation der Instanzen untereinander erfolgt durch Kommunikation.

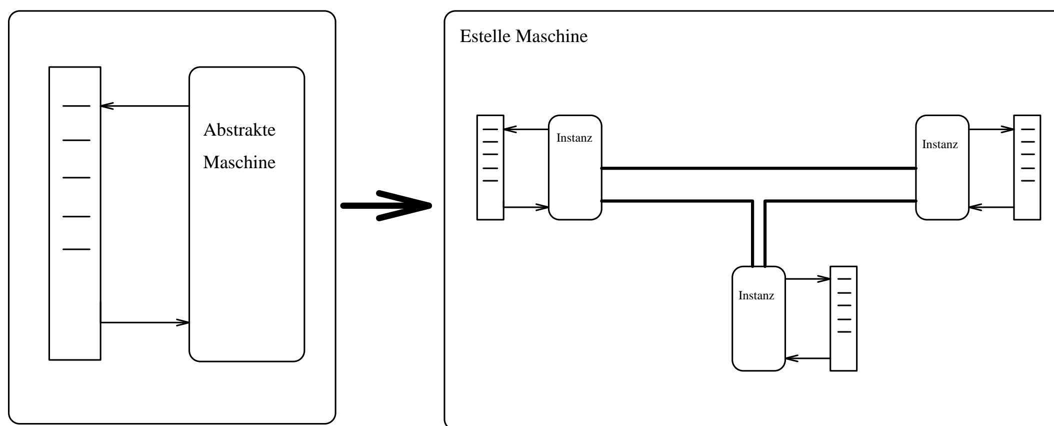

Abbildung 2.1: Estelle-Maschine

Für die Entwicklung einer solchen Estelle-Maschine muß festgelegt werden, welche nebenläufige Ausführung der abstrakten Maschine in eine parallele Ausführung überführt werden soll. Davon ausgehend wird das Ausführungsmodell der Instanzen und das Synchronisationsprotokoll erstellt. Sind diese beiden Schritte erfolgt, muß man eine Verteilfunktion definieren, welche die jeweils notwendigen Teile der Spezifikation den einzelnen Instanzen zuordnet.

Diese Vorgehensweise wird auch bei dem Pet-Dingo Entwicklungssystem verfolgt. Dies soll im folgenden Abschnitt beschrieben werden.

2.2.1 Pet-Dingo Ausführungsmodell

Das Pet-Dingo Entwicklungssystem ermöglicht es, beliebige Modulinstanzen zu verteilen und auszuführen, d. h. daß möglicherweise für jede Modulinstanz eine Instanz der *Estelle-Maschine* zur Verfügung stehen muß, auf der die Transitionen dieses Moduls ausgeführt werden. Bei der Nebenläufigkeit, die für eine parallele Ausführung genutzt wird, handelt es sich also um Nebenläufigkeit, welche

bei der Auswahl und Ausführung von Transitionen verschiedener Modulinstanzen zugrunde liegt. Die Instanzen der *Estelle-Maschine* arbeiten dann jeweils die Transitionen der Modulinstanzen ab, falls diese verteilt werden sollen. Dabei sind den Subsysteme in jedem Falle eigene Instanzen zugeordnet.

Das Problem, welches sich bei einer Parallelisierung der Ausführung der Modulinstanzen ergibt, ist die damit einhergehende Verteilung der globalen Momentbeschreibung gid_{SP}. Für das Ausführen der Relationen $AS(gid_{SP} \backslash S_i)$ und $t(gid_{SP})$ müssen die relevanten Bestandteile von gid_{SP} bei einer Verteilung über Kommunikation erlangt werden. Beispielweise muß ein Subsystem bei der Ausführung der Relation $AS(gid_{SP} \backslash S_i)$ mit den Sohnmodulinstanzen kommunizieren, um festzustellen, ob diese schaltbare Transitionen besitzen. In den folgenden beiden Abschnitten wird gezeigt, wie das verteilte Ausführungsmodell von Pet-Dingo realisiert wurde. Insbesonders wird auf die Aspekte eingegangen, die der Einhaltung der Estelle-Semantik dienen[SIGA88].

Auswahlphase

In Estelle können die $AS(gid_{SP} \backslash S_i)$ der Subsysteme in beliebiger Reihenfolge zueinander ausgeführt werden. Damit wird die vollständige asynchrone nebenläufige Ausführung der Subsysteme ausgedrückt. Bei einer verteilten Implementierung kann man demnach die Auswahl der Subsysteme unabhängig voneinander vornehmen. Allerdings sind einige andere Aspekte zu beachten, um die Auswahl konform zur Estelle-Semantik zu gestalten. Diese werden im folgenden aufgelistet:

- Bei Beginn der Auswahlphase hat die höchste Modulinstanz des Subsystems das Auswahlrecht und überprüft, ob sie schaltbare Transitionen besitzt. Wenn mindestens eine schaltbare Transition vorhanden ist, wird diese ausgeführt.

- Hat eine Vatermodulinstanz das Auswahlrecht aber keine schaltbaren Transitionen, gibt sie das Auswahlrecht an alle Sohnmodulinstanzen weiter.

- Bei einer mit *(system)-process* attributierten Vatermodulinstanz wird das Ausführungsrecht zugleich mit dem Auswahlrecht an alle Söhne weitergegeben.

- Eine mit *(system)-activity* attributierte Vatermodulinstanz fragt alle Sohnmodulinstanzen an, ob diese schaltbare Transitionen haben. Erst wenn alle geantwortet haben, erfolgt die Auswahl, und das Ausführungsrecht wird an eines der Kinder[1] weitergegeben.

[1] nicht-deterministisch

- Die Auswahl der schaltbaren Transitionen erfolgt innerhalb jeder Modulinstanz. Dabei wird die nicht-deterministische Auswahl zwischen schaltbaren Transitionen gleicher Priorität mit Hilfe eines Zufallgenerators nachgebildet.

- In Estelle erfolgt die Auswahl zu einem bestimmten Zeitpunkt, was insbesonders bedeutet, daß alle *delay-timer* seit der letzten Auswahl um den gleichen Wert dekrementiert wurden. Da hier die Auswahl auf ein Zeitintervall verteilt ist, muß zu einem bestimmten Zeitpunkt der zur Auswahl relevante Zustand ermittelt werden, auf dem dann die Auswahl des gesamten Subsystems erfolgt.

Ausführungsphase

Die Ausführung der ausgewählten Transitionen eines Subsystems und damit die Anwendung der Relation t auf die globale Momentbeschreibung gid_{SP} erfolgt nach dem folgenden Schema:

- Bei einer mit *systemprocess* oder *systemactivity* attributierten Modulinstanz, welche in der Auswahlphase eine schaltbare Transition hatte, wird diese unmittelbar ausgeführt.

- Erhält eine mit *process* attributierte Modulinstanz zur Auswahlphase das Ausführungsrecht, und besitzt eine schaltbare Transition, führt es diese unmittelbar aus.

- Nachdem eine mit *activity* attributierte Modulinstanz in der Auswahlphase signalisiert hat, eine schaltbare Transition zu besitzen, kann sie eine der schaltbaren Transitionen ausführen, wenn sie das Ausführungsrecht von der Vatermodulinstanz erhält.

- Jede Sohnmodulinstanz schickt nach Ausführung einer Transition eine Nachricht an die Vatermodulinstanz, welche das Ende der Ausführung signalisiert.

- Erhält eine Modulinstanz, die nicht mit *system* attributiert ist, von allen Sohnmodulinstanzen, denen sie das Ausführungsrecht gegeben hat die Nachricht, welche das Ende der Ausführung signalisiert, so gibt sie diese Nachricht an die Vatermodulinstanz weiter.

- Ein Subsystem beendet den Ausführungszyklus, wenn es eine schaltbare Transition ausgeführt hat oder von allen Sohnmodulinstanzen, denen es das Ausführungsrecht übergeben hatte, die Nachricht der Beendigung der Ausführung empfangen hat.

In Estelle erfolgt die Auswahl der schaltbaren Transitionen zu einem bestimmten Zeitpunkt, an den sich nach und nach die Ausführung dieser ausgewählten Transitionen anschließt. Die Ausführung einer Transition eines Subsystems darf demnach keine Auswirkung auf die Auswahl seiner anderen Transitionen in demselben Ausführungszyklus haben. Bei dem hier vorgegebenen Ausführungsmodell der Instanzen der *Estelle-Maschine* werden die beiden Phasen nicht zeitlich sequentiell ausgeführt. Aus diesem Grund müssen Maßnahmen getroffen werden, die sicherstellen, daß die Estelle-Semantik eingehalten wird. Das Resultat der Anwendung der Relation t auf gid_{SP} muß deshalb in einigen Fällen verzögert werden.

- Interaktionen werden nicht direkt beim Empfänger abgeliefert, sondern erst in einer temporären Warteschlange zwischengespeichert.

  - Interaktionen, die über die Vatermodulinstanz geschickt werden, werden in eine temporäre Warteschlange, die sogenannte *ascending_queue*, eingefügt.

  - Interaktionen, die für eine Sohnmodulinstanz bestimmt sind, werden in die *descending_queue*, die der jeweiligen Sohnmodulinstanz zugeordnet ist, eingefügt.

  - Die Interaktionen in der *ascending_queue* werden am Ende der Ausführungsphase jeweils mit der Nachricht, die das Ende der Ausführung signalisiert, an die Vatermodulinstanz weitergegeben. Dort werden sie in die entsprechende Warteschlange eingereit. Eine Interaktion wandert durch alle mit *attach* verbundenen Interaktionspunkte in der Modulhierachie nach oben.

  - Die Interaktionen in der *descending_queue* werden bei der nächsten Auswahlphase zusammen mit den Synchronisationsnachrichten an die Sohnmodulinstanzen weitergegeben. Sie gelangen damit rechtzeitig zur Auswahlphase bei der Modulinstanz an, für welche die Interaktion bestimmt ist.

  Damit erreicht eine Interaktion die Zielmodulinstanz frühestens zu Beginn des nächsten Ausführungszyklus des Subsystems. Zur Steigerung der Effizienz wurde das Verfahren in einer optimierten Version von Pet-Dingo derart verändert, daß Subsysteme direkt Interaktionen austauschen, und nicht über den inaktive Vater kommunizieren. Dadurch ist die Einhaltung der Estelle-Semantik aber nicht mehr sichergestellt, insbesonders kann die Kausalität verletzt werden[Br94].

- Kopien der exportierte Variablen werden jeweils mit den Synchronisationsnachrichten von Vater- und Sohnmodulinstanz verschickt.

  - Die Vatermodulinstanz schickt jeweils eine Kopie der exportierten Variablen mit der ersten Synchronisationsnachricht des aktuellen Ausführungzyklus zu den Sohnmodulinstanzen.
  - Die Sohnmodulinstanz schickt die Kopien der exportierten Variablen nach der Ausführung der Transition mit der das Ende der Ausführung signalisierenden Nachricht zur Vatermodulinstanz.

- Alle Estelle-Anweisungen, die die Kanal- bzw. Modulstruktur verändern[2], beziehen sich jeweils auf Sohnmodulinstanzen. Wenn sie ausgeführt werden, kann in keinem Fall eine Transition der Sohnmodulinstanz ausgeführt werden[3].

Durch diese Maßnahmen ist sichergestellt, daß die Ausführung von Transitionen die Auswahl von Transitionen im gleichen Ausführungzyklus desselben Subsystems nicht beeinflußt. Der Vorteil ist, daß in einem Ausführungzyklus Auswahl und Ausführung von Transitionen, welche nicht in einer Nachkommenschaftbeziehung stehen, parallel ausgeführt werden können.

Den Ausführungszyklus einer Instanz der *Estelle-Maschine*, die ein mit *system* attributiertes Modul repräsentiert, kann man in die folgenden drei Phasen unterteilen:

Phase 1 Die Ermittlung des zur Auswahl des Subsystems relevanten Zustandes. Dazu gehört die Ermittlung der vergangenen Zeit und der von außen eingegangenen Interaktionen.

Phase 2 Die Auswahl und Ausführung der schaltbaren Transitionen. Dabei werden die Interaktionen und exportierten Variablen mit den Synchronisationsnachrichten den entsprechenden Modulinstanzen zugestellt.

Phase 3 Das Senden der Interaktionen, deren Empfänger außerhalb des Subsystems lokalisiert sind. Dies erfolgt mit einer direkten Nachricht an das entsprechende Subsystem.

Alle anderen Instanzen der *Estelle-Maschine* besitzen keinen eigenen Ausführungszyklus. Sie befinden sich im wartenden Zustand und können nur auf Anforderung durch die Vatermodulinstanz eine Auswahl oder Ausführung von Transitionen vornehmen.

2.2.2 Beispiel

An einem Beispiel soll der Ablauf eines Ausführungzyklus demonstriert werden. Die Beispielspezifikation (Abbildung 2.2) besteht aus zwei Subsytemen, wobei

---
[2] connect,terminate,...
[3] Vater-Sohn Vorrangprinzip

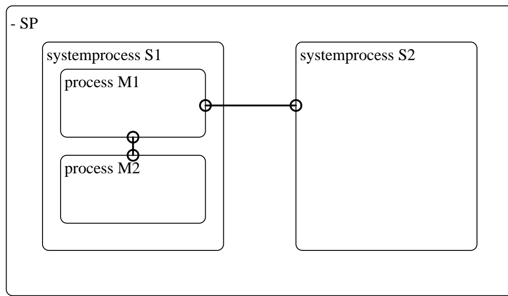

Abbildung 2.2: Beispielspezifikation

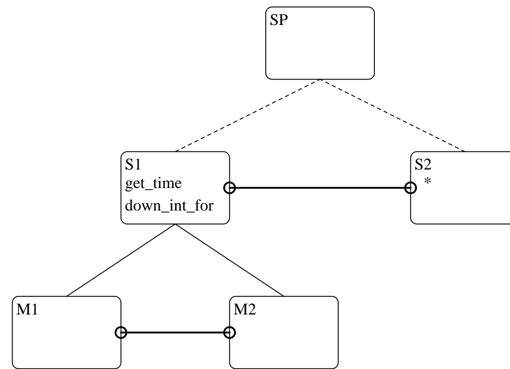

Abbildung 2.3: Phase 1

das eine Subsystem zwei Sohnmodulinstanzen besitzt, die mit *process* attributiert sind. Beide Subsysteme laufen vollständig asynchron. Es wird vom Ausführungszyklus des ersten Subsystem ausgegangen, das andere befindet sich in einer beliebigen Phase. Alle Modulinstanzen seien eigene Instanzen der Estelle-Maschine.

Phase 1 Die Kontrolle des Subsystem befindet sich bei der Modulinstanz S1. Die Modulinstanz S1 ermittelt die Zeit, um die die *delay-Timer* dekrementiert werden (Abbildung 2.3). Außerdem ermittelt sie alle seit dem letzten Ausführungszyklus durch den externen Interaktionspunkt angekommenen Interaktionen und fügt sie in die *descending_queue* ein.

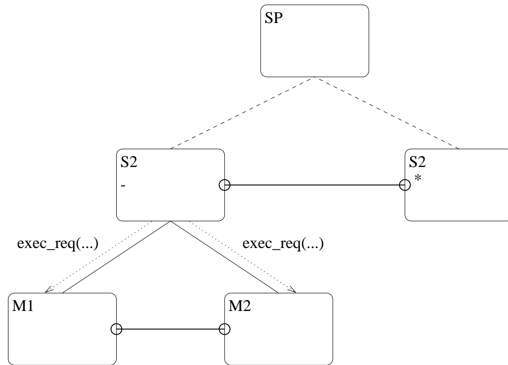

Abbildung 2.4: Phase 2a

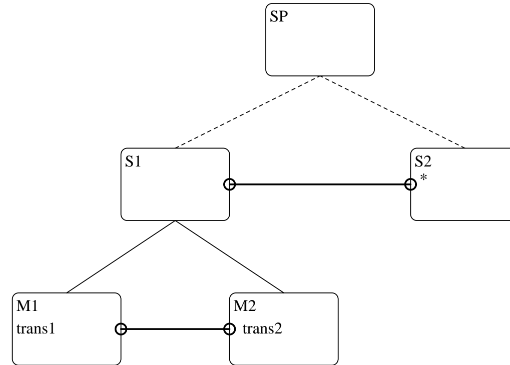

Abbildung 2.5: Phase 2b

Phase 2 Die Modulinstanz S1 ermittelt, ob sie schaltbare Transitionen besitzt. Da keine schaltbaren Transitionen vorhanden sind, gibt sie das Auswahlrecht und, wegen der *systemprocess*-Attributierung, auch das Ausführungsrecht an die Sohnmodulinstanzen M1 und M2 weiter. Dies erfolgt durch die Protokoll-Primitive *exec_request*, mit welcher auch die Interaktionen der *descending_queue*, die Kopien der exportierten Variablen und die vergangene Zeit übermittelt werden. Die Modulinstanzen M1 und M2 wählen

aus ihren schaltbaren Transitionen jeweils eine aus, die ausgeführt werden soll(Abbildung 2.4).

Die Modulinstanzen M1 und M2 führen jeweils ihre ausgewählte Transition aus (Abbildung 2.5). Die ausgewählte Transition *trans1* der Modulinstanz enthält jeweils eine Anweisung zur Versendung einer Interaktion an M2 und S2. Beide werden in die *ascending_queue* eingefügt. Die Auswahl und Ausführung der schaltbaren Transitionen der Modulinstanzen M1 und M2 erfolgt parallel. Nachdem die Modulinstanzen M1 und M2 ihre Transitionen ausgeführt haben, erfolgt die Rückgabe des Ausführungsrechts durch die Protokoll-Primitive *end_of_execution*. Mit dieser Nachricht wird der Inhalt der *ascending_queue* an die Vatermodulinstanz verschickt.(Abbildung 2.6).

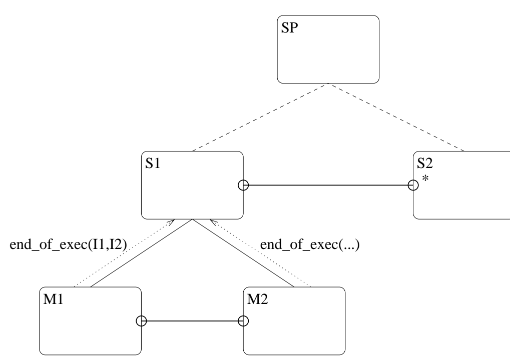

Abbildung 2.6: Phase 2c

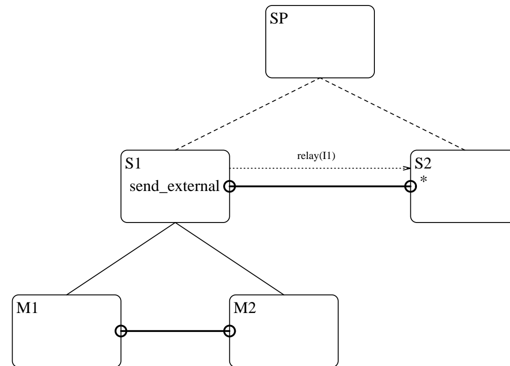

Abbildung 2.7: Phase 5

Phase 3 Das Subsystem S1 schickt die Interaktion I2 mit der Protokoll-Primitiven *relay* zu dem Subsystem S1(Abbildung 2.7).

2.3 Generierung der Implementierung

Es wird die Vorgehensweise des Pet-Dingo Entwicklungssystems beschrieben, wie ausgehend von einer Estelle-Spezifikation eine verteilte Implementierung generiert wird[SIST91A][SIST91B].

2.3.1 Implementierung des Ausführungsmodells

Pet-Dingo stellt keinen Interpreter zur Verfügung, auf dem die Spezifikation ausgeführt wird, sondern es wird ausführbarer Kode generiert, der eine Instanziierung der *Estelle-Maschine* mit der Spezifikation darstellt. Dabei ist das Ausführungsmodell der Instanzen der *Estelle-Maschine* in einer Menge von Basisklassen festgehalten. Für jede Attributierung eines Moduls existiert eine solche Basisklasse. Für

jede Modulinstanz wird das spezielle Verhalten durch Ableiten der entsprechenden Basisklasse und der Generierung virtueller Methoden erzeugt. Die virtuellen Methoden werden anhand der konkreten Spezifikation erstellt; sie beschreiben das spezielle Verhalten der Modulinstanz.

Die Basisklassen der Modulinstanzen enthalten im wesentlichen zwei Methoden. Die Methode *__startExec* ist eine Implementierung des Ausführungsmodells der Instanzen der *Estelle-Maschine*. Im Falle, daß es sich um ein Subsystem handelt, ist es eine Endlosschleife, in der die drei Phasen abgearbeitet werden. Falls Vater- und Sohnmodulinstanz verteilt vorliegen, kommunizieren diese entsprechend über ein Synchronisationsprotokoll miteinander. Diese Kommunikation wird dabei von der Vatermodulinstanz initiiert und geschieht in der Methode *__childrenExec*.

