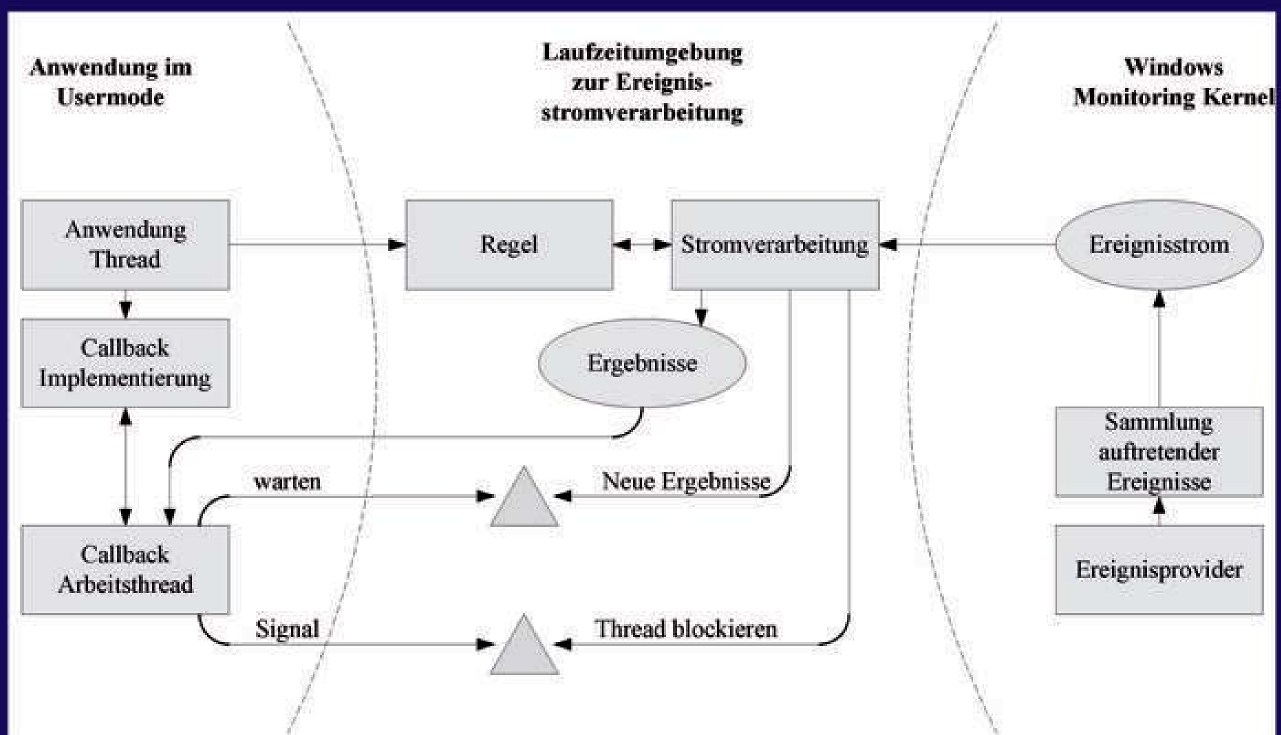
# VERARBEITUNG VON EREIGNISSTRÖMEN IM BETRIEBSSYSTEMKERN

Michael Schöbel, M.Sc.



Cuvillier Verlag Göttingen
Internationaler wissenschaftlicher Fachverlag

# Verarbeitung von Ereignisströmen im Betriebssystemkern

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
(Dr.-Ing.)

eingereicht im
Fachbereich Informatik
der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Michael Schöbel, M.Sc.

Potsdam, 5. Juli 2010

Betreuer: Prof. Dr. rer. nat. habil. Andreas Polze
Hasso-Plattner-Institut für Softwaresystemtechnik
Universität Potsdam

**Bibliografische Information der Deutschen Nationalbibliothek**
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.
1. Aufl. - Göttingen : Cuvillier, 2010
Zugl.: Potsdam, Univ., Diss., 2010

978-3-86955-601-7

© CUVILLIER VERLAG, Göttingen 2010
Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

Alle Rechte vorbehalten. Ohne ausdrückliche Genehmigung des Verlages ist es nicht gestattet, das Buch oder Teile daraus auf fotomechanischem Weg (Fotokopie, Mikrokopie) zu vervielfältigen.
1. Auflage, 2010
Gedruckt auf säurefreiem Papier

978-3-86955-601-7

# Zusammenfassung

Die Beobachtung von Systemverhalten durch Aufzeichnung von auftretenden Ereignissen im System (*event tracing*) wird in unterschiedlichen Bereichen eingesetzt - beispielsweise zur Fehlersuche, zur Leistungsanalyse oder zur Überwachung von Dienstgüteeigenschaften.

Herausforderungen ergeben sich dabei aus der großen Menge von aufzunehmenden Daten, einer unter Umständen starken Systembeeinflussung, der zeitlichen Lücke zwischen Analyse und möglicher Reaktion, sowie aus dem Detaillierungsgrad der aufgezeichneten Ereignisse.