2.3.2 Übersetzung der Spezifikation

Der Übersetzungsvorgang ist beim Pet-Dingo Entwicklungspaket in mehrere Schritte unterteilt. Mit *Pet* wird die Estelle-Spezifikation, die als Textdatei vorliegt, in einen Zwischenkode umgewandelt, der eine kompakte Repräsentation der Spezifikation ist. *Pet* nimmt dabei eine Überprüfung der Syntax der Spezifikation vor. *Dingo* erhält als Eingabe diesen Zwischenkode und erzeugt anhand desselben C++-Kode.

Die Umwandlung dieses Zwischenkodes in C++-Kode erfolgt im wesentlichen über das Definieren von Klassen und Funktionen. Dabei ist der Generierungsvorgang in den meisten Fällen eine direkte Abbildung der Spezifikation in die Klassen und Funktionen. Im wesentlichen trägt dazu die Kapselung des verteilten Ausführungsmodells in die Basisklassen der Laufzeitbibliothek bei. Da Estelle auf dem Sprachumfang von Pascal aufgebaut wurde, muß eine Umwandlung von Pascal auf C++ erfolgen, die bis auf die Sichtbarkeitsregeln auch von einfacher Natur ist. Anhand eines Beispiels soll der Übersetzungsvorgang demonstriert werden.

Beispiel

Die Definition eines Moduls ist in Estelle zweigeteilt. Zum einen wird der Modulkopf definiert, der die Art des Moduls und deren externe Interaktionspunkte festlegt. Der andere Teil ist der Modulrumpf, vom dem auch mehrere verschiedene pro Modulkopf definiert sein können. In diesem werden die Transitionen, Zustände und Variablen definiert. Es soll hier gezeigt werden, wie dieses Modul von *Pet-Dingo* in C++-Kode umgesetzt wird.

```
MODULE user1Type SYSTEMPROCESS;
 IP userPM1: transmit(user) COMMON QUEUE;
```

```
    BODY user1_body FOR user1Type{#14};
     VAR count: INTEGER;
        Sound:INTEGER;
        sdu: Sdutype;

      STATE
         sending,
         ending;
      .....
```

Für die Definition des Modulkopfes wird eine Klasse generiert, welche von der Klasse __SysProcess[4] der Laufzeitbibliothek abgeleitet ist. Das zusätzliche Element UserPM1 ist vom Typ __SIPType_commonQ, welches einen Interaktionspunkt mit gemeinsamer Warteschlange repräsentiert. Das Verhalten einer Instanz dieser Klasse beschränkt sich auf das Verhalten der Klasse __SysProcess.

```
struct _User1Type : __SysProcess {
   __SIPType_commonQ UserPM1;


  void __setPars();
  void __buildExtIps();
  _User1Type( istream& is) { __readPars( is); __buildExtIps();}
  _User1Type() { __setPars(); __buildExtIps(); }
};
```

Für den Modulrumpf wird eine Klasse __MI_User1_body generiert, welche von der Klasse des Modulkopfes abgeleitet wurde. Es werden als weitere Elemente die Variablen des Modulrumpfs definiert, sowie ein Objekt einer Klasse, welches einen delay-Timer beschreibt. In den generierten Methoden ist das Verhalten des Modulrumpfs implementiert. Bei Aufruf der Methode __selAndExec werden beispielsweise die schaltbaren Transitionen der Modulinstanz ausgewählt und ausgeführt. __init führt zur Auswahl und Ausführung der init-Transition. __customStart dient der Initialisierung der Modulinstanz.

```
struct __MI_User1_body: _User1Type {
```

[4] Eine Ableitung der Klasse __MInstance

```
  __frame_User1_body __MIframe;
  __MI_User1_body_delays __timers;
  _Integer Count;
  _Integer Sound;
  _SduType Sdu;
  __MI_User1_body( istream& is) : _User1Type(is) { }
  __MI_User1_body() : _User1Type() { }
  __Interact* __buildInter( int code);
  char** __locVarNameList();
  void __showLocVar( ostream& os, int idx);
  char** __stateNameList();
  int __init();
  void __update( int __dt, int loc=1);
  int __selAndExec(int __dt);
  void __customStart();
  void __startExec( );
};
```

2.3.3 Erzeugung des ausführbaren Kodes

Der erzeugte C++-Kode wird auf die Rechner verteilt und dort jeweils zusammen mit den Pet-Dingo Laufzeitbibliotheken und den C++ Laufzeitbibliotheken übersetzt. Zum Ausführen muß auf jeder Maschine ein sogenannter *site-server* zur Verfügung stehen, welcher vorab gestartet wird und die Netzwerkadresse der anderen *site-server* mitgeteilt bekommt. Eine Spezifikation wird von einem dieser *site-server* gestartet und erzeugt, entsprechend der Spezifikation, die Subsysteme und eigenständigen Modulinstanzen auf den Rechnern als Betriebsystemprozesse.

Kapitel 3

Software-Monitor

Für die Durchführung einer Leistungsanalyse eines Systems gibt es verschiedene Methoden. Diese kann man im wesentlichen in drei Klassen einteilen, wobei jede dieser Klassen ein bevorzugtes Anwendungsgebiet besitzt[FER83][LAN92].

Bei der *analytischen Modellierung* wird ein Modell des zu untersuchenden Systems erstellt. Die Leistungsanalyse erfolgt mit Hilfe von mathematischen Methoden auf diesen Modellen. Ein Beispiel dafür ist die operationale Analyse auf einem Warteschlangenmodell. Das bevorzugte Anwendungsgebiet liegt bei der Entwicklung und dem Design von Systemen.

Ab einem gewissen Grad der Komplexität der Modellierung sind mathematische Methoden nicht, oder nur mit großem Aufwand, anwendbar. In diesem Fall kann mit Hilfe der *Simulation* eine Leistungsanalyse durchgeführt werden.

Die Ergebnisse der beiden angeführten Methoden sind abhängig von der Qualität der Modellierung; d. h. je mehr von der Realität abstrahiert wird, umso weniger sind die Ergebnisse auf die Realität übertragbar.

Bei den *Meßmethoden* werden Messungen direkt an dem ausführbaren System vorgenommen. Die so erlangten Informationen geben somit direkt die Realität wieder und erlauben die beste Leistungsanalyse eines Systems. Sie ist aber erst einsetzbar, wenn ein ablauffähiges System vorliegt, also erst in einer späten Phase des Entwicklungsprozesses eines Systems.

Es soll hier auf die *Meßmethoden* eingegangen werden, speziell darunter auf die Messung mit einem Software-Monitor. Dafür wird im ersten Abschnitt ein Meßmodell entwickelt. Dieses Meßmodell legt die Menge von Laufzeitinformation fest, aus denen eine Leistungsanalyse einer Protokollimplementierung erfolgen kann. Im darauffolgenden Abschnitt wird das Meßverfahren entwickelt. Es werden dafür verschiedene Meßmethoden vorgestellt und deren Vor- und Nachteile gezeigt. Der dritte Abschnitt zeigt, wie das Meßmodell mit dem ausgewählten Meßverfahren implementiert wurde.

3.1 Meßmodell

Aus der Problembeschreibung geht hervor, daß mit Hilfe der Leistungsanalyse zwei Ziele erreicht werden sollen. Es soll einerseits das Leistungsverhalten einer Protokollimplementierung beurteilt werden können. Andererseits sollen die Stellen des ausführbaren Kodes ermittelt werden, an denen die Möglichkeit bzw. Notwendigkeit einer Leistungssteigerung besteht.

Über die Leistung eines Systems kann direkt keine quantitative Aussage gemacht werden. Vielmehr ist es nötig, sogenannte Leistungsmaße zu definieren, anhand derer man ein System beurteilen kann. Diese Leistungsmaße ergeben sich dabei in direkter Abhängigkeit des zu beurteilenden Systems und dem Zweck der Leistungsanalyse.

In dem hier vorliegenden Fall handelt es sich bei dem System um eine Protokollimplementierung; allerdings werden zwei voneinander verschiedene Ziele verfolgt. Es bietet sich demnach an, die Leistungsmaße in zwei Klassen aufzuteilen.

- Bei der ersten Klasse handelt es sich um Leistungsmaße aus der Protokollwelt. Dazu zählen beispielweise die maximale Anzahl von Paketen, die an eine Partnerinstanz versendet werden können, oder die Verzögerung von Paketen bei der Verarbeitung durch eine Protokollinstanz.

- Die zweite Klasse besteht aus Leistungsmaßen, die bei der Leistungsanalyse von Programmen auftauchen. Dabei handelt es sich beispielsweise um die Ausführungshäufigkeit bzw. die Ausführungsdauer eines Programmsegments.

Wegen der Unterschiedlichkeit dieser Leistungsmaße soll für jede dieser beiden Klassen ein Meßmodell entwickelt werden, welches die zur Berechnung dieser notwendige Laufzeitinformation festlegt.

3.1.1 Meßmodell zur Leistungsanalyse von Protokollimplementierungen

Bevor die Definition des Meßmodells zur Ermittlung der Leistung einer Protokollinstanz erfolgen kann, müssen als erstes die Leistungsmaße definiert werden. Im allgemeinen sind Leistungsmaße aus der Welt der Protokolle aus Sicht des Dienstnutzers definiert, d. h. es wird die Leistung des Dienstes bewertet[VER89]. Die Leistung des Dienstes der Schicht N ergibt sich dabei aus der Leistung der Protokollinstanzen und der Leistung des Dienstes der Schicht N-1. Bei der Leistungsanalyse von Protokollimplementierungen soll aber möglichst die Leistung der Protokollimplementierung, unabhängig von dem verwendeten Basisdienst, ermittelt werden.

Dies hat zwei Gründe:

- Möchte man die generierte Protokollimplementierung als Prototypen verwenden, so werden die Protokollinstanzen auf einen realen Dienst aufgesetzt, der verschieden von dem in der Spezifikation spezifizierten Dienst[1] ist. Eine Leistungsanalyse, die auf den Leistungsmaßen aus Sicht der Dienstnutzer basiert, würde nicht direkten Rückschluß auf die Leistung der Protokollimplementierung liefern.

- Möchte man Protokollimplementierungen vergleichen, so geben die Leistungsmaße des Dienstes keinen direkten Aufschluß über die wirklichen Leistungsunterschiede der verschiedenen Protokollimplementierungen. Beispielweise ist eine Halbierung der Verzögerung durch die Protokollinstanz an der Dienstschnittstelle nicht meßbar, da die Verzögerung des darunterliegenden Dienstes immer mitgemessen wird.

Betrachtet man die Leistung einer Protokollinstanz, indem man vom Diensterbringer darunter abstrahiert, so kann man die folgenden Leistungsmaße definieren:

- Die *Verzögerung* eines Paketes durch eine Protokollinstanz ist das Zeitintervall, welches zwischen dem Empfangen eines Paketes über eine Schnittstelle und dem Versenden über eine, möglicherweise andere Schnittstelle vergeht.

- Der *Durchsatz* einer Protokollinstanz ist die Anzahl der Pakete, die eine Protokollinstanz pro Zeiteinheit verarbeitet. Dieses Leistungsmaß ist nicht unabhängig von der Umgebung bestimmbar, es ist nur der Durchsatz bei vorgegebenem Dienstnutzer und Basisdienst angebbar. Möchte man den maximalen Durchsatz bestimmen, so benötigt man einen Basisdienst, der eine höhere Kapazität als die Protokollinstanz besitzt, und einen Dienstnutzer, der diese Kapazität ausnutzt.

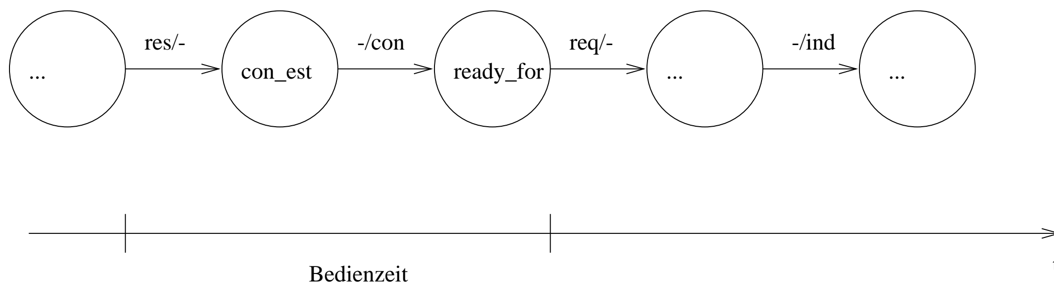

Abbildung 3.1: Bedienzeit

- Die *Bedienzeit* einer bestimmten Operation der Protokollinstanz ist das Zeitintervalls, welche zwischen zwei festgelegten Zuständen der Protokollinstanz liegt (siehe Abbildung 3.1). Die Festlegung der Zustände ist

[1]Dieser ist *nur* spezifiziert, um ein geschlossenes System zu erhalten

abhängig von dem zu messenden Protokoll. Bei einem verbindungsorientierten Protokoll kann es sich beispielsweise um das Zeitintervall, welches zwischen dem Empfangen eines connection.response und der Bereitschaft einen Data.request durch den Dienstnutzer entgegennehmen zu können handeln. Dieses Zeitintervall ist dann unabhängig von Dienstnutzer und Basisdienst, wenn zwischen den beiden Zuständen nur spontane Transitionen liegen, d. h. für keinen der Übergänge eine Interaktion an einer Schnittstelle anliegen muß.

Um diese Leistungsmaße ermitteln zu können, wird eine Menge von Laufzeitinformation benötigt. Dabei sollte es sich um eine möglichst kleine Menge von Laufzeitinformation handeln, aus Gründen die in Abschnitt 3.2 aufgezeigt werden. Für die oben angeführten Leistungsmaße kommt man dabei auf die folgenden Laufzeitinformationen:

Empfange Paket Der Zeitpunkt des Empfangens eines Paketes über eine Schnittstelle. In Estelle ist dies der Zeitpunkt, in dem eine Interaktion in einer Eingabewarteschlange eingereit wird.

Sende Paket Der Zeitpunkt des Sendens eines Paketes über eine Schnittstelle. In der Estelle-Semantik ist dies der Zeitpunkt der Anwendung der Funktion $transmission_P(gid_{SP})$, welche Interaktionen an die entsprechenden Eingabewarteschlangen der Zielmodulinstanz verteilt.

Kontrolle bei Bei den Zuständen, welche das Zeitintervall festlegen, handelt es sich nicht notwendigerweise um die Hauptzustände der Estelle-Spezifikation. Vielmehr können es beliebige Zustände des erweiterten Zustandraumes sein. Dieser erweiterte Zustandsraum einer Modulinstanz ändert sich nur bei der Ausführung der ausgewählten Transitionen. Um ein Zeitintervall festzulegen, müssen deshalb nur Zeitpunkte festgehalten werden, wenn der Kontrollfluß bestimmte Stellen in der Transitionsausführung passiert.

3.1.2 Meßmodell zur Leistungsanalyse des Ausführungmodells

Wie im letzten Abschnitt müssen auch hier Leistungsmaße definiert werden. Dabei sollen diese Leistungsmaße helfen bei der Entscheidung, an welchen Stellen der Protokollimplementierung eine Steigerung der Effizienz möglich bzw. notwendig ist.

Dafür muß zuerst beschrieben werden, durch welche Faktoren das Leistungsverhalten einer Protokollimplementierung festgelegt wird[GBE95].

1. Für ein vorgegebenes Protokoll kann man i. allg. mehr als eine Estelle-Spezifikation erstellen, die dasselbe, nach außen sichtbare, funktionale Verhalten beschreibt. Das zeitliche Verhalten der automatisch generierten Protokollimplementierungen dieses unterschiedlich spezifizierten Protokolls kann sich aber sehr wohl voneinander unterscheiden. Beispielsweise kann das Leistungsverhalten einer Implementierung von der Anzahl der Transitionen abhängig sein, die vom Empfangen eines Paketes vom Dienstnutzer bis zur Weitergabe an den Basisdienst ausgeführt werden.

2. Die Implementierung des verteilten Ausführungmodells, und damit der *Estelle-Maschine* ist von entscheidender Bedeutung für das Leistungsverhalten einer Implementierung. Dieses Leistungsverhalten wird zu einem großen Teil von der Implementierung des Synchronisationsprotokolls beeinflußt. Je mehr der verfügbaren Rechenzeit für die Synchronisierung der verteilten Modulinstanzen verbraucht wird, umso geringer ist die Leistung einer Protokollimplementierung.

3. Die Effizienz des erzeugte Kodes stellt einen weiteren Faktor dar. Die einfachen Übertragung der Pascalteile der Spezifikation in einen C++-Kode ist dabei weniger kritisch als die Methoden, die zur Auswahl und Ausführung der Transitionen generiert werden. Durch geschickte Erzeugung der Methode für die Auswahl kann die Rechenzeit zur Ermittlung der schaltbaren Transition möglichst verringert werden.

4. Die eingebundenen Laufzeitbibliotheken des C++ Kompilers haben einen weiteren Einfluß auf das Leistungsverhalten. Beispielweise kann eine ineffiziente Implementierung einer Klasse in den Laufzeitbibliotheken, die von der Implementierung benutzt wird, zu erheblichen Laufzeiteinbußen der Implementierung führen.

5. Die Effizienz des durch den C++-Kompiler generierten ausführbaren Kode ist nicht unerheblich für das erbrachte Leistungsverhalten. Eine Optimierung durch den Kompiler kann, je nach Kode, zu nicht unerheblichen Leistungssteigerungen führen.

6. Als letzten Faktor muß man den bzw. die Rechner und das Kommunikationssystem in Betracht ziehen, auf dem die Protokollimplementierung abläuft.

Es soll sich hier auf die ersten drei Faktoren beschränkt werden; die anderen Faktoren werden als fest und vorgegeben angesehen. Dies liegt darin begründet, daß die letzteren Faktoren das Leistungsverhalten eine Handimplementierung gleichermaßen beeinflussen.

Bei der Bewertung der Leistung handelt es sich demnach im wesentlichen um die Bewertung der *Estelle-Maschine*, sowohl des Konzeptes als auch deren

Implementierung. Die Spezifikation geht derart in das Leistungsverhalten ein, inwieweit diese auf die vorgegebene *Estelle-Maschine* abgestimmt ist.

Die Leistungsmaße müssen einen Einblick in die Effizienz des erzeugten Kodes geben. Dies führt zu den folgenden Leistungsmaßen:

- Die *Ausführungsdauer* in der die *Estelle-Maschine* eine Operation ausführt.
- Die *Ausführungshäufigkeit* einer bestimmten Operation der *Estelle-Maschine*.
- Die *Verzögerung* zwischen der Ausführung von zwei Operationen.

Um diese Leistungsmaße bestimmen zu können, benötigt man als Laufzeitinformation den Zeitpunkt, sowie die Dauer der Ausführung einer Operation. Wegen der i. allg. großen Menge von Operationen, soll das Meßmodell in einzelne Ebenen unterteilt werden, um so ein systematisches Vorgehen zu ermöglichen. Zur Festlegung dieser Ebenen wird nach dem Prinzip der Verfeinerung vorgegangen.

Dafür wird betrachtet, wie die Spezifikation auf der Estelle-Maschine ausgeführt wird. Die einzelnen Instanzen der Estelle-Maschine führen die Subsysteme aus, wobei die Ausführung sich in die drei Phasen aufteilt. Die Ausführung von Phase 2 unterteilt sich wiederum in Auswahl und Ausführung von Transitionen der Modulinstanzen des Subsystems. Die Ausführung einer Transition ist das Ausführen der Anweisungen des Transitionsblock.

Die einzelnen Ebenen sollen als Spezifikationsebene, Subsystemebene, Modulebene und Transitionsebene bezeichnet werden. Die Operationen in diesen Ebenen sollen im einzelnen aufgezählt werden.

Spezifikationsebene

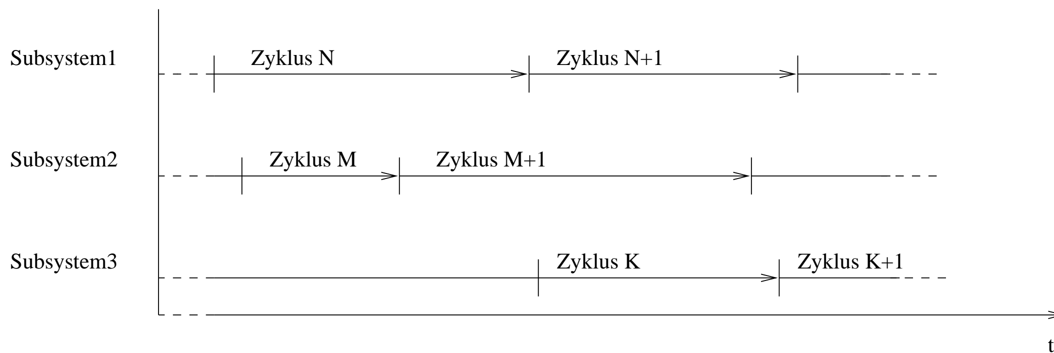

Abbildung 3.2: Spezifikationsebene

Auf der Spezifikationsebene besteht eine Sicht auf asynchron voneinander ausführender, miteinander kommunizierender Subsysteme. Auf dieser Ebene ist nur eine Operation sichtbar:

Zyklus Jedes der Subsysteme befindet sich zu jedem Zeitpunkt in einem Ausführungszyklus.

Subsystemebene

Die Ausführung eines Subsystem, und damit ein Ausführungszyzklus gliedert sich in drei Phasen. Als weitere Laufzeitinformationen, die in dieser Ebene von Interesse sind, kann man das Versenden und Empfangen von Interaktionen, und die Größe der Eingabewarteschlange während der Ausführung sehen. Es ergeben sich also die folgenden Operation, deren Laufzeitinformation erlangt werden muß:

Subsystem1 Phase 1 Phase 2 Phase 3 Phase 1 Phase 2 Phase 3

Subsystem1 Phase 1 Phase 2 Phase 3 Phase 1 Phase 2 Phase 3

t

Sende Interaktion Empfange Interaktion Warteschlange

Abbildung 3.3: Subsystemebene

Phase 1 In der Phase 1 ermittelt das Subsystem den Zustand, auf dem die folgende Auswahl erfolgt.

Empfange Interaktion Es werden die von anderen Subsystemen empfangenen Interaktionen in die Warteschlange eingereit. Dabei ist der Typ der empfangenen Interaktion von Interesse, nicht die Dauer der Operation.

Phase 2 In dieser Phase wird die Menge der schaltbaren Transitionen ermittelt und ausgeführt.

Warteschlange Die Größe einer Eingabewarteschlange *vor* der Auswahl der schaltbaren Transitionen wird festgehalten. Es handelt sich um keine Operation, sondern um eine Zustandsinformation.

Phase 3 In dieser Phase werden die Interaktionen verschickt, welche für andere Subsysteme bestimmt sind.

Sende Interaktion Die für andere Subsysteme bestimmten Interaktionen werden verschickt. Die übrigen Eigenschaften sind analog zu *Empfange Interaktion.*

Modulebene

Die Ausführung einer Transition folgt unmittelbar der Auswahl dieser schaltbaren Transition. Dabei kann dies, je nach Attributierung und Verteilung, parallel in den Modulinstanzen eines Subsystems geschehen. Die Operationen der *Estelle-Maschine* auf dieser Ebene sind die folgenden(siehe Abbildung 3.4):

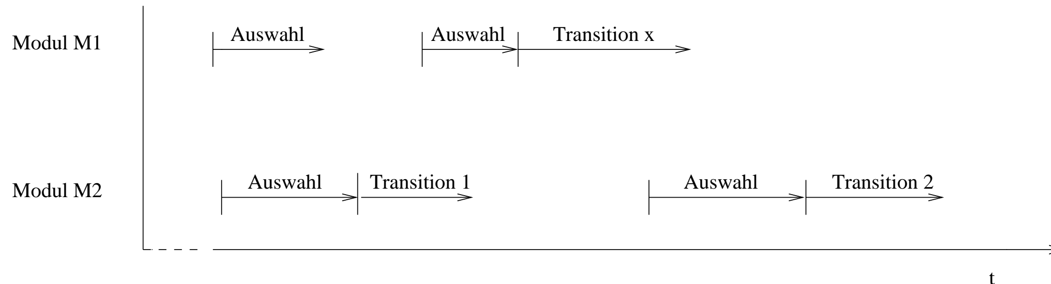

Abbildung 3.4: Modulebene

Auswahl Die Auswahl der schaltbaren Transition einer Modulinstanz.

Transition Die Ausführung der ausgewählten Transition.

Transitionsebene

Die Ausführung einer Transition führt zum Ausführen der Operationen, die im Transitionsblock spezifiziert sind. Diese lassen sich in drei Klassen unterteilen:

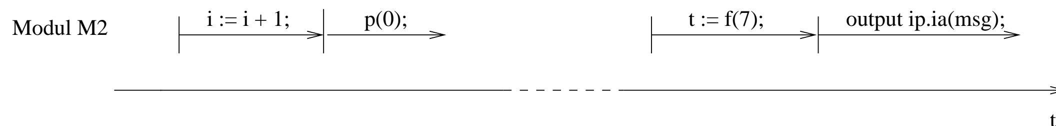

Abbildung 3.5: Transitionsebene

Anweisung Das Ausführen einer Estelle-Anweisung

Zuweisung Das Ausführen einer Zuweisung.

Prozedur Das Ausführen einer Prozedur.

3.2 Meßverfahren

Für das Ermitteln der Laufzeitinformation der beiden in Abschnitt 3.1 beschriebenen Meßmodelle wird ein Meßverfahren entwickelt. Dabei zeigt es sich, daß

bei der Entwicklung dieses Meßverfahrens einige Aspekte in Betracht gezogen werden müssen, um aussagekräftige Laufzeitinformationen zu erhalten. Bei dem verwendeten Meßverfahren soll es sich um einen Software-Monitor handeln.

3.2.1 Aufbau eines Software-Monitors

Bei einem Software-Monitor handelt es sich um ein Werkzeug, welches entweder als Modifikation eines Programmes oder als Bestandteil des Betriebsystems Laufzeitinformation über ein Softwareobjekt liefert. Dabei kann man den allgemeinen Aufbau eines Monitors als ein Schichtenmodell (Abbildung 3.6) wiedergeben, wobei jede der Schichten eine Funktionalität repräsentiert[LAN92].

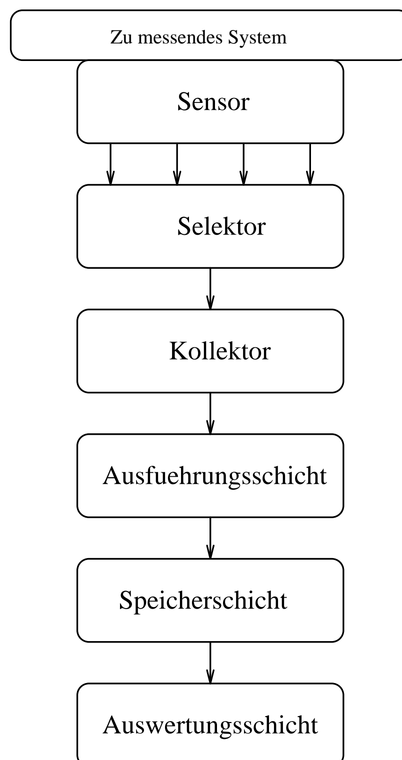

Abbildung 3.6: Schichtenmodell

- Der *Sensor* ermittelt die Meßgrößen des Systems. Dabei kann es sich um das Auftreten eines Ereignisses, oder aber um den Wert bestimmter Variablen zu einem Zeitpunkt handeln.
- Der *Selektor* wählt unter den ermittelten Laufzeitinformation diejenigen aus, welche für die aktuelle Messung von Interesse sind. Es handelt sich also um eine Art von Filter, der nur einen vorher festgelegten Teil von Meßgrößen an die nächste Schicht weitergibt.

- Durch den *Kollektor* können die ermittelten Laufzeitinformationen weiter verdichtet werden, um so die Datenmenge zu reduzieren. Beispielweise kann man aus einer Menge von Laufzeitinformation den Mittelwert ermitteln, und nur diesen festhalten.
- Die Funktionalität der *Ausführungsschicht* ist das Umwandeln der ermittelten Laufzeitinformation in ein für den Benutzer lesbares Format.
- Die ermittelten Informationen werden durch die *Speicherschicht* auf einem sicheren Medium festgehalten.
- Die Auswertung der ermittelten Laufzeitinformation erfolgt durch die *Auswertungsschicht*.

Ein Monitor muß nicht alle diese Schichten besitzen. Zum einen kann die Anordnung der Schichten in ihrer zeitlichen Abfolge vertauscht werden, zum anderen müssen nicht alle Schichten und damit Funktionalitäten vorhanden sein. Sowohl das eine als auch das andere erweist sich in manchen Fällen als vorteilhaft, da ein Monitor den folgenden Forderungen genügen muß[FER83]:

1. Über die durch den Software-Monitor ermittelten Laufzeitinformation muß genug Wissen vorhanden sein, um diese für die Analyse verwenden zu können.
2. Die Implementierung des Software-Monitor soll eine möglichst geringe Änderung der Laufzeitumgebung und des zu messenden Systems erfordern.
3. Die Interferenz, i. e. die Beeinflussung des Laufzeitverhaltens durch den Software-Monitor, muß möglichst minimal sein.
4. Der Software-Monitor sollte möglichst wenig Speicher zur Laufzeit benötigen.

3.2.2 Meßmethoden

Es gibt im wesentlichen zwei Ansätze, um an Laufzeitinformation heranzukommen. Zum einen kann man den Monitor aktivieren, wenn bestimmte Ereignisse stattfinden, wobei ein Ereignis als Transition eines Systemzustands in einen anderen definiert ist. Zum anderen kann der Monitor so implementiert werden, daß er zu bestimmten Zeitpunkten aktiviert wird, um den aktuellen Zustand aufzuzeichnen. Diese beiden Methoden sollen hier kurz gegenübergestellt werden.

Ereignisabhängige Monitore

Ein ereignisabhängiger Monitor führt bei Auftreten von vorgegebenen Ereignissen eine gleichermaßen vorgegebene Aktion aus. Bei dieser Aktion kann es sich im einfachsten Fall um das Hochzählen eines Zählers handeln, womit man eine Aussage über die Zahl des Eintretens des Ereignisses erhält. Es kann aber auch der Zeitpunkt des Auftretens festgehalten werden, sodaß man den zeitlichen Ablauf des Systems bezüglich dieses Ereignisses rekonstruieren kann. Weiter können auch Informationen über den Zustand des Systems zum Ereigniszeitpunkt festgehalten werden.

Bei einem Software-Monitor werden an den Stellen des Kodes, an denen das Ereignis auftritt, Meßpunkte in Form von Kode eingefügt. Tritt das Ereignis auf, passiert der Kontrollfluß des Systems den Meßpunkt und es wird die festgelegte Aktion ausgeführt.

Bei dieser Vorgehensweise sind einige Aspekte zu beachten:

- Die Interferenz ist direkt abhängig von der Anzahl der Meßpunkte. Dies bedeutet, daß je mehr Meßpunkte gesetzt werden, und je höher die Frequenz des Auftretens der Ereignisse an diesen Meßpunkten ist, umso größer ist die Beeinflussung des zeitliche Verhalten des zu messenden Systems durch den Monitor. Dies ergibt sich daraus, daß der eingefügte Kode i. allg. vom selben Prozessor ausgeführt wird wie das zu messende System.

- Da die Anzahl der gemessenen Ereignisse eine dynamische Größe ist, muß, im Falle der Ereignisaufzeichnung, entweder ein sehr großer Puffer zur Verfügung stehen oder die Daten zur Laufzeit auf Sekundärspeicher ausgelagert werden. Dabei kann ersteres zu einer Konkurrenzsituation mit dem System bzgl. des Speichers führen. Das Auslagern auf Sekundärspeicher kann, je nach System, zu einer größeren Beeinflussung des Laufzeitverhalten des Systems führen, falls der Prozessor für diesen Vorgang benötigt wird.

Ereignisunabhängige Monitore

Ein ereignisunabhängiger Monitor hält zu bestimmten Zeitpunkten die Informationen des aktuellen Zustands fest, die von Interesse sind. Es können bestimmte Werte festgehalten werden, oder aber der aktuelle Zustand selbst. Man benötigt eine große Anzahl von Laufzeitinformationen, um eine Aussage über eine System machen zu können, da es sich bei jeder ermittelten Meßgröße um eine Stichprobe handelt. Hat man eine große Zahl von Stichproben, so kann man mit statistischen Methoden Aussagen über das Laufzeitverhalten eines System machen. Die Implementierung eines solchen Software-Monitors erfolgt über Interrupts, die durch eine Uhr in regelmäßigen Abständen, dem Meßintervall, aktiviert werden. Folgende Aspekte müssen aber in Betracht gezogen werden[FER83]:

- Man kann nur über die Dauer eines Zustandes eine Aussage machen, wenn der Zeitpunkt des Ereignisses unabhängig vom Meßintervall ist. Wenn beispielweise zwischen dem Zeitpunkt des Zustandsübergang und dem Zeitpunkt der Messung ein zeitlicher Zusammenhang besteht, so erhält man eine Aussage, die vollkommen wertlos ist.

- Die Interferenz ist bei dieser Methode abhängig von der Größe des Meßintervalls und der Länge der Messung, und damit unabhängig von der Anzahl der Ereignisse. Bei einer vorgegebenen Interferenz ist die Größe des Meßintervalls damit nach unten beschränkt.

- Für eine Aussage über einmalige Ereignisse benötigt man eine große Zahl von Meßläufen.

Zeit

Beide Arten von Monitoren gehen davon aus, daß das System eine Uhr zur Verfügung stellt, mittels der die aktuelle Zeit festgestellt werden kann. Beim ereignisabhängigen Monitor benötigt man, falls eine Ereignisaufzeichnung erfolgt, diese Uhr, um ein Ereignis mit einem Zeitstempel zu versehen. Ein ereignisunabhängiger Monitor benötigt ohnehin eine Uhr.

In Mehrprozessystemen ist das Ermitteln der aktuellen Zeit nicht problemlos. Dabei sind es insbesonders zwei Dinge, die den Umgang mit der Zeit erschweren. Zum einen ist die Synchronität zweier räumlich verteilter Uhren nur bis zu einer gewissen Grenze erreichbar[LA90]. Man kann also nicht davon ausgehen, daß zwei Ereignisse, die räumlich verteilt auftreten und denen jeweils ein Zeitstempel zugeordnet wird, sich wirklich in der sich daraus ergebenen Reihenfolge ereignet haben. Zum anderen hängt das Zeitintervall zwischen zwei Ereignissen eines Prozesses nicht nur von den Ausführungszeiten der Anweisungen dieses Prozesses ab, sondern kann auch durch Unterbrechungen und Prozesswechsel vergrößert werden.

Das erste Problem der synchronen Uhren kann nur bis auf einen Fehler gelöst werden. Für das zweite Problem wurde in Mehrprozessystemen die sogenannte virtuelle, software oder Prozeßzeit eingeführt[FER83]. Bei dieser Zeit handelt es sich um einen Zähler, der jedem Prozeß zugeteilt ist und zu bestimmten Zeiten vorgeschaltet wird, wenn der Prozessor sich in Besitz des Prozesses befindet. Um entscheiden zu können, welche der Zeiten für den Software-Monitor vorteilhaft sind, sollen die Vor- und Nachteile der beiden Zeiten im einzelnen aufgezeigt werden.

Realzeit Der Zugriff auf die Realzeit erfolgt bei den meisten Systemen über den Zugriff auf ein Register einer Hardware-Uhr. Die Auflösung ist auf den gängigen Maschinen im Bereich von μs. Für das Feststellen der zeitlichen Abfolge von

Ereignissen ist diese Zeit dann einsetzbar, wenn der minimale Abstand zweier Ereignisse größer als die Auflösung der Zeit ist. Wenn dies erfüllt ist, hat man für Ereignisse, welche dieselbe Uhr zur Ermittlung des Zeitstempels verwendet haben, eine eindeutig kausale Ordnung. Über Rechnergrenzen kann allerdings wegen der fehlenden absoluten Synchronität der Uhren die Kausalität verletzt werden. Es kann demnach innerhalb der Fehlerschranke der Synchronität keine Aussage über die zeitliche Abfolge von Ereignissen gemacht werden.

Ein weiteres Problem bei der Messung mit dieser Zeit ist, daß die wirkliche Rechenzeit zwischen zwei Ereignissen nicht bestimmt werden kann, da möglicherweise Unterbrechungen und Prozesswechsel innerhalb dieses Intervalls stattfinden.

Prozeßzeit Die Prozeßzeit hat in den meisten Betriebssystemen eine um Größenordnungen grobere Auflösung als die Realzeit. Sie bewegt sich auf gängigen Systemen im Bereich von ms. Benötigt man eine genauere Aussage über die Zeitdauer eines mit dieser Zeit festgehaltenen Zeitintervalls, so kann dies über eine statistische Auswertung einer großen Zahl von festgehaltenen Zeitintervallen erfolgen.

Es läßt sich auch nur schwer überprüfen, inwieweit das Betriebssystem diese Zeit korrekt verwaltet, d. h. die Zeitticks dem richtigen Prozeß zuordnet [FER83]. Beispielsweise muß festgelegt sein, welchem Prozeß der Zeittick zugeordnet wird, wenn gerade eine Unterbrechung abgearbeitet wird. Ein weiteres Problem ist die mögliche Korrelation des Vorschaltens der Prozeßzeit mit den zu messenden Ereignissen. In diesem Fall erhält man trotz statistischer Auswertung falsche Werte.

3.2.3 Entwickeltes Meßverfahren

Ausgehend von dem Meßmodell und den Meßmethoden soll ein Meßverfahren entwickelt werden, welches den Forderungen aus Abschnitt 3.2.1 genügt.

Für das Meßverfahren wird das Konzept eines ereignisabhängigen Monitors mit Ereignisaufzeichnung gewählt. Dies erscheint deshalb als sinnvoll, da für das erste Meßmodell die Ereigniszeitpunkte von Wichtigkeit sind, welche mit dem ereignisunabhängigen Monitor nicht feststellbar sind. Für das zweite Meßmodell wäre auch der ereignisunabhängige Monitor denkbar, allerdings wären damit zum einen nur über die Dauer der Intervalle eine Aussage zu machen, zum anderen wäre der Meßaufwand für einmalige, kurze Intervalle bedeutend größer.

Für die Ereignisaufzeichnug benötigt man eine Uhr, auf die zugegriffen werden kann und die aktuelle Zeit ermittelt werden kann. Es werden dafür sowohl Realzeit als auch Prozeßzeit verwendet, da je nach Ziel der Messung unterschiedliche Aspekte dieser Zeiten benötigt werden. Die Realzeit ist unumgänglich, da nur mit ihr eine zeitliche Ordnung der gemessenen Ereignisse bei parallel abarbeitenden Prozessen gegeben ist. Die Prozeßzeit ist nur innerhalb eines Prozesses

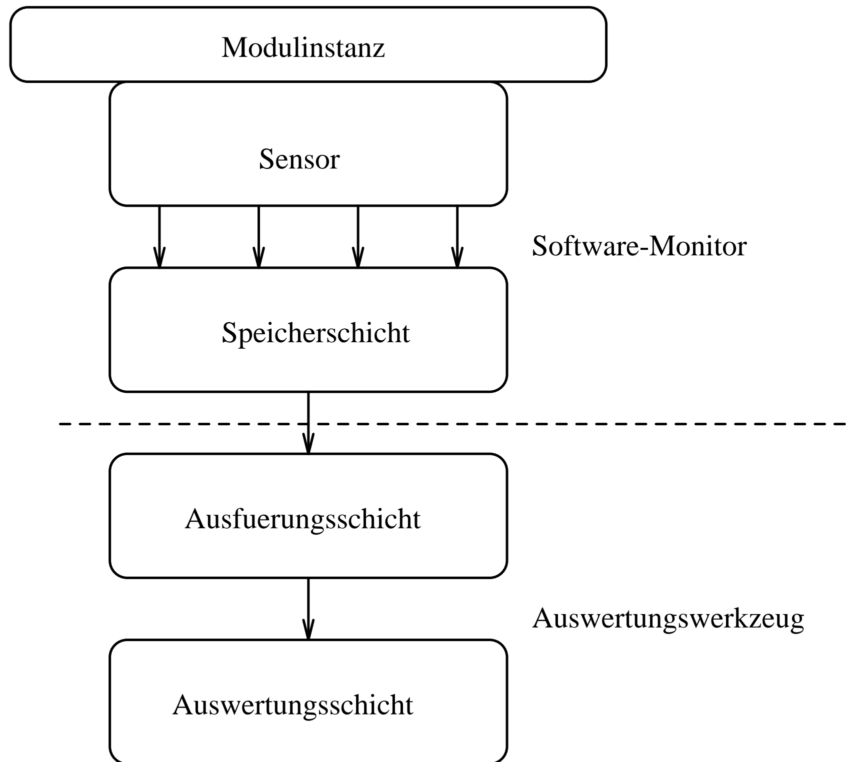

Abbildung 3.7: Schichtenmodell des Meßverfahrens

gültig, und bietet überdies eine zu geringe Auflösung um die sequentielle Abfolge der Ereignisse festlegen zu können. Andererseits treten bei der Realzeit die oben angeführten Probleme mit Unterbrechungen und Kontextwechseln auf. Um bei der Bestimmung der verbrauchten Rechenzeit von größeren Intervallen eine bessere Aussage zu bekommen, benötigt man demnach die Prozeßzeit .

Die Auswahl der Meßpunkte, d. h. die Entscheidung welche Ereignisse aufgezeichnet werden sollen, erfolgt zur Generierungszeit. Damit entfällt die Selektor-Schicht des Schichtenmodells eines Software-Monitors (siehe Abbildung 3.7). Zugleich werden alle zur späteren Auswertung notwendigen Informationen über die Ereignisse, welche nicht erst zur Laufzeit verfügbar sind, vorab auf eine Datei geschrieben. Es muß demnach schon zur Generierungszeit des zu messenden Systems feststehen, welche Ereignisse für die Leistungsanalyse von Interesse sind. Diese Vorgehensweise wurde gewählt, um die zusätzliche Interferenz, die durch die Auswahl zur Laufzeit gegeben wäre, zu vermeiden.