Die Verarbeitung von Ereignisströmen direkt im Betriebssystemkern stellt einen neuen Ansatz zur Systemanalyse dar. In dieser Arbeit wird eine im Betriebssystemkern integrierte Laufzeitumgebung zur Verarbeitung von Ereignisströmen vorgestellt. Diese erlaubt die Erkennung von Konstellationen mehrerer Ereignisse in unterschiedlichen Ausführungskontexten im systemweiten Ereignisstrom, die sofortige Verarbeitung und Analyse direkt während des Auftretens der Ereignisse und die Ausführung von Aktionen sofort nach der Erkennung bestimmter Ereigniskonstellationen.

Mit Hilfe des beschriebenen Ansatzes kann die Menge der aufzuzeichnenden Daten reduziert werden, indem nur die Informationen aufgezeichnet werden, die tatsächlich für die Erkennung einer spezifischen Konstellation notwendig sind. Die Integration in den Betriebssystemkern ermöglicht die Analyse von nahezu beliebigen Anwendungen, die auf einem System mit einem solchen Kern ausgeführt werden.

Die Laufzeitumgebung und die verwendete Instrumentierungsinfrastruktur sind als Prototyp für den Windows Betriebssystemkern implementiert worden. Weiterhin wird eine Spezifikationssprache zur Beschreibung von Ereigniskonstellationen dargestellt. Die Besonderheiten der Ereignisstromverarbeitung im Betriebssystemkern werden analysiert und berücksichtigt.

Der in dieser Arbeit beschriebene Ansatz zur Verarbeitung von Ereignisströmen im Betriebssystemkern ermöglicht beispielsweise die Implementierung eines Softwareprüfstands oder die Umsetzung von Konzepten des Autonomic Computing.

# Abstract

Monitoring system behaviour by recording occurring events is used in different application domains - for example for debugging, for performance analysis, or for controlling the quality of service properties.

The main challenges of event tracing are the huge amount of data, a high degree of system disturbance, the time gap between analysis and reaction, and the level of detail of the recorded events.

The processing of event streams directly in the operating system kernel is a new approach for system analysis. In this work an operating system kernel integrated runtime environment is proposed. This runtime environment allows the detection of event constellations in the global event stream across different execution contexts. Events can be analyzed at the very same moment they occur. Furthermore, the runtime environment can trigger actions if a specific event constellation is detected.

By considering only the information necessary to detect specific constellations, the described approach helps to reduce the amount of data to record. The integration into the operating system kernel allows analyzing almost every application executed in an environment using this kernel.

The runtime environment and the underlying instrumentation infrastructure were prototypically implemented for the Windows operating system. Furthermore, in this work a specification language for describing event constellations was developed. Particularities of event stream processing in an operating system kernel were analyzed and considered in the implementation.

The described approach for processing event streams in an operating system kernel enables the implementation of a software testbed or the implementation of autonomic computing concepts.

# Danksagung

An dieser Stelle möchte ich einigen Menschen danken, die wesentlich zum Gelingen der vorliegenden Arbeit beigetragen haben - an erster Stelle meinem Betreuer Prof. Dr. Andreas Polze, der diese Arbeit ermöglichte und mich mit Anregungen und Hinweisen unterstützte. Auch die gesamte Arbeitsgruppe „Betriebssysteme und Middleware“ trug mit Diskussionen und Vorschlägen zur Verbesserung der Arbeit bei.

Weiterhin möchte ich mich ganz allgemein beim Hasso-Plattner-Institut bedanken. Das HPI ermöglichte mein Studium und die Anfertigung dieser Arbeit im Rahmen des Forschungskollegs „Service-Oriented Systems Engineering“ - in einer erstklassigen Umgebung.

Insbesondere möchte ich mich bei den Kommilitonen und Kollegen Benjamin Hagedorn und Stephan Kluth bedanken, die immer für Gespräche zu fachlichen und privaten Themen bereit standen. Außerdem bedanke ich mich bei Alexander Schmidt für die gute Zusammenarbeit im Rahmen des „Windows Research Kernels“.

Nicht zuletzt möchte ich mich auch bei meiner Familie und insbesondere bei meiner Frau Valerie für die Korrektur der Arbeit und die ständige Unterstützung bedanken.