Die gemessenen Ereignisse werden nicht verdichtet, sondern direkt in einen Puffer abgelegt. Somit fehlt auch die Kollektor-Schicht des Schichtenmodells. Bei diesem Puffer handelt es sich um einen statischen Puffer, d. h. der Speicherplatz für diesen Puffer wird am Anfang der Messung angefordert und die Größe des Puffers kann während der Messung nicht verändern werden. Reicht dieser Puffer nicht zur Aufnahme aller Meßereignisse aus, so werden die ältesten Meßer-

eignisse überschrieben. Der Inhalt des Puffers wird am Ende der Messung als Datei abgespeichert. Diese Maßnahmen sind notwendig, da es sich sowohl bei der dynamische Speicherverwaltung als auch bei der Ausgabe auf eine Datei um zeitaufwendige und zeitdauerunbestimmte Vorgänge handelt. D. h. es würde sich zum einen die Interferenz beträchtlich erhöhen, zum anderen könnte man bei der Auswertung nicht die Interferenz bestimmen. Somit unterbleibt jede dynamische Speicherverwaltung seitens des Software-Monitors zur Meßzeit.

3.3 Implementierung des Software-Monitors

Bei der Implementierung des Meßverfahrens wurden einige Aspekte beachtet, um denen in Kapitel 1, sowie in Abschnitt 3.2.1 gestellten Anforderungen gerecht zu werden.

- Damit der Monitor leicht auf andere Kode-Generatoren übertragen werden kann, wurde die gesamte "Meßtechnik" in Klassen gekapselt. Diese Klassen können ohne Änderungen in beliebigen C++ Programmen zur Messung verwendet werden. Gleichzeitig bilden diese Klassen die Schnittstelle zu dem gleichfalls erstellten Auswertungswerkzeug.

- Bei der Implementierung der Klassen wurde darauf geachtet, daß sich durch die Kapselung keine Einbußen der Effizienz ergeben. Dafür wurde auf das Konzept der "inline"-Methoden zurückgegriffen. Die dynamische Speicherverwaltung wurde teilweise explizit ausgeführt, um die ineffizienten Vorgehensweise der Initialisierung zu umgehen.

- Es wurden nur minimal Änderungen an den Laufzeitbibliotheken vorgenommen, um einerseits eine gute Übertragbarkeit auf zukünftige Versionen zu gewährleisten. Andererseits konnte so die Interferenz soweit wie möglich klein gehalten werden.

- Am Kode-Generator selbst wurden viele Änderung vorgenommen, um die Meßpunkte automatisch generieren zu können. Dabei sind alle Änderungen, die den Software-Monitor betreffen durch eine Präprozessordirektive zu deaktivieren, womit man die ursprüngliche Version erhält.

Die Implementierung des Software-Monitors soll in den nächsten Abschnitten beschrieben werden.

3.3.1 Klassen des Software-Monitors

Für die Implementierung des Software-Monitors wurden einige Klassen erstellt, die hier im einzelnen aufgeführt werden.

time_s Ein Objekt dieser Klasse stellt einen Zeitpunkt dar. Die Klasse stellt Methoden zur Darstellung, zur Arithmetik und zum Vergleich von Objekten zur Verfügung. Zusätzlich existieren Methoden, welche einem Objekt den aktuellen Zeitstempel in Realzeit oder Prozeßzeit zuordnen.

monitor_stamp Ein Objekt der Klasse monitor_stamp stellt ein Meßereignis dar. Die Klasse enthält jeweils zwei Elemente für das Aufzeichnen der Zeitpunkte in Real- und Prozesszeit. Außerdem enthält sie ein Element *key*, das den Meßpunkt eindeutig identifiziert und zwei anderen Elemente, die jeweils einen numerischer Wert aufnehmen können. Zum Setzen der Elemente enthält die Klasse Methoden, die alle "inline" definiert sind. Außerdem sind Methoden definiert, die das Schreiben auf und das Lesen von einer Datei ermöglichen.

monitor_log Die Klasse monitor_log beschreibt den Puffer, in dem die Meßereignisse zur Laufzeit gespeichert werden. Der Puffer ist dabei als ein Feld von Objekten der Klasse *monitor_stamp* organisiert. Es steht eine Methode zur Verfügung, welche die Initialisierung des Puffers zu Beginn der Messung ermöglicht. Bei der Ausführung des Destruktors wird der Puffer automatisch in eine Datei gerettet. Zur Steigerung der Effizienz existiert eine Methode, welche einen Zeiger auf den jeweils nächsten freien bzw. längsten unbenutzten Pufferplatz zurückgibt. Damit ist es möglich, daß zur Meßzeit keine temporären Objekte der Klasse *monitor_stamp* angelegt werden und das Kopieren von Objekten vermieden werden kann. Die Methoden zur Ereignisaufzeichnung können damit direkt auf dem vorher angelegten Puffer erfolgen.

monitor_info Ein Objekt dieser Klasse dient zur Aufnahme der statischen Information eines Meßpunktes. Bei dieser Information handelt es sich um den Typ des Ereignisses und ein bzw. zwei Strings, die den Meßpunkt näher beschreiben.

monitor_db Diese Klasse beschreibt eine Datenbank, die jeder Schlüsselnummer eines Meßpunktes die zugehörige statische Information zuordnet.

monitor Ein Objekt dieser Klasse enthält alle relevanten Informationen, um die Meßpunkte setzen zu können. Dafür enthält es die Information, welche Ereignisse gemessen werden sollen. Es sind alle Methoden definiert, die das Setzen eines Meßpunktes bei der Generierung der Implementierung ermöglichen. Der Aufruf dieser Methoden führt aber nur dann zur Generierung der Meßpunkte, wenn die betreffenden Meßpunkte vorher als zu Setzende ausgewählt wurden. Für jeden gesetzten Meßpunkt können die statischen Informationen auf eine Datei geschrieben.

3.3.2 Änderungen des site_server

Bei Pet-Dingo erfolgt das Beenden einer Ausführung einer Implementierung durch das Senden eines Signals an alle Prozesse durch den *site_server*. Dabei wurde das Signal gewählt[2], welches zu einem sofortigen Abbruch führt und nicht abgefangen werden kann. Dies hat zur Folge, daß keine Destruktoren aufgerufen werden, um die Objekte zu deinitialisieren. Da zum Ende der Messung, welches bei dem vorgegeben Ansatz mit dem Ende der Ausführung zusammenfällt, der Meßpuffer auf Datei abgespeichert werden muß, wird das Signal durch ein solches ersetzt, welches durch eine Interruptroutine abgefangen werden kann. In diesem Fall wurde SIGUSR2 gewählt, da dieses nicht durch das Betriebsystem mit anderen Aufgaben belegt ist.

3.3.3 Änderungen der Laufzeitbibliotheken

In den Laufzeitbibliotheken wurden sowenig wie möglich Änderungen vorgenommen, um so eine möglichst leichte Anpassung des Software-Monitors an Folgeversionen zu ermöglichen.

- Die Klasse __MInstance, welche die Basisklasse für alle attributierten Module ist, wurde als abgeleitete Klasse von *monitor_log* definiert. Damit existiert ein Zeiger auf den Meßpuffer, wo immer auf ein Objekt der Klasse __MInstance zugegriffen werden kann. Bei dem generierten Kode ist in allen generierten Funktionen eine Referenz auf die Modulinstanz, und somit der Zugriff auf den Meßpuffer gegeben. Als zweiter Vorteil ergibt sich, daß automatisch jeder Modulinstanz ein eigener Meßpuffer zugeordnet wird und zur Meßzeit nicht mehr festgestellt werden muß, in welcher Modulinstanz das Ereignis stattgefunden hat, sondern dies implizit vorgegeben ist.

- Für die Klasse __MInstance mußte der Destruktor entsprechend verändert werden, damit bei einer Deinitialisierung der Vatermodulinstanz auch alle Sohnmodulinstanzen deinitialisiert werden. Dazu mußte zusätzlich der Destruktor von __Child definiert werden.

Diese Änderungen führen in keinem Fall zu einer funktionalen Beeinflussung der Implementierung bzgl. der vorgegebenen Spezifikation. Die Laufzeiteigenschaften verändern sich nur insofern, daß bei der Initialisierung und Deinitialisierung eines Objektes der Klasse *__MInstance* ein zusätzlicher Aufruf des Konstruktors und Destruktors der Klasse *monitor_log* erfolgen muß. Das Laufzeitverhalten zwischen Initialisierung und Deinitialisierung bleibt aber unverändert. Damit ist nur Vorsicht geboten bei Messungen, die das Initialisieren von Modulinstanzen betreffen. Für alle anderen Messungen ist das Laufzeitverhalten der Bibliotheken unverändert.

[2]SIGKILL

3.3.4 Änderungen des DINGO

In DINGO wurde ein große Zahl von Änderungen vorgenommen. Eine genaue Aufzählung aller dieser Änderungen liegt dem Quellkode bei. Es sollen hier nur die wichtigsten Änderungen erläutert werden.

- Die Prozedur genMainsAndMakefile wurde so erweitert, daß automatisch für jede Modulinstanz, die als eigenständiger Betriebsystemprozess abläuft, die Interruptroutine installiert wird, die das Signal SIGUSR2 abfängt. Diese veranlaßt, daß der Prozeß zu Beginn des nächsten Zyklus abbricht.

- Es wird eine Instanz der Klasse *monitor* mit den an *Dingo* übergebenen Optionen initialisiert. An allen möglichen Stellen, an denen ein Meßpunkt generiert werden könnte, wurden die entsprechenden Methoden der Klasse *monitor* in den Quellkode von *Dingo* eingefügt. Zum Setzen der Meßpunkte führt dies allerdings nur, falls die entsprechenden Optionen beim Aufruf von *Dingo* gesetzt werden.

3.3.5 Meßpunkte des Leistungmodells

Die Meßpunkte für das Leistungsmodell werden mit Hilfe eines veränderten *Dingo* generiert. Die Auswahl der Meßpunkte erfolgt durch die entsprechenden Optionen und über spezielle Kommentare in der Spezifikation. Dabei ist die Vorgehensweise die Folgende:

Über einen speziellen Kommentar wird bei der Übersetzung der Spezifikation eines zusammengesetzten Typs (Array, Set, Record) ein zusätzliches Element definiert, welches es ermöglicht, einer Instanz dieses Datentypes eine eindeutige numerischen Wert, die sogenannte Sequenznummer, zuzuordnen. Diese Sequenznummer wird beim Erzeugen einer Instanz oder explizit an bestimmten Stellen der Berechnung vergeben. Beim Auftreten eines der Ereignisse des Leistungsmodells, **Empfange Paket**, **Sende Paket** oder **Kontrolle bei** kann der Zeitpunkt des Ereignisses zusammen mit der Sequenznummer abgespeichert werden.

Bei der Implementierung der einzelnen Meßpunkte des Leistungmodells treten wegen der Struktur des verteilten Ausführungmodells einige Schwierigkeiten auf. Deshalb wird die Implementierung im einzelnen beschrieben.

- Als Zeitpunkt des Ereignisses **Sende Paket** wird der Zeitpunkt der Umwandlung der Interaktion in den Ausgabestrom kurz vor dem Verschicken an ein anderes Subsystem genommen. Bei der Auswertung der Meßdaten muß dies in Betracht gezogen werden. Auf die Interaktion kann aber zu einem späteren Zeitpunkt nicht mehr als einzelne zugegriffen werden, weshalb dieser Zeitpunkt gewählt wurde. Die Umwandlung erfolgt in der virtuellen Methode *print_on* der Klasse *__Interact*, die durch *Dingo* generiert wird, wodurch die selektive Generierung der Meßpunkte kein Problem darstellt.

- Das Feststellen des Zeitpunktes des Ereignisses **Empfange Paket** wird dadurch erschwert, daß nur zu bestimmten Zeitpunkten der Ausführung, nämlich in der **Phase 1**, überprüft wird, ob Nachrichten angekommen sind, und diese entgegen genommen werden. Erst bei der Umwandlung der Eingabestroms in einzelne Interaktion kann die Sequenznummer eines Paketes festgestellt und das Ereignis aufgezeichnet werden. Die einzige sichere Aussage über die Ankunft des Paketes ist die, daß es in dem Zeitraum seit der letzten **Phase 1** angekommen ist[3]. Genauer ließe sich der Zeitpunkt nur dann bestimmen, wenn das Ankommen von Interaktionen über einen Interrupt signalisiert würde und der Eingabestrom dann direkt in die Interaktionen umgewandelt würde. In der bestehenden Implementierung des Ausführungsmodells ist dies nicht möglich. Die Meßpunkte werden auf die gleiche Art wie bei dem Ereignis **Sende Paket** generiert, in diesem Fall in der Methode *readParsFrom*.

- Der Meßpunkt für das Ereignis **Kontrolle bei** wird an die Stelle im Kode gesetzt, an der sich ein spezieller Kommentar in der Spezifikation befindet. Dabei kann optional die Sequenznummer eines Datenpaketes mit abgespeichert werden.

Beispiel

An einem Beispiel, dem Ereignis **Sende Paket**, soll die Implementierung des Leistungsmodells demonstriert werden.

Jede Interaktion, die über Betriebsystemprozessgrenzen versendet werden soll, muß in einen Strom umgewandelt werden. Bei *Pet-Dingo* werden dafür für jeden Datentyp Methoden zur Umwandlung generiert, die dann bei dem Versenden und Empfangen entsprechend angewendet werden.

Dingo wird für die Messung entsprechend modifiziert, so daß die Zeitpunkte dieser Umwandlung festgehalten werden können. Die Vorgehensweise für die Implementierung ist dabei die folgende:

1. Der Klasse des Datenpaketes wird ein Element hinzugefügt, *s_number*, welches die Sequenznummer repräsentiert. Desweiteren wird eine Methode *print_s_number* generiert, welche ein Objekt der Klasse *monitor_stamp* vom Ringpuffer reserviert, um darin das Meßereignis abzuspeichern.

2. Enthält die Estelle-Spezifikation einen speziellen Kommentar in der Definition des Kanal bzgl. einer Interaktion und Datenpaketes wird 3. ausgeführt.

3. In der Methode der Interaktion zum Einlesen aus dem Eingabestrom wird nach dem Einlesen des Datenpaketes die Methode *print_s_number* ausgeführt.

[3]Diese Aussage gilt nur unter der Voraussetzung, daß nur eine bestimmte Zahl (10) Interaktionen pro Ausführungszyklus bei einem Modulinstanz ankommen

Beispiel

1.
```
struct _SduType {
    _Char val[1000];

    long s_number;

    _SduType():s_number(act_s_number++) {}
    _Char& operator[](int i) {  return val[i-0];}


    _SduType& operator=(_SduType& a);
    _SduType( _SduType& a);

    print_s_number(__MInstance* mi, TYPE tp);

};

ostream& operator<<( ostream& os, _SduType& a) {
   os << "{ "

      << a.s_number << " ";

   for (int i=0;i<1000;i++) os << a.val[i] << " ";
   return os << "}";
}

_SduType::print_s_number(__MInstance* mi, TYPE tp)  {
#ifdef SM
    monitor_stamp* ms = mi->new_monitor_stamp();
#endif
    if (tp == SEND_PACKET)
    {
#ifdef SM
        ms->set_key(101);
#endif
    }
    if (tp == RECEIVE_PACKET)
    {
#ifdef SM
        ms->set_key(102);
#endif
```

```
    }
#ifdef SM
    ms->start();
    ms->start_pc();
#endif
#ifdef SM
    ms->set_value1(s_number);
#endif
}
```

2.
```
CHANNEL transmit(user,provider);
BY user:  req(sdu{#}: sduType);
BY provider: ind(sdu: sduType);
```

3.
```
ostream& _TransmitReq::printOn( ostream& os) {
  __Interact::printOn( os);
  os << "{ ";

  Sdu.print_s_number(mi,SEND_PACKET);

  os << " " << Sdu;
  return os << "}";
}
```

3.3.6 Meßpunkte des Ausführungmodells

Für die Implementierung der Meßpunkte des Ausführungmodells sind keine weiteren Änderungen der Laufzeitbibliothek außer den weiter oben beschriebenen nötig. Die Meßpunkte können alle in die virtuellen Methoden eingefügt werden, die von *Dingo* für die Instanzen neu definiert werden. Die Implementierung erfolgt dabei folgendermaßen:

- Die Meßpunkte für die Ereignisse **Zyklus** und **Phase 1 - 3**, sowie **Warteschlange** werden in der Methode *__startExec* gesetzt. Dafür wird *Dingo* modifiziert, so daß diese Methode für jedes Subsystem generiert wird.
- Bei den Meßpunkten **Sende Interaktion** und **Empfange Interaktion** wird analog wie bei denselben Ereignissen mit Paketen vorgegangen; die Klassen, die die Interaktionen beschreiben, werden dafür um das Element *s_number* erweitert.
- Für die Ereignisse **Auswahl** und **Transition** werden die Meßpunkte bei der Generierung der Methode *__selAndExec* gesetzt.

- Die Meßpunkte der Ereignisse **Anweisung**, **Zuweisung** und **Prozedur** werden bei der Erzeugung der Prozeduren für die Transitionen und Prozeduren generiert.

Beispiel

Für das Ereignis **Zyklus** wird der Zeitpunkt des Beginnes und des Endes eines Ausführungzyklus festgehalten. Dafür werden in der Methode *__startExec* die Meßpunkte zu Beginn und zum Ende der while-Schleife gesetzt.

```
void __MI_PM_body::__startExec( ) {
// infinite loop for subsystem execution cycle;
 struct timeval lastTime;
 struct timezone tzp;
extern void waitforactivity(int);
extern u_long timeSince( struct timeval *t);
#ifndef PARESTL
 u_long elapsed = 0;
#endif
 gettimeofday(&lastTime,&tzp);
#ifdef timelogging
 logFile = new LogFile(1000, False);
#endif
 while (!terminated)
{
   // gets all descending if any; if __mayExecuteNext<0, wait until
   // until something new arrives from outside with a timout of
   // __mayExecuteNext

#ifdef SM
    monitor_stamp* cycle_time = new_monitor_stamp();
#endif
#ifdef SM
    cycle_time->set_key(215);
#endif
#ifdef SM
    cycle_time->start();
    cycle_time->start_pc();
#endif

   if (__mayExecuteNext<0) {
      waitforactivity(-__mayExecuteNext);
   }
   // downInteracts may be excluded for testing
```

```
   __NETI->downInteractsFor( this);
   elapsed = timeSince(&lastTime);
   __update( (int)elapsed); // local update;
   if (!__selAndExec((int)elapsed)) __childrenExec((int) elapsed);
      // NOTE: sel and exec updates all children if local exec;
      // childrenExec will also forward results of last exec cycle, if any;
   __sendExternal(); // forwards interactions in __ascendingQ to
      // their destination subsystems;

#ifdef SM
    cycle_time->stop();
    cycle_time->stop_pc();
#endif

 }
}
```

Kapitel 4

Entwicklung des Auswertungswerkzeugs

Mit Hilfe der durch den Software-Monitor zur Laufzeit der Protokollimplementierung ermittelten Informationen soll eine Leistungsanalyse vorgenommen werden. Für die Leistungsanalyse müssen einerseits diese Informationen in eine lesbare Form umgewandelt werden. Andererseits soll, laut Problembeschreibung, ein Werkzeug zur Verfügung stehen, welches den Vorgang der Leistungsanalyse weitestgehend unterstützt.

Zur Entwicklung dieses Werkzeugs wird zuerst auf die Eigenschaften der Laufzeitinformationen eingegangen werden, um beurteilen zu können, was man aus diesen schließen bzw. nicht schließen kann in Bezug auf das Laufzeitverhalten der Protokollimplementierung.

Ausgehend von diesen Eigenschaften und den Leistungsmaßen, die zu ermitteln sind, werden die Funktionalitäten beschrieben, die das Werkzeug bieten muß.

In dem letzten Abschnitt wird darauf eingegangen, wie die Implementierung des Auswertungswerkzeugs erfolgt ist.

4.1 Eigenschaften der Laufzeitinformationen

Bei den ermittelten Laufzeitinformationen handelt es sich um eine Menge von aufgezeichneten Ereignissen, denen der Eintrittszeitpunkt, sowie weitere, ereignisspezifische Information zugeordnet sind. Hier soll untersucht werden, inwieweit es sich um korrekte Information handelt, d. h. inwieweit entspricht diese Information der Realität des gemessenen Systems. Dazu muß das Verfahren betrachtet werden, das diese Laufzeitinformation geliefert hat. Dies erfolgte im vorherigen Kapitel. Wie man daraus erkennen konnte, handelt es sich um verläßliche Laufzeitinformation, mit Ausnahme der ermittelten Zeiten

Da diese Zeiten der Mittelpunkt des Interesses sind, sollen die Zeiten näher betrachtet werden, um doch noch zu Aussagen gelangen zu können. Dafür sind

die Gründe der Verfälschung festzuhalten und Maßnahmen zu beschreiben, die ein sinnvolles Arbeiten mit dieses Zeiten ermöglichen.

- Der Software-Monitor benötigt zur Ermittlung und Sicherung der Laufzeitinformation einen Anteil der Rechenzeit. Dadurch wird die Laufzeitinformation selbst beeinflußt. Eine Größe, die die Qualität einer Messung beschreibt, ist dieser Anteil der Rechenzeit, der sogenannte *overhead*[FER83].

- Ist die Dauer eines Zeitintervalls nicht ein ganzzahliges Vielfaches der Auflösung der verwendeten Uhr, so ergibt sich eine gemessene Dauer, die maximal um die Auflösung T von der tatsächlichen Dauer abweicht(siehe Abbildung 4.1). Bei dem Fehler handelt es sich dabei um eine gleichverteilte Zufallsvariable auf dem Intervall $[-T, T]$. Benötigt man ein genaueres Ergebnis, so erhält man dieses durch Mittelwertbildung über eine große Zahl von Zeitintervallen. Der Erwartungswert bei einer vorgenommenen Mittelung ist die exakte Dauer des Zeitintervalls, da der Erwartungswert der auf $[-T, T]$ gleichverteilte Zufallsvariable 0 ist[FER83].

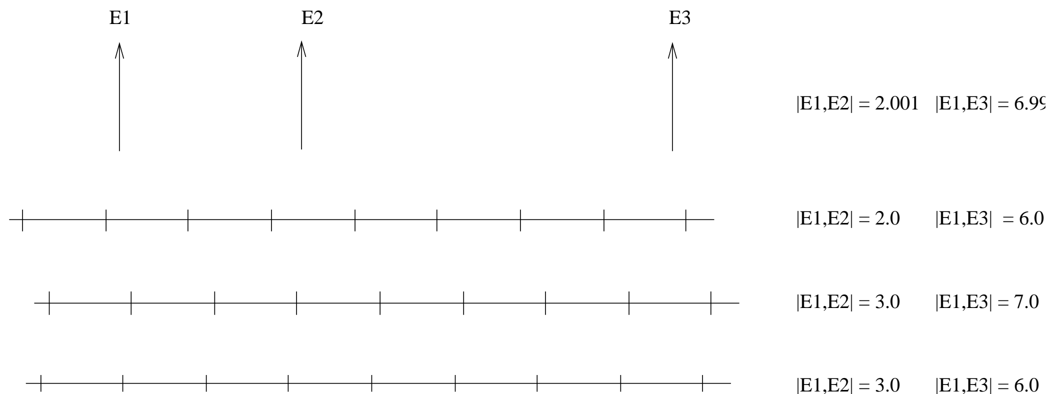

Abbildung 4.1: Meßfehler wegen diskreter Zeit

- Das zur Laufzeit ermittelte Zeitintervall wird durch das Einfügen von Kode des Software-Monitors größer. Ist es bekannt, um wieviel Zeit der eingefügte Kode das Zeitintervall vergrößert, so kann man diesen Wert bei der Bestimmung der Dauer des Zeitintervalls abziehen.

- Bei Realzeitmessungen, bei denen verschiedene Uhren verwendet werden, tritt das Problem der Synchronität der Uhren auf. Dieses Problem ist nur bis zu einer bestimmten Fehlerschranke lösbar. Weiß man die zeitliche Differenz der Uhren zur Laufzeit, bis auf diese Fehlerschranke, dann kann man bei der Analyse der Meßdaten dieses Wissen einbringen.

- Sind die Zeitpunkte zweier Ereignisse mit Realzeit festgestellt, so ist die Dauer des Zeitintervalls zusätzlich von Unterbrechungen und Kontextwechseln abhängig. Die allgemeine Rechenlast hat demnach Einfluß auf das Meßergebnis. Es ist deshalb bei Verwendung dieser Zeit darauf zu achten, daß zur Laufzeit eine möglichst geringe Rechenlast vorliegt. Wenn dieses gegeben ist und man eine große Zahl von Laufzeitinformation der gleichen Ereignisse hat, dann besteht die Möglichkeit, solche Intervalle zu verwerfen, deren Dauer sehr weit vom Mittelwert abweichen. Insbesonders ist dies geboten für Intervalle, die um ein Vielfaches der Zeitscheibe größer sind, wie in Abbildung 4.2 dargestellt. Im Falle, daß die reale Rechenzeit zwischen den beiden Ereignissen gleich ist, ergibt sich, nach Verwerfung dieser Werte, ein neuer Mittelwert, welcher näher an der realen Rechenzeit liegt. Wenn allerdings das Zeitintervall größer als die Zeitscheibe des Betriebsystems ist, erhält man über Mittelwertbildung und Verwerfung keine verläßliche Aussage. In diesem Fall muß man eine neue Messung vornehmen, in der das große Intervall in mehrere kleine Intervalle unterteilt wird, mit der Voraussetzung, daß alle diese kleiner als die Zeitscheibe sind. Die Mittelwerte dieser kleinen Intervalle können nach dem oben angeführten Verfahren ermittelt werden. Die Summe der Mittelwerte der Teilintervalle ergeben eine deutlich bessere Abschätzung des realen Mittelwertes als die direkte Mittelwertbildung über das große Intervall. Eine andere Möglichkeit wäre, für die Ermittelung von Intervalle dieser Größenordnung Prozeßzeit zu verwenden.

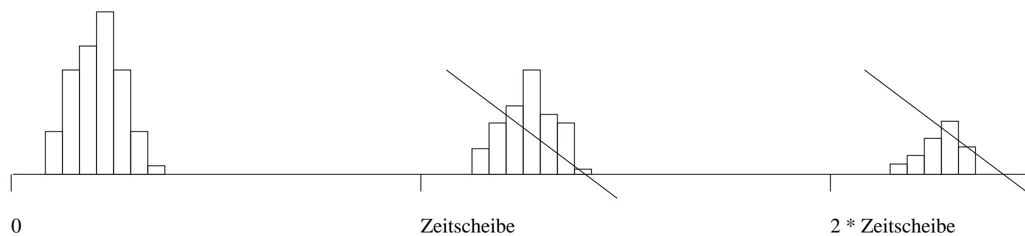

Abbildung 4.2: Verwerfung von gemessenen Intervallen

- Bei der Verwendung von Prozeßzeit besteht auch die Möglichkeit, daß die empirischen Mittelwerte einen großen Fehler aufweisen. Dieser ergibt sich insbesonders dann, wenn das Vorschalten einer Uhr mit dem Erhalten der Zeitscheibe in irgendeiner Art korreliert. Je größer diese Korrelation ist, umso weiter ist der empirische von dem wirklichen Mittelwert des Zeitintervalls entfernt. Diese Korrelation ist nicht einfach feststellbar, es können nur große Korrelationen mit Sicherheit festgestellt werden. Beispielweise ist Korrelation erkennbar, wenn eine ermittelte Zeitdauer in Prozesszeit größer als dieselbe in Realzeit ist. In diesem Fall ist der ermittelte Wert zu verwerfen.

4.2 Funktionalität des Auswertungswerkzeugs

Die Aufgabe des Auswertungswerkzeugs *PATO* (**P***erformance* **A***nalysing* **TO***ol*) ist es, eine Leistungsanalyse aufgrund der erhaltenen Laufzeitinformationen zu ermöglichen. Dazu gehört die Darstellung in einem lesbaren Format sowie die möglichst weitgehende Unterstützung bei der Auswertung der Laufzeitinformationen. Das Auswertungswerkzeug besitzt demnach die Funktionalität der Ausführungs- und Auswertungsschicht des Modells eines Softwaremonitors aus Abschnitt 3.2.1. Die Funktionalitäten sollen hier im einzelnen vorgestellt werden:

Lesbares Format

Die einzelnen Meßereignisse werden als ein Tupel dargestellt, welches aus den folgenden Einträgen besteht:

- Der eindeutige Namem der Modulinstanz, der von *Dingo* erzeugt wird, allerdings *ohne* den Rechnernamen.
- Die Realzeit, in der das gemessene Ereignis stattgefunden hat. Bei gemessenen Intervallen werden jeweils der Zeitpunkt des Anfang- und Endereignisses dargestellt.
- Die Prozesszeit wird analog der Realzeit dargestellt.
- Die eindeutige Bezeichnung der Ereignisses, wie sie im Meßmodell angegeben wurden.

Meßspezifische Information

Um die Messung besser beurteilen zu können, sind für jede Modulinstanz einige Informationen über den Meßvorgang verfügbar.

- Der genaue Zeitpunkt des Beginnes der Messung.
- Die Dauer der Messung.
- Die Anzahl der gemessenen Ereignisse.
- Die Rechenzeit, die der eingefügte Kode für die Meßpunkte eines Intervalls benötigte.
- Die Rechenzeit, die der eingefügte Kode für den Meßpunkt eines Ereignisses benötigte.
- Der *Overhead*, der Anteil an der Gesamtrechenzeit, welcher für das Ausführen der Meßpunkte benötigt wurde.

Leistungsmaße

Aus den gemessenen Laufzeitinformationen sollen die Leistungsmaße ermittelt werden. Die Leistungsmaße ergeben sich dabei als zeitliche Beziehungen der Laufzeitinformationen. *PATO* unterstützt die Ermittlung der folgenden Leistungsmaße:

- Die *Verzögerung* wird ermittelt als der zeitliche Abstand zwischen zwei Meßereignissen. Für die Realzeit wird von der gemessenen Zeit die Rechenzeit abgezogen, die für das Ausführen des Meßpunktes eines Meßereignisses benötigt wird (siehe Abbildung 4.3).

- Die *Ausführungsrate* ergibt sich aus der Anzahl der Ereignisse pro Zeiteinheit. Für die Realzeit wird sie durch den Reziprok des zeitlichen Abstands des Ereignisses mit dem Folgeereignis des gleichen Meßpunktes ermittelt. Bei der Prozeßzeit werden, wegen der geringen Auflösung der Prozeßzeit, die Ereignisse pro Zeiteinheit gezählt .

- Die *Ausführungszeit* ist nur für Intervalle definiert. Einem gegebenen Intervall wird die Dauer in der jeweiligen Zeit zugeordnet. Auch hier wird, im Falle der Realzeit, von der gemessenen Zeit eine bestimmte Zeit abgezogen. Es handelt sich dabei um die Zeit, die zur Ermittlung der Zeitmarke benötigt wird (siehe Abbildung 4.3).

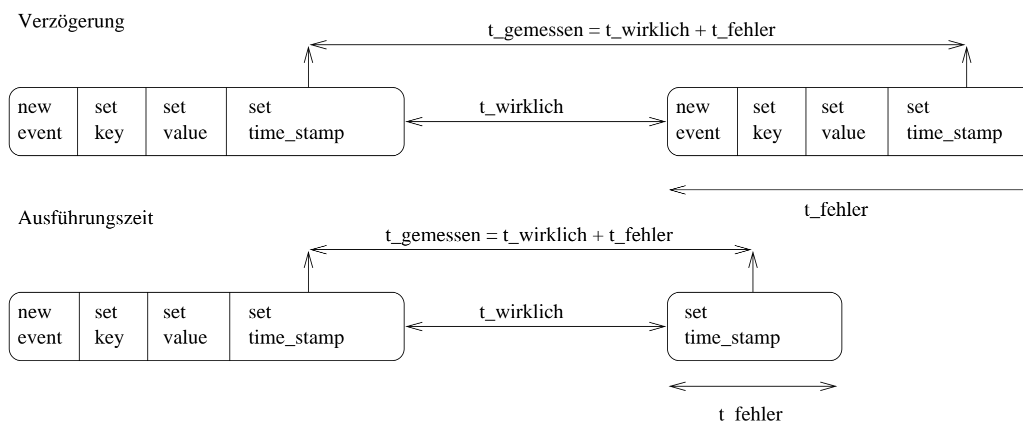

Abbildung 4.3: Ermittlung der Leistungsmaße

Statistik

Wie schon weiter oben festgestellt, erhält man aussagekräftige Leistungsmaße nur mit Hilfe statistischer Mittel. *PATO* stellt diese Funktionalität zur Verfügung. Es wird für jedes ermittelte Leistungsmaß die folgende Statistik erzeugt und dargestellt:

- Der *Stichprobenumfang* n.
- Das *Maximum*, das als $\max_{i \leq n} X_i$ definiert ist.
- Das *Minimum*, das als $\min_{i \leq n} X_i$ definiert ist.
- Der *Mittelwert*, der über die Formel $\overline{X_n} = \sum_{i=1}^{n} X_i$ ermittelt wird.
- Die *Varianz*, die mit Hilfe der Formel $s_n^2 = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \overline{X_n})$ berechnet wird.
- Die *Standartabweichung*, die sich als $s_n = \sqrt{s_n^2}$ ergibt.
- Das *Konfidenzintervall*, das eine Aussage macht über die Güte der Schätzung des Mittelwertes. Es ergibt sich aus dem Zusammenhang $P[\overline{X_n} - c * \frac{s_n}{\sqrt{n}}, \overline{X_n} + c * \frac{s_n}{\sqrt{n}}] = 1 - \alpha$. Im Falle, daß es sich bei den ermittelten Werten um identisch verteilte Zufallsvariablen handelt, gibt dieses Intervall um den Mittelwert die Wahrscheinlichkeit an, daß der Erwartungswert in diesem Intervall liegt. Nach dem zentralen Grenzwertsatz besteht zwischen c und α der Zusammenhang $\int_{-c}^{c} \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}} = 1 - \alpha$. Für $1 - \alpha = 0.99$ und $1 - \alpha = 0.95$ wird dieses c jeweils ermittelt und $c * \frac{s_n}{\sqrt{n}}$ ausgegeben. Es ist allerdings zu beachten, daß es sich nur um ein empirisches Konfidenzintervall handelt, welches mit Hilfe der Stichprobenvarianz berechnet wird. Je nach Korrelation der in die Schätzung eingehenden Werte kann es zu groß oder zu klein sein[LAN92].

Von der Statistik wird ein Histogramm dargestellt, womit man einen graphischen Eindruck der ermittelten Werte erlangt. An diesem Histogramm kann man die Werte erkennen, welche offensichtlich durch Einflüsse wie Kontextwechsel verfälscht sind. Dafür gibt es Möglichkeiten, den minimalen und maximalen zu akzeptierenden Wert anzugeben, der in die Statistik eingehen soll.

Weitere Möglichkeiten

Ist es möglich, die zeitliche Verschiebung der Uhren der verschiedenen Rechner untereinander festzustellen, so kann dieses Wissen PATO über jeweils eine Datei pro Rechner im Arbeitsverzeichnis mitgeteilt werden. Die ermittelten Zeitstempel der Ereignisse werden dann entsprechend korrigiert.

4.3 Implementierung des Auswertungswerkzeugs

Bei der Implementierung des Auswertungswerkzeugs *PATO* wurden die objektorientierten Fähigkeiten der Sprache C++ genutzt. Damit soll zum einen erreicht werden, die Lesbarkeit des Quellkodes zu gewährleisten und somit eine einfache Wartung des Werkzeugs zu ermöglichen. Zum anderen soll sich daraus eine einfache Erweiterbarkeit des Werkzeugs um zusätzliche Funktionalitäten ergeben[LIP91].

Für die Vereinfachung der Bedienung und der übersichtlichen Darstellung der Analysedaten besitzt das Auswertungswerkzeug eine graphische Bedienoberfläche. Diese wurde mit Hilfe des *Motif toolkit* erstellt und ist wegen der auch dort verwendeten objekt-orientierten Vorgehensweise einfach erweiterbar[MOA94].

Die Vorgehensweise war die Folgende: Zuerst wurde eine Menge von Klassen erstellt, die der Verwaltung und Bearbeitung der Laufzeitinformation dienen. Mit Hilfe des *Motif toolkit* wurde eine Bedienoberfläche generiert. Für diese Bedienoberfläche wurden sogenannte *Callback-Funktion* erstellt, welche an Ereignisse der Bedienoberfläche gekoppelt sind. Es handelt sich demnach um keinen kontinuierlichen Kontrollfluß; nur wenn durch den Bediener Ereignisse, wie z. B. das Drücken eines *Push-Buttons*, ausgelöst werden, werden die entsprechenden *Callback-Funktionen* ausgeführt.

In den folgenden Abschnitten sollen die speziell für das Auswertungswerkzeug erstellten Klassen und die Bedienoberfläche beschrieben werden. Dabei soll nur aufgezeigt werden, wo welche Funktionalität erbracht wird, und nicht der Quellkode im einzelnen beschrieben werden.

Klassen des Auswertungswerkzeugs

Die Implementierung des Auswertungswerkzeugs *PATO* baut auf den Klassen des Software-Monitors auf. Bei diesen Klassen handelt es sich um jene, welche zur Festhaltung der Laufzeit- und statischen Informationen des Software-Monitors dienen. Dabei sind hier die Methoden von Interesse, welche einen lesenden Zugriff auf die gespeicherte Information ermöglichen. Bei diesen Klassen handelt es sich um *monitor_stamp*, *monitor_log*, *monitor_db*, *monitor_info* und *time_s*. Diese Klassen bilden die Schnittstelle zwischen dem Software-Monitor und dem Auswertungswerkzeug; damit ist sichergestellt, daß die Bedeutung und das Format der Informationen für beide Teile identisch sind.

Als weitere Klassen wurden, neben einigen "Arbeitsklassen", die folgenden definiert:

monitor_event Ein Objekt dieser Klasse repräsentiert ein Meßereignis. Die Instanziierung erfolgt über das Zusammenfügen der relevanten Information aus den Instanzen der Klasse *monitor_stamp* und *monitor_info*. Damit

enthält eine Instanz dieser Klasse sowohl die statische als auch die Laufzeitinformation eines Meßereignisses. Die Klasse enthält Methoden für den Vergleich zweier Instanzen, für die Darstellung der statischen Information und der Laufzeitinformation und für die Ermittlung der Leistungsmaße.

monitor_list Bei dieser Klasse handelt es sich um eine Containerklasse, d. h. sie dient zur Verwaltung von Daten. In diesem Fall handelt es sich um eine geordnete Menge von Instanzen der Klasse *monitor_event*. Die Ordnung wird durch die Vergleichsoperatoren der Klasse *monitor_event* festgelegt. Da es sich um eine Menge handelt, müssen alle Instanzen vergleichbar sein, im Falle der Gleichheit zweier Instanzen wird nur eine in die Menge aufgenommen.

monitor_file_list Bei dieser Klasse handelt es sich um eine Ableitung der Klasse monitor_list. Es werden durch diese Klasse die Meßereignisse einer Modulinstanz verwaltet. Die Meßereignisse werden nach der realen Zeit geordnet, bei zu geringer Auflösung derselben wird nur ein Meßereignis pro diskretem Zeitpunkt in die Menge aufgenommen.

monitor_log_list Für diese Klasse gilt im wesentlichen dasselbe wie für *monitor_file_list*, es werden aber die Meßereignisse aller Modulinstanzen verwaltet. Die Ordnung basiert zusätzlich zur realen Zeit auf der lexikographischen Ordnung der von Dingo kreierten eindeutigen Modulinstanznamem.

monitor_event_list Diese Klasse ist auch eine Ableitung der Klasse *monitor_list*, allerdings wird hier nach den von Dingo vergebenen Schlüsselnummern der Meßpunkte geordnet. Für jeden Meßpunkt wird deshalb nur ein Ereignis in die Menge aufgenommen.

statistic Diese Klasse bietet alle Funktionalität, um eine statistische Auswertung vornehmen zu können. Sie stellt Methoden bereit für die Erstellung einer Statistik und die Ausgabe derselben als Text und Histogramm.

Bedienoberfläche des Auswertungswerkzeugs

Die Bedienoberfläche des Auswertungswerkzeugs gliedert sich in drei Sektionen(siehe 4.4). Die erste Sektion dient der Verwaltung der Dateien mit den Meßereignissen der einzelnen Modulinstanzen. In der zweiten Sektion können die Leistungsmaße bzgl. der Meßpunkte ermittelt werden. Die dritte Sektion stellt die gesamten aufgezeichneten Meßereignisse dar. Der Aufbau der drei Sektionen soll im folgenden näher beschrieben werden.

Ereignisdateien Die Ereignisdateien werden mit Hilfe eines Filerequesters einzeln ausgewählt und geladen. Zu jeder der geladen Ereignisdateien kann die in Abschnitt 4.2 beschriebene meßspezifische Information abgerufen werden.

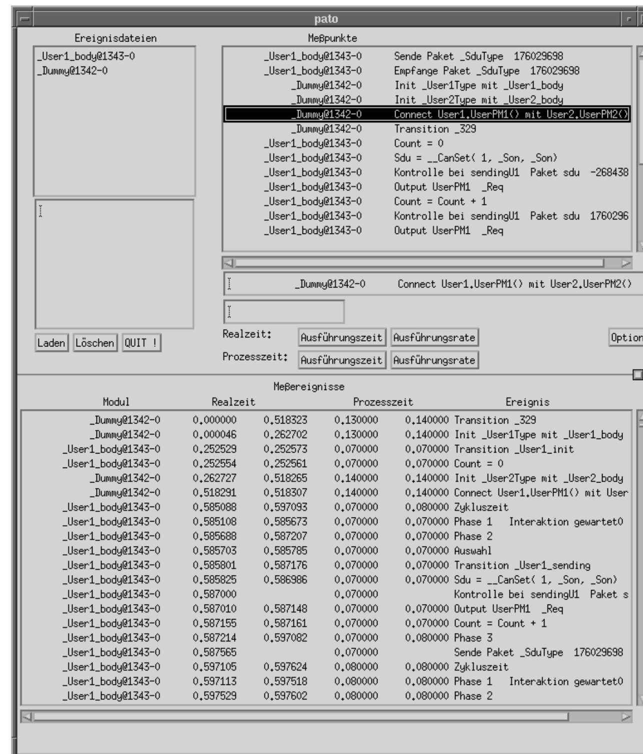

Abbildung 4.4: Bedienoberfläche des Auswertungswerkzeugs

Es können beliebig viele Dateien geladen werden, die nur kollektiv wieder gelöscht werden können.

Meßpunkte In dieser Sektion werden alle Meßpunkte angezeigt. Es können ein bzw. zwei dieser Meßpunkte ausgewählt werden, von denen ausgehend die Leistungsmaße bzgl. der dort aufgetretenden Meßereignisse ermittelt werden können. Die Leistungsmaße können sowohl ausgehend von der realen Zeit als auch ausgehend von der Prozeßzeit ermittelt werden.

Für jede der beiden Zeiten stehen jeweils drei Auswahlschalter zur Verfügung, um die Berechnung der Leistungsmaße zu veranlassen. Es erscheint bei der Berechnung der Leistungsmaße zuerst das Histogramm, welches alle ermittelten Leistungsmaße darstellt (siehe Abbildung 4.5).

An diesem kann über die Tasten $\ll$ und $\gg$ der größte bzw. kleinste Wert verworfen werden. Zu dem, gegebenenfalls durch Verwerfung einiger Werte, erlangtem Histogramm, kann eine Statistik (Abbildung 4.6) bzw. eine Liste der einzelnen Werte (4.7) ausgegeben werden. Für beide besteht die Möglichkeit, die dargestellten Werte auf eine Datei abzuspeichern.

Für das Leistungmaß *Verzögerung* können zusätzlich Optionen ausgewählt werden (siehe Abbildung 4.8). Über die zwei ersten Optionen wird be-

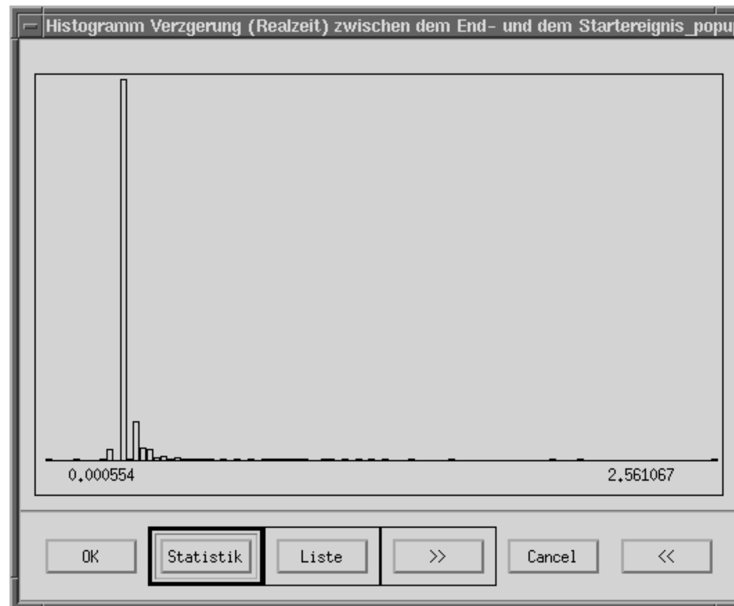

Abbildung 4.5: Histogramm

stimmt, wie die Verzögerung zweier Intervalle zu berechnen ist. Standardmäßig ist sie als die zeitliche Differenz des Endereignisses des ersten Intervalls und dem Startereignis des zweiten Intervalls festgelegt. Wenn es anders benötigt wird, kann dies entsprechend ausgewählt werden. Die dritte Option legt fest, in welche zeitliche Richtung nach dem zweiten Ereignis gesucht werden soll. Beispielweise kann ein rückwärtiges Suchen sinnvoll sein, wenn auf mehrmaliges Auftreten eines Ereignisse A eine Ereignis B auftritt und man an der *Verzögerung* zwischen dem letzten Auftreten von A vor B interessiert ist. Die vierte Option gibt an, ob die Sequenznummern für die Suche verwendet werden sollen. Bei positiver Wahl wird die *Verzögerung* nur zwischen Ereignissen mit gleicher Sequenznummer ermittelt. Andernfalls wird nach dem nächsten, entsprechend der Suchrichtung liegenden Ereignis gesucht.

Ereignisliste Die Ereignisliste stellt alle gemessenen Ereignisse in zeitlicher Ordnung dar. Im Falle, daß die Uhren hinreichend synchron sind, kann man die zeitliche Ordnung der Ereignisse anhand dieser Liste feststellen.

Die Fehlerausgaben erfolgen über das Fenster, von dem aus das Werkzeug gestartet wurde.

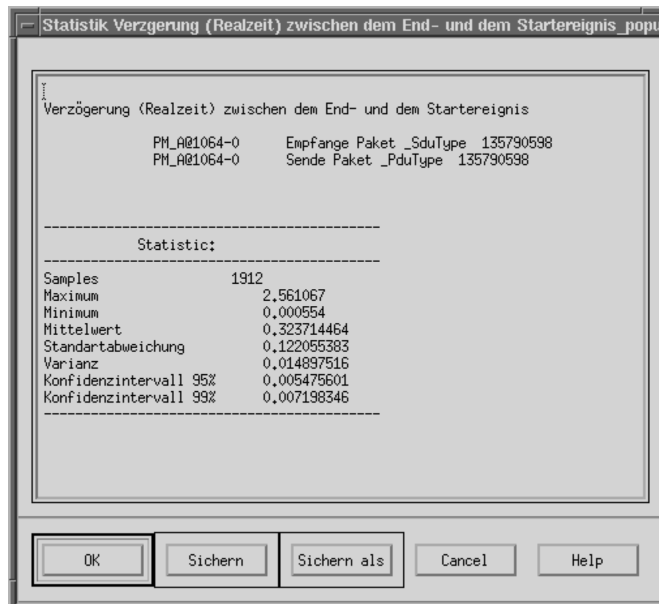

Abbildung 4.6: Statistik der Leistungsmaße

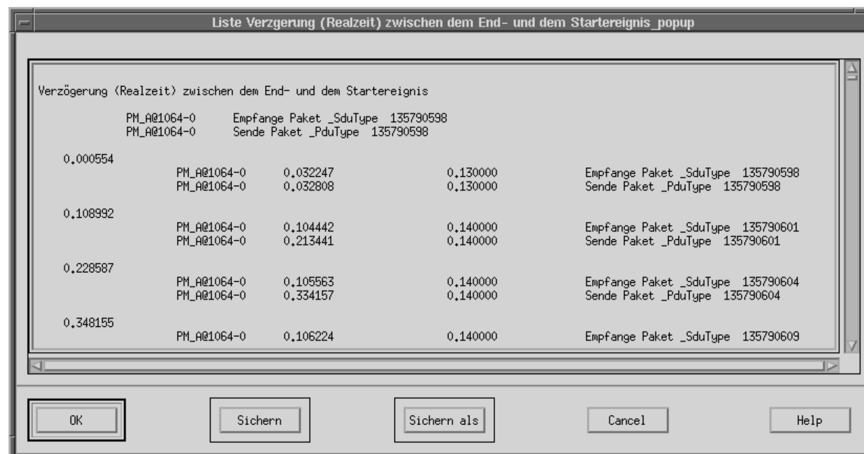

Abbildung 4.7: Liste der Leistungsmaße

Abbildung 4.8: Optionen

Kapitel 5

Ergebnisse

In diesem Kapitel soll anhand einiger Leistungsanalysen die Leistungsfähigkeit des Software-Monitors und des Auswertungswerkzeugs *PATO* demonstriert werden. Es wird im ersten Abschnitt ein einfaches AB-Protokoll analysiert. Im zweiten Abschnitt erfolgt eine Analyse der Leistungsfähigkeit der von Pet-Dingo generierten Implementierungen. Die Messungen wurden auf einer bzw. zwei Sparc 20 mit einem verfügbaren Hauptspeicher von $> 300Mb$ vorgenommen. Die Messungen wurden an einem Tag vorgenommen, an dem keine andere Benutzer an diesen Maschinen arbeiteten.

5.1 Leistungsanalyse eines AB-Protokolls

An einem einfachen AB-Protokoll soll eine Leistungsanalyse vorgenommen werden. Als Grundlage für die Leistungsanalyse wird dabei eine Estelle-Spezifikation des Rechnernetze Praktikums genommen, dessen Estelle-Spezifikation im Anhang erscheint. Da mit Estelle nur geschlossene Systeme spezifiziert werden können, sind, neben der Protokollinstanzen, auch die Dienstnutzer und der Basisdienst spezifiziert. Nach der Ausführung der initialen Transition der Spezifikation handelt es sich, wie in Abbildung 5.1 ersichtlich, um fünf asynchron voneinander ausführende Subsysteme.

Der endliche Automat, dem die Spezifikation der Protokollinstanzen zugrunde liegt, ist in Abbildung 5.2 dargestellt. Wie man an diesem erkennen kann, handelt es sich um ein Protokoll der Transportschicht. Dabei werden Datenpakete vom Typ SDU über Dienstprimitiven T_*, Datenpakete vom Typ PDU über Dienstprimitive N_* verschickt.

5.1.1 Analyse mit dem ersten Meßmodell

Die Protokollimplementierung wurde auf die folgenden Leistungsmaße hin untersucht:

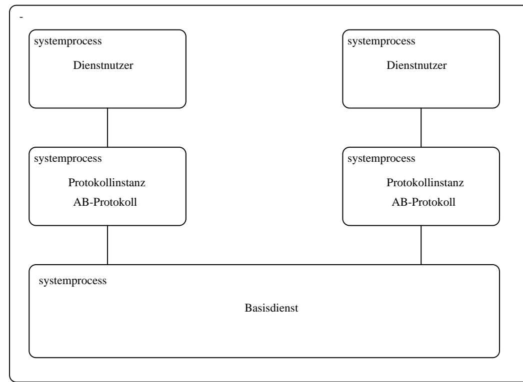

Abbildung 5.1: Spezifikation des AB-Protokolls

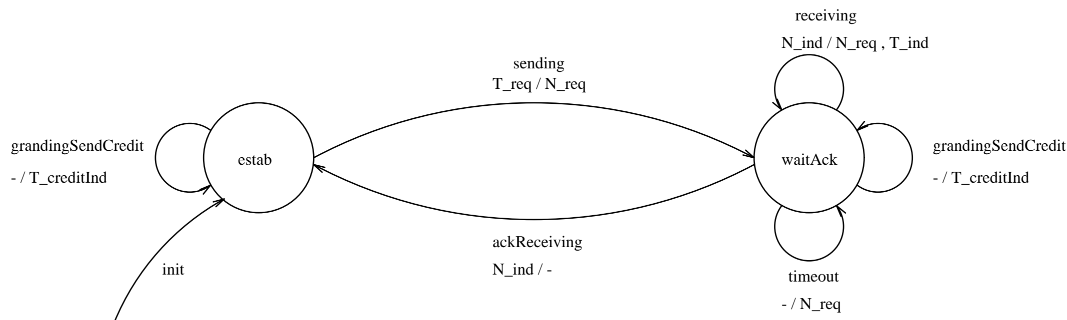

Abbildung 5.2: Endlicher Automat der AB-Protokollinstanz

- Die Verzögerung der Datenpakete durch die Protokollinstanz. Die Verzögerung wird dabei in zwei Richtungen untersucht:
  - Die Verzögerung zwischen Empfang einer SDU und der Verschickung als PDU.
  - Die Verzögerung zwischen Empfang einer PDU und der Verschickung als SDU.
- Der Durchsatz an Datenpaketen der Protokollinstanz. Dies läßt sich wieder unterteilen in die zwei möglichen Richtungen.
  - Der Durchsatz an PDUs, die über den Basisdienst verschickt werden.
  - Der Durchsatz an SDUs, die an den Dienstnutzer weitergegeben werden.

Um diese Messungen durchführen zu können, mußte die Spezifikation an einigen Stellen verändert werden. Alle diese Änderungen hatten keine Auswirkung auf das funktionale Verhalten der Protokollimplementierung.

- Bei dem Typ der zu übertragende Datenpakete muß es sich um nicht einfache Typen handeln, *" sduType = CHAR; "* wurde deshalb durch *"sduType = array[0..0] of INTEGER;"* ersetzt.
- Um aussagekräftige Messungen zu erhalten, muß das Protokoll einige Zeit in Gange sein; es wird also eine deutlich größere Zahl als sechs Pakete pro Richtung verschickt.
- In die Transition des Dienstnutzer, welche die Interaktion versendet, wird ein Kommentar eingefügt, welcher veranlaßt, daß jedem versendeten Paket eine Sequenznummer zugeordnet wird.
- In der Protokollinstanz werden Kommentare eingefügt, welche eine Übertragung der Sequenznummer von den SDUs auf die PDUs und umgekehrt veranlassen.

Aus der so veränderten Spezifikation wurde eine Implementierung generiert, welche Meßpunkte des ersten Meßmodells und damit eine Paketverfolgung ermöglicht. Die Verteilung der Implementierung erfolgte so, daß die zu messende Protokollinstanz auf einer Maschine alleine ausgeführt wurde, während alle anderen Modulinstanzen auf einer anderen Maschine zusammen ausgeführt wurden. Diese Maßnahme wurde ergriffen, um die Interferenz der zu messenden Protokollinstanz durch die anderen Instanzen auszuschließen. Allerdings muß deshalb die gesamte Kommunikation, in diesem Fall das Versenden von Interaktionen, über den socket-Mechanismus erfolgen. Wegen der so erfolgten Verteilung erhält man bzgl. des Leistungsmaßes *Durchsatz* keine repräsentativen Werte, da der Leistungsengpaß bei den anderen Instanzen liegt. Auf das Leistungmaß *Verzögerung* hat dies aber keine Auswirkung haben.

Nach einem Meßlauf und darauf folgender Auswertung mit dem Auswertungswerkzeug, ergaben sich die in Tabelle 5.1.1 aufgeführten Leistungsmaße[1]

Leistungsmaß	Richtung	Zeit	Mittelwert	Min	Max	Varianz	Umfang
Verzögerung	Empfange SDU Sende PDU	Realzeit	0.213069	0.000564	0.414314	0.000843	439
Verzögerung	Empfange SDU Sende PDU	Prozesszeit	0.0	0.0	0.0	0.0	439
Verzögerung	Empfange PDU Sende SDU	Realzeit	0.000550	0.000510	0.001014	0.0	1176
Verzögerung	Empfange PDU Sende SDU	Prozesszeit	0.0	0.0	0.0	0.0	1176
Durchsatz	Sende SDU	1/Realzeit	2.362766	0.196474	2.235003	0.021444	439
Durchsatz	Sende PDU	1/Realzeit	2.393672	1.611302	4.541779	0.038385	1176

Tabelle 5.1: Leistungsmaße der Protokollinstanz des AB-Protokolls

Wie zu erkennen ist, wird nur ein sehr geringer Durchsatz erreicht. Dies ist bedingt durch die Verteilung und die Verwendung des socket-Mechanismus als

[1]Die Einheiten aller Werte sind Sekunden bzw. $\frac{1}{Sekunde}$.

Kommunikationsplattform. Die Verzögerung in die verschiedenen Richtungen hat aber, im Gegensatz zum Durchsatz, eine sehr große Abweichung. Im Mittel ist die Verzögerung von Dienstnutzer zu Basisdienst im Mittel 400-mal größer als in die umgekehrte Richtung. In Prozeßzeit sind aber beide Verzögerungen und damit die Zeit, die ausgeführt wird, so klein, daß sie nicht festgestellt werden kann. Die zusätzliche Verzögerung könnte dadurch bedingt sein, daß zu einem zu frühen Zeitpunkt der Dienstnutzer das Datenpaket übergibt. Wie man anhand des endlichen Automaten der Protokollinstanz in Abbildung 5.2 erkennen kann, geschieht dies im Zustand *waitAck*, falls die Bestätigung des zuletzt gesendeten Paketes auf sich warten läßt. In diesen Fall wird im Zustand *waitAck* die Transition *grandingSendCredit* ausgeführt, die Protokollinstanz kann aber erst nach dem Empfangen der Bestätigung des zuletzt gesendeten Paketes in den Zustand *estab* übergehen und die Transition *sending* ausführen. Für die Messung der Verzögerung bietet es sich deshalb an, die Transition *grantingSendCredit* nur im Zustand *estab* auszuführen. Nachdem die Spezifikation entsprechend verändert wurde, ergaben sich die in Tabelle 5.1.1 aufgeführten Leistungsmaße. Die übrigen Leistungsmaße blieben unverändert.

Leistungsmaß	Richtung	Zeit	Mittelwert	Min	Max	Varianz	Umfang
Verzögerung	Empfange SDU Sende PDU	Realzeit	0.000550	0.000510	0.000800	0.000000	480
Durchsatz	Sende PDU	1/Realzeit	1.611908	1.211691	2.412423	0.004203	480

Tabelle 5.2: Leistungsmaße der veränderten Protokollinstanz

Wie man erkennen kann, ist nach dieser Veränderung die Größe der Leistungsmaße für beide Richtungen von gleicher Ordnung. Der Durchsatz an PDUs ist aber durch diese Maßnahme um ein Drittel gesunken. Für die Beurteilung, welche Variante besser ist, muß zum einen entschieden werden, welches Leistungsmaß wichtiger im konkreten Fall ist. Zum anderen ist, wie schon weiter oben festgestellt, die Messung des Durchsatzes in der gegebenen Konstellation nicht besonders sinnvoll. Aussagekräftige Werte erlangt man erst bei Messungen in der Zielumgebung, also dort, wo das Protokoll wirklich eingesetzt wird.

5.1.2 Analyse mit dem zweiten Meßmodell

Nach der Beurteilung der Protokollimplementierung mit Hilfe des ersten Meßmodells, soll die Implementierung aus einer anderen Sicht beurteilt werden. Dafür wurden Messungen mit dem zweiten Meßmodell durchgeführt. Dadurch sollen die Stellen der Implementierung ermittelt werden, die für die (geringe) Leistung verantwortlich sind. Es wurden Messungen auf der Spezifikationsebene, der Subsystemebene und der Modulebene vorgenommen. Die ermittelten Leistungsmaße sind in Tabelle 5.1.2 aufgelistet.

Bei der Betrachtung der Leistungsmaße der Spezifikationsebene erkennt man, daß der minimale und maximale Wert der Ausführungszeit in Realzeit um

Leistungsmaß	Typ	Zeit	Mittelwert	Min	Max	Varianz	Umfang
Ausführungszeit	Zyklus	Realzeit	0.124563	0.000318	0.808630	0.027299	2107
Ausführungszeit	Zyklus	Prozesszeit	0.000009	0.0	0.01	0.0	2107
Ausführungszeit	Phase 1	Realzeit	0.124110	0.000270	0.609331	0.027346	1786
Ausführungszeit	Phase 2	Realzeit	0.000066	0.000019	0.000248	0.0	1786
Ausführungszeit	Phase 3	Realzeit	0.000301	0.000001	0.009038	0.0	1786
Ausführungszeit	Auswahl	Realzeit	0.000039	0.000015	0.000122	0.0	1015
Ausführungszeit	Transition sending	Realzeit	0.000031	0.000029	0.000042	0.0	169
Ausführungszeit	Transition ackReceiving	Realzeit	0.000006	0.000005	0.000013	0.0	168
Ausführungszeit	Transition timeout	Realzeit	0.000059	0.000059	0.000059	0.0	1
Ausführungszeit	Transition grantingSendCredit	Realzeit	0.000030	0.000028	0.000048	0.0	170

Tabelle 5.3: Leistungsmaße der Protokollinstanz

den Faktor 2000 sich unterscheiden. Wenn man zusätzlich das Histogramm in Abbildung 5.3 betrachtet, kann man zum einen erkennen, daß die Meßwerte nicht gleichmäßig verteilt sind. Vielmehr treten in sehr regelmäßigen Abständen Häufungen von Meßwerten auf. Die Hälfte der Meßwerte befinden sich in der Nähe des Minimums; sie sind dort aber auch nicht gleichmäßig verteilt, wie man am zweiten Histogramm in Abbildung 5.4 erkennen kann.

Jede dieser Häufungen steht für einen bestimmten Zustand des Subsystems. Beispielsweise werden die Meßwerte in der nahen Umgebung des Minimums dann erhalten, wenn keine Interaktion von außen empfangen wird, und keine schaltbare Transition ermittelt werden kann.

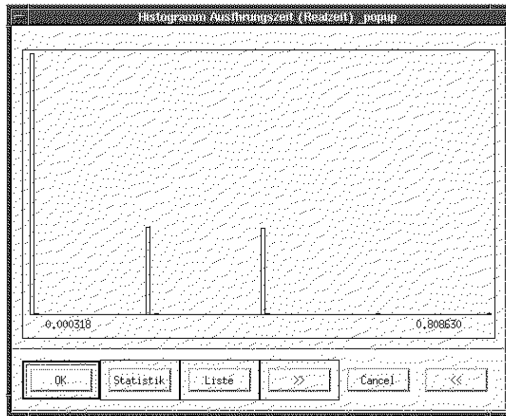

Abbildung 5.3: Histogramm aller Meßwerte

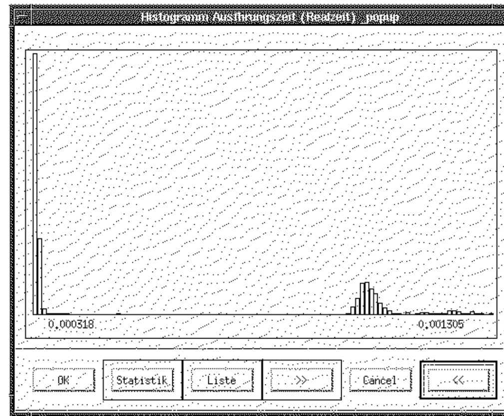

Abbildung 5.4: Histogramm nach Vorauswahl

Auf Subsystemebene kann man erkennen, daß die meiste Zeit in der Phase 1 verbracht wird. Die Dauer von Phase 2 und Phase 3 zusammen ist im Mittel nur 0.3% der Dauer eines Ausführungszyklus. Es ist aber auch zu bedenken, daß in der Phase 1 eine feste Zeit gewartet wird, wenn im letzten Zyklus keine Transition

geschaltet werden konnte und keine Interaktion empfangen wurde.

Auf Modulebene kann man erkennen, daß für die Auswahl und Ausführung ungefähr jeweils der gleiche Anteil an Rechenzeit verbraucht wird.

Abschließend kann man sagen, daß es sich um eine nicht sehr effiziente Implementierung handelt. Wie man an diesem Beispiel erkennen kann, muß eine Steigerung der Effizienz in der Phase1 beginnen, bevor an anderen Stellen der Implementierung eine Leistungssteigerung einen Sinn macht.

5.2 Leistungsanalyse des Pet-Dingo Entwicklungspaketes

In diesem Abschnitt soll systematisch untersucht werden, welches Leistungsverhalten eine mit Pet-Dingo generierte Implementierung erreichen kann. Dabei ist zum einen von Interesse, wo die Grenzen einer Implementierung liegen. Es sollen aber auch die Stellen ermittelt werden, an denen die Grenzen verschoben werden müssen. Dafür werden zwei einfache Spezifikationen betrachtet, die im Anhang aufgeführt sind.

5.2.1 Analyse eines Systems ohne Kommunikation

Die erste Spezifikation beschreibt ein Subsystem, welches eine immer schaltbare Transition besitzt. Der Transitionsblock dieser Transition ist leer. Es werden Messungen auf Spezifikation, Subsystem und Modulebene durchgeführt. Die erhaltenen Werte stellen eine Oberschranke der Leistung einer beliebigen Modulinstanz dar. Wie aus Tabelle 5.2.1 ersichtlich, liegt die Begrenzung der Leistungsfähigkeit in der Phase 1. In dieser wird 90% der Rechenzeit verbraucht. Desweiteren kann man erkennen, daß ein Subsystem weniger als 4000 Ausführungszyklen pro Sekunde abarbeiten kann.

Leistungsmaß	Richtung	Zeit	Mittelwert	Min	Max	Varianz	Umfang
Ausführungszeit	Zyklus	Realzeit	0.000254	0.000249	0.000856	0.0	4095
Ausführungszeit	Zyklus	Prozesszeit	0.000258	0.0	0.01	0.000002	4096
Ausführungsrate	Zyklus	Realzeit	3722	1141	3787	34710	4095
Ausführungszeit	Phase 1	Realzeit	0.000227	0.000224	0.000633	0.0	1365
Ausführungszeit	Phase 1	Prozesszeit	0.000219	0.0	0.01	0.000002	1365
Ausführungszeit	Phase 2	Realzeit	0.000008	0.000007	0.000062	0.0	1365
Ausführungszeit	Phase 2	Prozesszeit	0.000021	0.0	0.01	0.0	1365
Ausführungszeit	Phase 3	Realzeit	0.000000	0.000000	0.000051	0.0	1365
Ausführungszeit	Phase 3	Prozesszeit	0.000007	0.0	0.01	0.0	1365
Ausführungszeit	Auswahl	Realzeit	0.000006	0.000004	0.000341	0.0	2048
Ausführungszeit	Auswahl	Prozesszeit	0.000004	0.0	0.01	0.0	2048
Ausführungszeit	Transition	Realzeit	0.000002	0.0	0.000082	0.0	2048
Ausführungszeit	Transition	Prozesszeit	0.000004	0.0	0.01	0.0	2048

Tabelle 5.4: Leistungsmaße des Systems ohne Kommunikation

5.2.2 Analyse eines Systems mit Kommunikation

Die zweite Spezifikation beschreibt ein System, welches aus zwei Subsystemen besteht. Davon sendet eines der Subsysteme ununterbrochen Interaktionen an das zweite Subsystem. Anhand dieser Konstellation soll ermittelt werden, wieviele Pakete maximal empfangen und versendet werden können. Dafür werden die beiden Subsysteme auf zwei Rechner verteilt. Daraus ergibt sich bei Pet-Dingo eine Kommunikation mit TCP/IP. wobei die Kommunikation über den socket-Mechanismus erfolgt.

Leistungsmaß	Richtung	Zeit	Mittelwert	Min	Max	Varianz	Umfang
Ausführungszeit	Zyklus	Realzeit	0.001185	0.001000	0.002528	0.0	511
Ausführungszeit	Phase 1	Realzeit	0.000292	0.000278	0.000863	0.0	511
Ausführungszeit	Phase 2	Realzeit	0.000089	0.000084	0.000284	0.0	511
Ausführungszeit	Phase 3	Realzeit	0.000660	0.000586	0.001171	0.0	511

Tabelle 5.5: Leistungsmaße des sendenden Subsystems

Leistungsmaß	Richtung	Zeit	Mittelwert	Min	Max	Varianz	Umfang
Ausführungszeit	Zyklus	Realzeit	0.014452	0.006967	0.0216330	0.000003	240
Ausführungszeit	Phase 1	Realzeit	0.014272	0.006816	0.0214840	0.000003	511
Ausführungszeit	Phase 2	Realzeit	0.000128	0.000103	0.000517	0.0	511
Ausführungszeit	Phase 3	Realzeit	0.000003	0.000001	0.000122	0.0	511

Tabelle 5.6: Leistungsmaße des empfangenden Subsystems

Wie man anhand der Tabellen 5.2.2 und Tabelle 5.2.2 erkennen kann, benötigt ein Subsystem etwa zehnmal soviel Zeit, um eine Interaktion zu empfangen, als sie zu versenden. Es können etwa 850 Interaktionen pro Sekunde verschickt, aber nur etwa 70 Interaktionen empfangen werden. Dies zeigt wiederum recht deutlich, daß eine Optimierung in der Phase 1 angreifen muß, wenn man Pet-Dingo als Kode-Generator für die Generierung von Prototypen verwenden will.

Kapitel 6

Ausblick

Zusammenfassend läßt sich sagen, daß die in der Problembeschreibung gesetzten Ziele erreicht worden sind . Nichtsdestotrotz sind Erweiterungen denkbar, die das Messen und Auswerten noch weiter vereinfachen könnten.

Vorstellbar wäre eine integrierte Meßumgebung mit graphischer Bedienoberfläche, von der aus das Setzen der Meßpunkte, die Generierung der Implementierung, das Starten der Meßläufe und die Auswertung erfolgen kann.

Die Übertragung des Meßverfahrens auf andere Kode-Generatoren, und so die Möglichkeit eines direkten Vergleiches verschiedener solcher Generatoren ist durch Kapselung der "Meßtechnik" in einige wenige Klassen einfach realisierbar. Der Vergleich mit einer Handimplementierung ist deshalb auch einfach möglich, sofern der Quellkode dieser als C++-Kode vorliegt.

Das Auswertungswerkzeug bietet zur Zeit nur eine einfache textuelle Ausgabe auf Dateien. Es ist durchaus denkbar, eine graphische Auswertung der statistischen Werte vorzunehmen und diese in einem solchen Format abzuspeichern, daß sie direkt von einer Textverarbeitung übernommen werden können.

Bei jeder Übertragung des Meßverfahrens auf andere Kode-Generatoren sind in jedem konkreten Fall die Forderungen aus Abschnitt 3.2.1 zu überprüfen. Der *Overhead* des Meßverfahrens kann sich insbesonders im Bereich der Hochgeschwindigkeitsprotokolle als zu groß erweisen, um eine sinnvolle Leistungsanalyse vornehmen zu können. Dies gilt es bei allen zukünftigen Anwendungen zu beachten.

Anhang A

Analysierte Spezifikation

Die Estelle-Spezifikationen, deren Ergebnisse der Leistungsanalyse in Kapitel 5 dargestellt wurde, werden hier im einzelnen aufgeführt. In den Spezifikationen werden die Zeilen in Fettdruck dargestellt, in denen eine Veränderung vorgenommen bzw. ein spezieller Kommentar eingefügt werden mußte, um eine Messung vornehmen zu können.

A.1 AB-Protokolls

Das ursprüngliche AB-Protokoll wurde an einigen Stellen geändert, um eine Leistungsanalyse vorgenehmen zu können. Die Änderungen, die das funktionale Verhalten betreffen, werden hier im einzelnen aufgeführt.

- Der Typ der SDU wurde in ein Array der Größe 1 von Integer umgewandelt, da Paketverfolgung nur bei strukturierten Datentypen möglich ist.
- Die Zahl der zu versendenden Pakete wurde auf 10000 erhöht.
- Es werden nur Pakete in eine Richtung verschickt, um so besser Messungen vornehmen zu können. Der Datenfluß in eine Richtung kann so ohne Beeinflussung durch den Datenfluß in die entgegengesetze Richtung erfolgen.
- Das unzuverlässige Medium wurde durch ein zuverlässiges Medium ersetzt. Die Messungen beziehen sich deshalb nur auf den fehlerfreien Fall.

```
SPECIFICATION ABprotokoll{^-1};

  { Praktikum Entwicklung von Kommunikationssystemen SoSe 95,
    Jan Bredereke, Reinhard Gotzhein
    Loesungsvorschlag fuer Aufgabe 2.
  }

  { Einfuehrung einer Flusskontrolle zwischen den Protokollmaschinen,
```

```
  Erweiterung um "Backpressure"-Flusskontrolle hin zum sendenden User.
  Einfuehrung eines unzuverlaessigen Mediums;
  daher wurden Massnahmen zur Behebung von Nachrichtenverlusten notwendig:
  das Alternating-Bit-Protokoll;
  Erweiterung auf eine symmetrische Uebertragung in beide Richtungen.
}

TIMESCALE SECONDS; { Zeiteinheit fuer die DELAY-Klauseln. }

CONST
  retransmissionTime = 5;
  firstSeqNum = 0;

TYPE
  netUserRangeType = 0..1;                  { Bereich der Indizes fuer   }
                                            { die Nutzer des Netzwerkes. }
  userKindType = (userA_kind, userB_kind);  { Parametertyp: Was fuer ein }
                                            { User? userA sendet 0 bis 10000 }
                                            { userB sendet 20000 bis 20000.   }

  sduType    = array[0..0] of INTEGER;      { Service Data Unit.         }

  pduIdType  = (dataRI, ack);               { Selektortyp fuer den       }
                                            { folgenden PDU-Typ.         }
  seqNumType = 0;                           { Sequenznummern-Typ.        }
  pduType    = RECORD                       { Protocol Data Unit:        }
                 seqNum: seqNumType;        { - Sequenznummer,           }
                 CASE pduId: pduIdType OF   { - Typ der PDU,             }
                   dataRI: (netData: sduType); {   * Nettodaten oder     }
                   ack:  ()                 {   * Bestaetigung.          }
               END;

{ Der Kanal, ueber den die Protokollmaschine ihrem User ihren Dienst
  zur Verfuegung stellt: }
CHANNEL transmitService(user, provider);

  BY user:     T_dataReq(sdu{#}: sduType);    { Aufforderung zur Datenuebertragung.}
  BY provider: T_dataInd(sdu{#}: sduType);    { Anzeige einer Datenuebertrag. }

               T_creditInd(credit: INTEGER);  { Gewaehrung von Sendekredit    }
                                              { an den User gegenueber der PM }
                                              { fuer die lokale Flusskontrolle}

{ Der Kanal, ueber den das Netzwerk einer Protokollmaschine ihren Dienst
  zur Verfuegung stellt: }

CHANNEL networkService(user, provider);

  BY user:     N_dataReq(pdu{#}: pduType);
  BY provider: N_dataInd(pdu{#}: pduType);


{ Aeussere Schnittstelle fuer einen User: }
MODULE userType SYSTEMPROCESS(userKind: userKindType);
  IP userToPm: transmitService(user) INDIVIDUAL QUEUE;
  END;
```

```
  { Aeussere Schnittstelle fuer eine Protokollmaschine: }
  MODULE protocolMachineType SYSTEMPROCESS;
    IP pmToUser: transmitService(provider) INDIVIDUAL QUEUE;
       pmToNet:  networkService(user) INDIVIDUAL QUEUE;
    END;

  { Aeussere Schnittstelle fuer das Netzwerk: }
  MODULE networkType SYSTEMPROCESS;
    { Das Netzwerk hat hier nur zwei IPs. Damit kann die Adressierung implizit
      erfolgen: Was in den einen IP hineingetan wird, kommt am anderen
      wieder heraus. }
    IP netToPm: ARRAY[netUserRangeType] OF networkService(provider)
       INDIVIDUAL QUEUE;
    END;

  {--------------------------------------------------------------------------}

  { Die (generische) Realisierung eines Users: }
  BODY userBody FOR userType;
    VAR sendMsg:     sduType;
lastSendMsg: sduType;
        receivedMsg: sduType;
        sendCredit: INTEGER;

    STATE sendingAndReceiving,
  onlyReceiving;

    STATESET either = [sendingAndReceiving, onlyReceiving];

    INITIALIZE                         { Initialisierung: }
      TO sendingAndReceiving { In den anfaenglichen Hauptzustand. }
NAME userInit:
BEGIN              { Und auch die Variablen initialisieren. }
  CASE userKind OF
  userA_kind:
    BEGIN

      sendMsg[0] := 0;
      lastSendMsg[0] := 10000;

    END;
  userB_kind:
    BEGIN

      sendMsg[0] := 20000;
      lastSendMsg[0] := 20000;

    END;
  END;
        sendCredit := 0;             { Zuerst ist noch kein Sendekredit vorhanden. }
        receivedMsg[0] := -1  { (Es ist immer gut, wenn Variablen einen }
      END;               { definierten Wert haben.) }

    { Gib der Protokollmaschine nacheinander Uebertragungswuensche fuer }
    { die Nutzdaten, sofern Sendekredit fuer die lokale }
```

```
    { Flusskontrolle vorhanden ist. }
    TRANS
      FROM sendingAndReceiving
        TO SAME
          PROVIDED (sendMsg[0] <> lastSendMsg[0]) AND (sendCredit > 0)
NAME userSending:
            BEGIN
              OUTPUT userToPm.T_dataReq(sendMsg){|sendMsg};
              sendMsg[0] := SUCC(sendMsg[0]);
              sendCredit := sendCredit - 1;
            END;

    { Empfange einen Sendekredit von der Protokollmaschine fuer die lokale }
    { Flusskontrolle. }
    TRANS
      FROM either
        TO SAME
          WHEN userToPm.T_creditInd(credit)
NAME userReceivingSendCredit:
            BEGIN
              sendCredit := sendCredit + credit
            END;

    { Gib der Protokollmaschine den letzten Uebertragungswunsch }
    { (sofern Sendekredit fuer die lokale }
    { Flusskontrolle vorhanden ist). }
    TRANS
      FROM sendingAndReceiving
        TO onlyReceiving
          PROVIDED (sendMsg[0] = lastSendMsg[0]) AND (sendCredit > 0)
NAME userSendingLast:
            BEGIN
              OUTPUT userToPm.T_dataReq(sendMsg);
              sendCredit := sendCredit - 1;
            END;

    TRANS
      FROM either
        TO SAME
          WHEN userToPm.T_dataInd(sdu)
NAME userReceiving:
            BEGIN
              receivedMsg := sdu;
              { $C$ printf ("user meldet den Empfang einer Nachricht \n"); }
            END;
  END;                                { BODY userBody FOR userType }

  {----------------------------------------------------------------------}

  BODY PM_body FOR protocolMachineType{^16};

    CONST

      creditLimit = 1;

    VAR
```

```
      msg: pduType;
      userSendCredit: INTEGER;
      sendSeqNum:    seqNumType;
      receiveSeqNum: seqNumType;

    FUNCTION nextSeqNum(currSeqNum: seqNumType): seqNumType;
    BEGIN
      nextSeqNum := SUCC(currSeqNum) MOD 2;
    END;

    { Eine totale Funktion auf einer nur manchmal definierten Komponente
      eines varianten Records: }
    FUNCTION isAckWithCorrectSeqNum(pdu: pduType; seqNum: seqNumType):
      BOOLEAN;
    BEGIN
      IF pdu.pduId <> ack THEN
isAckWithCorrectSeqNum := FALSE
      ELSE
isAckWithCorrectSeqNum := (pdu.seqNum = seqNum)
    END;

    { Eine totale Funktion auf einer nur manchmal definierten Komponente
      eines varianten Records: }
    FUNCTION isAckWithIncorrectSeqNum(pdu: pduType; seqNum: seqNumType):
      BOOLEAN;
    BEGIN
      IF pdu.pduId <> ack THEN
isAckWithIncorrectSeqNum := FALSE
      ELSE
isAckWithIncorrectSeqNum := (pdu.seqNum <> seqNum)
    END;

    STATE
      estab,
      waitAck;

    STATESET
      either = [estab,waitAck];

    INITIALIZE
      TO estab
NAME PM_init:
        BEGIN
          userSendCredit := creditLimit;
          sendSeqNum    := firstSeqNum;
          receiveSeqNum := firstSeqNum;
        END;

    { Gib einen vorhandenen Sendekredit der lokalen Flusskontrolle an }
    { den User weiter. Falls gleichzeitig auch eine andere Transition }
    { schaltbereit ist, findet eine nichtdeterministische Auswahl statt. }
    TRANS
      FROM estab
        TO SAME
          PROVIDED userSendCredit > 0
NAME PM_grantingSendCredit:
            BEGIN
```

```
              OUTPUT pmToUser.T_creditInd(userSendCredit);
              userSendCredit := 0
            END;

    TRANS
      FROM estab
        TO waitAck
          WHEN pmToUser.T_dataReq(sdu)
NAME PM_sending:
            BEGIN
              msg.seqNum  := sendSeqNum;
              msg.pduId   := dataRI;
              msg.netData := sdu;

              OUTPUT pmToNet.N_dataReq(msg){|msg=sdu};

              userSendCredit := userSendCredit + 1;
            END;

    TRANS
      FROM waitAck
        TO estab
          WHEN pmToNet.N_dataInd(pdu)
            PROVIDED isAckWithCorrectSeqNum(pdu, sendSeqNum)
NAME PM_ackReceiving:
              BEGIN
                sendSeqNum := nextSeqNum(sendSeqNum);
              END;

    { Ignoriere die Wiederholung einer bereits empfangenen Bestaetigung. }
    TRANS
      FROM waitAck
        TO SAME
          WHEN pmToNet.N_dataInd(pdu)
            PROVIDED isAckWithIncorrectSeqNum(pdu, sendSeqNum)
NAME PM_repeatedAck:
              BEGIN
              END;

    { Wiederhole die letzten Daten, wenn zu lange keine Bestaetigung kommt. }
    TRANS
      FROM waitAck
        TO SAME
          DELAY (retransmissionTime)
NAME PM_timeout:
            BEGIN
              OUTPUT pmToNet.N_dataReq(msg)
            END;

    { Bestaetige den Empfang einer PDU. Wenn sie noch nicht empfangen wurde, }
    { gib ihre Nutzdaten nach oben weiter. }
    TRANS
      FROM either
        TO SAME
          WHEN pmToNet.N_dataInd(pdu)
            PROVIDED pdu.pduId = dataRI
              VAR tmpMsg: pduType;
```

```
NAME PM_receiving:
                BEGIN
                  IF pdu.seqNum = receiveSeqNum THEN
                  BEGIN

                    OUTPUT pmToUser.T_dataInd(pdu.netData){|pdu.NetData=pdu};

                    receiveSeqNum := nextSeqNum(receiveSeqNum);
                  END;
                  tmpMsg.seqNum := pdu.seqNum;
                  tmpMsg.pduId := ack;
                  OUTPUT pmToNet.N_dataReq(tmpMsg);
                END;
  END; { BODY PM_body FOR protocolMachineType }

  {-------------------------------------------------------------------------}

  BODY networkBody FOR networkType;

  INITIALIZE
NAME network5Init:
    BEGIN
    END;

    TRANS
      WHEN netToPm[0].N_dataReq(pdu)
NAME leftToRight:
        BEGIN
          OUTPUT netToPm[1].N_dataInd(pdu)
        END;

    TRANS
      WHEN netToPm[1].N_dataReq(pdu)
NAME rightToLeft:
        BEGIN
          OUTPUT netToPm[0].N_dataInd(pdu)
        END;
  END; { BODY networkBody FOR networkType }


  {-------------------------------------------------------------------------}

  MODVAR
    userA: userType;
    userB: userType;
    PM_A: protocolMachineType;
    PM_B: protocolMachineType;
    network: networkType;

  INITIALIZE
NAME aufgabe2Init:
    BEGIN
      INIT userA WITH userBody(userA_kind);
      INIT userB WITH userBody(userB_kind);

      INIT PM_A WITH PM_body{%PM_A}{@"sep2"};
      INIT PM_B WITH PM_body{%PM_B}{@"sep1"};
```

```
      INIT network WITH networkBody;
      CONNECT userA.userToPm TO PM_A.pmToUser;
      CONNECT userB.userToPm TO PM_B.pmToUser;
      CONNECT PM_A.pmToNet TO network.netToPm[0];
     CONNECT PM_B.pmToNet TO network.netToPm[1];
    END;
END.
```

A.2 System ohne Kommunikation

Wie aus der Spezifikation ersichtlich handelt es sich um ein System, welches nur eine Transition mit leerem Transitionsblock enthält. Diese ist immer schaltbar. Die Ausführung dieser Spezifikation führt zur Ausführung des (leeren) Transitionsblock in jedem Ausführungszyzklus des Systems.

```
SPECIFICATION dummy SYSTEMPROCESS;
INITIALIZE
  BEGIN
  END;

    TRANS
NAME do_nothing:
      BEGIN
      END;
END.
```

A.3 System mit Kommunikation

Das System besteht aus zwei Subsystemen, wobei das eine Subsystem an das andere Subsystem Interaktionen verschickt. Das zweite Subsystem besitzt nur eine Eingabetransition, über die es Interaktion empfangen kann.

```
SPECIFICATION dummy;

 { TIMESCALE SECONDS;  Zeiteinheit fuer die DELAY-Klauseln. }
  TYPE

    sduType = array [0..0] of INTEGER ;


  CHANNEL transmit(user,provider);

   BY user:     req(sdu:{#} sduType);
                ende;
   BY provider: ind(sdu:{#} sduType);
                conf;
```

```
  MODULE user1Type SYSTEMPROCESS;
    IP userPM1: transmit(user) INDIVIDUAL QUEUE;
  END;

  MODULE user2Type SYSTEMPROCESS;

    IP userPM2: transmit(provider) INDIVIDUAL QUEUE{#};

  END;

  BODY user1_body FOR user1Type;
    VAR sdu: Sdutype;

  INITIALIZE
NAME user1_init:
    BEGIN
    END;

    TRANS
NAME user1_sending:
      BEGIN

        output userPM1.req(sdu){|sdu};

      END;
END;

BODY user2_body FOR user2Type;

  INITIALIZE
NAME user2_init:
    BEGIN
    END;

    TRANS
      WHEN userPM2.req(sdu)
NAME user2_receiving:
        BEGIN
        END;
END;

MODVAR user1: User1Type;
       user2: User2Type;

INITIALIZE
 BEGIN
    INIT user1 WITH user1_body;

    INIT user2 WITH user2_body{@"sep2"};

    CONNECT user1.userPM1 TO user2.userPM2;
  END;

END.
```

Anhang B

Tutorial

Dieser Anhang enthält eine genaue Beschreibung der Vorgehensweise, wie man für die Analyse einer Protokollimplementierung vorgehen muß. Im ersten Abschnitt wird beschrieben, wie die automatische Implementierung des Software-Monitors in die Protokollimplementierung vonstatten geht. Im zweiten Abschnitt wird die Bedienung des Auswertungswerkzeugs *PATO* beschrieben.

B.1 Bedienung des Software-Monitors

Um eine Messung an einer Protokollimplementierung vornehmen zu können, muß der Software-Monitor installiert werden. Dieser erfolgt bei der Generierung der Implementierung durch Pet-Dingo. Die Vorgehensweise ist dabei die folgende:

1. Im Estelle Kode können über spezielle Kommentare Meßpunkte gesetzt werden oder die Messung beeinflußt werden(AbschnittB.1.1).

2. Pet wird aufgerufen, um den Estelle-Kode in einen Zwischenkode umzuwandeln.

3. Beim Aufruf von Dingo wird eine zusätzliche Option übergeben, welcher das Meßmodell festlegt, welches für die Messung zugrunde liegen soll(Abschnitt B.1.2).

4. Um nicht alle möglichen Meßpunkte eines Meßmodells setzen zu müssen, können aus der durch Dingo erzeugten Meßpunkt-Datenbank diejenigen Meßpunkte ausgesucht werden, an denen gemessen werden soll. Dafür schreibt man die eindeutige Kennung der Meßpunkte, die man setzen bzw. nicht setzen will in jeweils eine Datei.

5. Die zuvor erzeugten Datei(en) werden bei einem zweiten Aufruf von Dingo als zusätzliche Parameter übergeben. Damit erhält man die gewünschten Meßpunkte.

Nach diesem Vorgehen besitzt man eine Protokollimplementierung mit integriertem Software-Monitor. Um Meßergebnisse erhalten zu können, muß die Protokollimplementierung mit Hilfe der *site_server* ausgeführt und beendet werden. Bei der Beendigung werden die ermittelten Meßdaten für jede Modulinstanz in eine Datei abgespeichert.

B.1.1 Formate der Kommentare

Für die verschiedenen Stufen bei der Vorbereitung und Ausführung der Messung werden einige Formate verwendet. Diese werden hier im einzelnen aufgeführt.

Puffergröße

Die Puffergröße bestimmt die Anzahl von Meßereignissen, die gespeichert werden können. Möchte man einen anderen, als den Default-Wert zur Verfügung stellen, so kann dieses über ein Kommentar bei der Definition des Modulrumpfes festgelegt werden.

Syntax

```
module_body_definition = "body" IDENTIFIER "for"  header-identifier
                         ["{^" buffer-size "}" ]";"
           buffer-size = ["-"] digit-sequence.
```

Semantik Die Größe des Puffers wird als $2^{buffer-size}$ festgelegt. Im Falle, daß es sich bei *buffer-size* um eine negative Zahl handelt, wird kein Puffer angelegt.

Beispiele

```
SPECIFICATION dummy{^-1}; \\ kein Puffer
BODY user1_body FOR user1Type{^16}; \\ Puffer mit Groesse 65536
```

Warteschlangen

Soll die Größe von einer Warteschlange über die Laufzeit verfolgt werden, so muß dieses im Estelle-Kode durch einen Kommentar veranlaßt werden.

Syntax

```
interaction-point-declaration =  IDENTIFIER-LIST
                                 ":"interaction-point-type ["{#}"]
                               | IDENTIFIER-LIST
                                 ":" "array" "[" index-type-list "]"
                                     "of" interaction-point-type
                                     {"{#" ip-index "}"}
                    ip-index = index-type.
```

Semantik Bei Erscheinen des Kommentars # bei der Deklaration des Interaktionspunkts wird die Größe der Warteschlange zu Beginn jedes Ausführungzyklus als Ereignis festgehalten. Im Falle, daß es sich um ein Array von Interaktionspunkten handelt, muß der Index des Array zusätzlich angegeben werden.

Beispiele

```
IP userToPm: transmitService(user) INDIVIDUAL QUEUE{#};
IP TSAP :  array[Sites] of Transport_Channel_type(provider){#'A'}{#'B'};
```

Paketverfolgung

Soll ein Paket innerhalb der Spezifikation verfolgt werden, so sind die folgenden Kommentare zu ergänzen.

Syntax

```
    channel-definition = channel-heading channel-block.
         channel-block = +{ interaction-group } .
     interaction-block = +{ "by" role-identifier
                          [ "," role-identifier ] ":"
                         +{interaction-definition} .
interaction-definition = IDENTIFIER
                         [ "(" VALUE-PARAMETER-SPECI
                         { ";" VALUE-PARAMETER-SPECI} ");".
 VALUE-PARAMETER_SPECI = identi-list ":" type-identifier.
           identi-list = identifier ["{#}"] ("," identifier["{#}"]).

        statement-part = compound-statement .
    compound-statement = "begin" statement-sequence "end"
    statement-sequence = statement [packet-com]
                         { ";"statement [packet-com] }.
            packet-com = "{|"destination_packet[.source_packet] "}".
         source_packet = structured-type-identifier .
    destination_packet = structured-type-identifier .
```

Semantik Erscheint # als Kommentar bei der Definition der Kanäle, so wird dem Datentyp der VALUE-PARAMETER-SPECIFICATION ein zusätzliches Feld zugeordnet, welches eine eindeutige Identifizierung einer Instanz dieses Datentypes ermöglicht. Einschränkend darf dieser Kommentar **nur** bei den Typen, die als Set, Array oder Record definiert sind, gesetzt werden. Der Kommentar |x vergibt der Variablen *destination_packet* eine neue eindeutige Nummer; der Kommentar |x.y überträgt die eindeutige Nummer der Variablen *source_packet* auf die Variable *destination_packet*. Falls die

Variablen in der Transition nicht referenziert werden, so ist für diese Variable das Statement *packet := packet;* zu ergänzen. Bei der Generierung des Kodes wird dieses Statement nicht erzeugt, die Variable ist aber dadurch im Transitionsblock referenzierbar.

Beispiele

```
CHANNEL transmitService(user, provider);
  BY user:     T_dataReq(sdu{#}: sduType);

  BY provider: T_dataInd(sdu{#}: sduType);
CHANNEL networkService(user, provider);
  BY user:     N_dataReq(pdu{#}: pduType);
  BY provider: N_dataInd(pdu{#}: pduType);

OUTPUT userToPm.T_dataReq(sendMsg){|sendMsg};

sdu := sdu; {Wenn sdu sonst nicht in Transitionsblock}
OUTPUT pmToNet.N_dataReq(tmpMsg){|tmpMsg=sdu};
```

Interaktionsverfolgung

Zur Verfolgung einer Interaktion wird der Kommentar an die folgende Stelle gesetzt.

Syntax

```
    channel-definition = channel-heading channel-block.
         channel-block = +{ interaction-group } .
     interaction-block = +{ "by" role-identifier
                          [ "," role-identifier ] ":"
                         +{interaction-definition} .
interaction-definition = IDENTIFIER
                         [ "(" VALUE-PARAMETER-SPECIFICATION
                         { ";" VALUE-PARAMETER-SPECIFICATION } ")"
                         ["{#}"] ] ";".
```

Semantik Wird der Kommentar # gesetzt, so wird der Interaktion IDENTIFIER ein zusätzliches Feld zugeordnet, welches eine eindeutige Identifizierung einer verschickten Interaktion erlaubt.

Beispiele

```
CHANNEL transmitService(user, provider);
  BY user:     T_dataReq(sdu: sduType){#};

  BY provider: T_dataInd(sdu: sduType){#};

OUTPUT userToPm.T_dataReq(sendMsg);
```

Meßpunkt

Es können an beliebiger Stelle im Transitionsblock bzw. in den Prozeduren und Funktionen Meßpunkte gesetzt werden.

Syntax

```
    statement-part = compound-statement .
compound-statement = "begin" statement-sequence "end"
statement-sequence = statement [packet-com]
                     { ";"statement [packet-com] }.
        packet-com = "{!"com_name
                     [.structured-type-identifier] "}"
          com_name = IDENTIFIER .
```

Semantik An der Stelle, an der Kommentar steht, wird ein Meßpunkt eingesetzt. Er ist bei der Auswertung über *com_name* referenzierbar. Optional kann die Sequenznummer einer Variablen bei der Messung ausgegeben werden. Es ist dabei aber dasselbe wie weiter oben zu beachten.

Beispiele

```
  sdu := [Son]{!name.sdu};
count := count + 1{!state-x};
```

B.1.2 Optionen von Dingo

Der erweiterte Dingo besitzt einige zusätzliche Optionen, mit denen die Generierung der Meßpunkte beeinflußt werden kann.

Das Aufrufformat für ist das Folgende:

dingo [-r] [-m Zahl] [-M Datei] [-N File] Objekt-File

-m Mit Hilfe dieser Option wird aus der Menge der möglichen Meßpunkte eine Vorauswahl getroffen. Dabei ist die Zahl bestimmt als die oder-Verknüpfung des Meßmodells bzw. der Meßebene, aus welcher Meßpunkte erzeugt werden sollen. Die Kodierung ist dabei die folgende:

1 Spezifikationsebene

2 Subsystemebene

4 Modulebene

8 Transitionsebene

16 Meßmodell des Protokolls

-M Diese Option schränkt die Vorauswahl durch die m-Option noch weiter ein, indem nur die Meßpunkte gesetzt werden, die in der Datei übergeben werden. Die Datei enthält dabei eine Folge von Sclüsselnummern, welche die Meßpunkte identifizieren. Diese Schlüsselnummern erhält man, indem man die von Dingo erzeugte Monitor-Datenbank betrachtet. In dieser werden alle erzeugten Meßpunkte in dem folgenden Format aufgelistet:

Schlüsselnummer Typ-Kode Meßereignistyp String_1 [String_2]

Der Name der Datei, die die aktuelle Monitor-Datenbank enthält, wird beim Aufruf von Dingo ausgegeben.

-N Diese Option schränkt die Vorauswahl durch die m-Option noch weiter ein, indem die Meßpunkte **nicht** gesetzt werden, die in der Datei übergeben werden.

B.2 Bedienung von PATO

Die Bedienung von PATO erweist sich wegen der graphischen Benutzeroberfläche als einfach. Sie soll hier im einzelnen beschrieben werden. Die Bedienung untergliedert sich in zwei Bereiche, die Verwaltung der Meßereignisse und die Ermittlung der Leistungsmaße.

B.2.1 Verwaltung der Meßinformationen

Nachdem *PATO* gestartet wurde, erscheint nach dem Drücken des *Push-Buttons* **Laden** eine Filerequester, mit Hilfe dessen man die entsprechende Datei mit den Meßereignissen auswählen kann. Das Laden dieser Datei kann zu folgenden Fehlern führen:

- Wenn sich die durch *Dingo* erzeugte Datei mit den statischen Information bzgl. der Meßpunkte nicht im gleichen Verzeichnis befindet wie die ausgewählte Datei, wird die ausgewählte Datei nicht geladen.

- Falls die ausgewählte Datei inkorrekte Werte enthält, wie zum Beispiel Meßereignisse zu nicht vorhanden Meßpunkten, dann führt dies zu einer Fehlermeldung, die korrekten Werte werden, soweit es möglich ist, übernommen

Der Vorgang des Ladens wird solange iteriert, bis alle gewünschten Dateien geladen sind. Möchte man eine andere Teilmenge von Dateien, so können die bisher geladenen durch drücken des *Push-Buttons* **Löschen** entfernt werden. Das Beenden von *PATO* erfolgt über das Drücken des *Push-Buttons* **Quit**.

Um die Meßspezifische Information dargestellt zu bekommen, muß die entsprechende Datei in der Liste mit einem *Doppel-Klick* angewählt werden.

B.2.2 Ermittlung der Leistungsmaße

Um die Leistungsmaße zu ermitteln, müssen die entsprechenden Meßpunkte in der Liste mit einem *Doppel-Klick* ausgewählt werden. Falls nur eines ausgewählt wurde, kann nur die *Ausführungszeit* und die *Ausführungsrate* ermittelt werden. Bei der Auswahl von zweien kann zusätzlich die *Verzögerung* ermittelt werden. In diesem Fall beziehen sich aber die beiden ersteren Leistungsmaße nur auf den zuerst ausgewählten Meßpunkt. Möchte man eine andere Wahl treffen, so wird die alte Wahl überschrieben.

Das Drücken einer der sechs *Push-Buttons* führt zur Ermittlung des entsprechenden Leistungsmaßes. Es wird das entsprechende Histogramm dargestellt. Über die *Push-Button* $\gg$ und $\ll$ erfolgt das Verwerfen der entsprechenden Umgebung des größten und kleinsten Wertes. Die anderen *Push-Button* **Statistik** und **Liste** öffnen einen neuen Dialog, der die Statistik bzw. die Liste aller Werte (nach der Vorauswahl) darstellt. In diesen wiederum kann man ein Abspeichern der dargestellten Daten auf Datei veranlassen. Die standardmäßige Datei ist "pato.stat".

Bei betätigen des *Push-Buttons* **Option** kann bzgl. des Leistungsmaßes *Verzögerung* eine Veränderung der Voreinstellung getroffen werden.

Literaturverzeichnis

[Br94] Bredereke, J. , *Atomarität in parallel implementierten Estelle-Spezifikationen* PIK 17, 1994.

[BUX84] Werner Bux and Harry Rudin, *Performance of Computer-Communication Systems*, Proceedings of the IFIP WG 7.3/TC 6 Second International Symposium on the Performance of Computer-Communication Systems, Zürich, North-Holland, 1984, 557 S.

[EIC93] Eichim, R. , *Erweiterung des Estelle-C++-Compilers vom National Institute of Standards and Technology um Funktionen zur Laufzeitmessung*, Studienarbeit, Universität Mannheim, Lehrstuhl für Praktische Informatik IV, 1993, 69 S.

[FER83] Domenico Ferrari, Giuseppe Serazzi, Alessandro Zeigner *Measurement and Tuning of Computer Systems*, Prentice-Hall, 1983, 523 S.

[GBE95] Gotzhein, R. , Bredereke, J. , Effelsberg, W. , Fischer, S. , König, H. , Held, T. , *Improving the Efficiency of Automated Protocol Implementation Using Estelle*, Interner Bericht Nr. 274/95, Universität Kaiserslautern, Fachbereich Informatik, 1995, 19 S.

[HAA95] Haas, M. , *Methodische Leistungsanalyse von Rechensystemen* München, Wien, Oldenbourg, 1995.

[ISO89] *Estelle - A Formal Description Technique Based on an Extended State Transition Model* ISO/TC97/SC21, IS 9074, 1989, 179 S.

[LA90] Levi S. T. , Agrawala A. K. , *Real-time system design*, McGraw-Hill, New York, 1990, 299 S.

[LAN92] Horst Langendörfer, *Leistungsanalyse von Rechensystemen: Messen, Modellieren, Simulation*, Hanser Verlag, München, 1992, 339 S.

[LIP91] Lippman, S. B. , *C++ Einführung und Leitfaden*, Addison-Wesley, 1991, .

[MOA94] Heller, D. , Ferguson, P. , *Motif Programming Manual Volume 6A*, O'Reilly & Associates Inc, 1994, 972 S.

[MOB93] Ferguson, P. , *Motif Reference Manual Volume 6B*, O'Reilly & Associates Inc, 1993, 908 S.

[RiMaDe83] Riedewald, G. , Maluszynski, J. , Dembinski, P. , *Formale Beschreibung von Programmiersprachen - Eine Einführung in die Semantik*, R. Oldenbourg Verlag, München, 1983, 205 S.

[SIGA88] Sijelmassi, R. , Gaudette, P. , *An object-oriented model for Estelle*, Proceedings !st International Conference on Formal Description Techniques for Communication Protocols and Distributed Systems – FORTE 88, Stirling, Scotland, Noth-Holland, Amsterdam, 1989, 91 - 105.

[SIST91A] Sijelmassi, R. , Strausser, B. , *The portable Estelle translator: an overview and user guide, Technikal Report NCSL/SNA 91/3*, National Institute of Standarts and Technology, Gaithersburg, MD 20899, USA, 1991.

[SIST91B] Sijelmassi, R. , Strausser, B. , *The distributed implementation generator: an overview and user guide, Technikal Report NCSL/SNA 91/3*, National Institute of Standarts and Technology, Gaithersburg, MD 20899, USA, 1991.

[STE93] Stevens, W. R. , *Advanced programming in the UNIX-Enviroment*, Addison-wesley Publishing Company, 1994, 744 S.

[STE93] Stevens, W. R. , *UNIX Network Programming*, Prentice Hall, 1990, 772 S.

[VER89] Pramode K. Verma, *Performance Estimation of Computer Communication Networks - A Structured Approach*, computer science press, 1989, 133 S.