Potsdam, 5. Juli 2010

# Inhaltsverzeichnis

**1 Einleitung** **1**

1.1 Kontext . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Problemdefinition . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Anwendungsszenarien . . . . . . . . . . . . . . . . . . . . . . . . 4

1.4 Beitrag dieser Arbeit . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.5 Struktur der Arbeit . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.6 Hinweise zur erstellten Software . . . . . . . . . . . . . . . . . . . 7

1.7 Hinweise zu Sprache und Notation . . . . . . . . . . . . . . . . . 7

**2 Grundlagen der Überwachung und Instrumentierung von Softwaresystemen** **9**

2.1 Klassifizierung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.1.1 Ansatzpunkte für Überwachung und Instrumentierung . . . . . . . . . . 10

2.1.2 Sammlung von Analysedaten . . . . . . . . . . . . . . . . . . . 11

2.1.3 Auswertung und Analyse . . . . . . . . . . . . . . . . . . . . . 13

2.1.4 Zusammenfassende Darstellung . . . . . . . . . . . . . . . . . . 13

2.2 Instrumentierung . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.2.1 Instrumentierungspunkte . . . . . . . . . . . . . . . . . . . . . 15

2.2.2 Statische Instrumentierung . . . . . . . . . . . . . . . . . . . . 16

2.2.3 Dynamische Instrumentierung . . . . . . . . . . . . . . . . . . 17

2.3 Automatisierte Überwachung . . . . . . . . . . . . . . . . . . . . 17

2.3.1 Regelkreis-basierte Überwachung . . . . . . . . . . . . . . . . . 18

2.3.2 Autonomic Computing . . . . . . . . . . . . . . . . . . . . . . 22

2.4 Begriffsbestimmung: Ereignis und Ereignisstrom . . . . . . . . . . . 25

2.4.1 Definition: Ereignis . . . . . . . . . . . . . . . . . . . . . . . . 25

2.4.2 Definition: Ereignisstrom . . . . . . . . . . . . . . . . . . . . . 27

2.5 Zusammenfassung . . . . . . . . . . . . . . . . . . . . . . . . . . 28

3 **Der Windows Monitoring Kernel** 29
3.1 Zielstellung der Implementierung . . . . . 29
3.1.1 Anforderungen . . . . . 29
3.1.2 Architektur . . . . . 30
3.2 Ereigniserzeugung . . . . . 31
3.2.1 Grundlagen . . . . . 31
3.2.2 Betriebssystemkernereignisse . . . . . 33
3.2.3 Anwendungsspezifische Ereignisse . . . . . 35
3.2.4 Ereignisse in DLLs . . . . . 38
3.2.5 Ereignisse in Treibern . . . . . 38
3.3 Ereignisaufzeichnung . . . . . 38
3.3.1 Pufferverwaltung . . . . . 38
3.3.2 Synchronisation . . . . . 40
3.3.3 Logdateiverwaltung . . . . . 42
3.4 Werkzeuge . . . . . 43
3.4.1 Steuerung der Ereignisaufzeichnung . . . . . 43
3.4.2 Auswertung von Logdateien . . . . . 43
3.5 Evaluation . . . . . 46
3.5.1 Testumgebung und Methodologie . . . . . 46
3.5.2 Messungen und Analysen . . . . . 47
3.5.3 Vergleich mit alternativen Systemen . . . . . 50
3.6 Zusammenfassung . . . . . 52

4 **Online Verarbeitung von Ereignissen** 53
4.1 Zielstellung der Implementierung . . . . . 53
4.2 Beschreibung von Mustern in Ereignisströmen . . . . . 54
4.2.1 Ereignistypen . . . . . 55
4.2.2 Konstellation . . . . . 58
4.2.3 Relationen . . . . . 62
4.2.4 Weitere Sprachelemente . . . . . 64
4.3 Automaten zur Mustererkennung . . . . . 65
4.3.1 Grundlagen . . . . . 65
4.3.2 Automatenerzeugung . . . . . 67
4.3.3 Compiler . . . . . 75
4.4 Laufzeitumgebung zur Mustererkennung . . . . . 80

4.4.1 Abarbeitungsmodell . . . 81
4.4.2 Verarbeitung von Ereignissen . . . 82
4.4.3 Verwaltung von Regeln . . . 84
4.4.4 Pufferverwaltung und Synchronisation . . . 85
4.4.5 Systemaufruf Schnittstelle . . . 88
4.5 Reaktion auf erkannte Muster . . . 89
4.5.1 Kernelmode Skripte . . . 89
4.5.2 Usermode Callbacks . . . 93
4.6 Werkzeuge . . . 96
4.7 Evaluation . . . 97
4.7.1 Messungen und Analysen . . . 97
4.7.2 Vergleich mit alternativen Systemen . . . 99
4.8 Zusammenfassung . . . 100

**5 Fallstudien und Leistungsbewertung** **101**
5.1 Einleitung . . . 101
5.2 Analyse der Bearbeitung von Anfragen an einen Webserver . . . 101
5.2.1 Instrumentierte Ausführung . . . 101
5.2.2 Ergebnisse . . . 102
5.2.3 Diskussion . . . 105
5.3 Analyse von Vorgängen im Betriebssystemkern . . . 105
5.3.1 Bootvorgang . . . 105
5.3.2 Quantumlängen und Scheduling . . . 107
5.3.3 Diskussion . . . 109
5.4 Softwareprüfstand . . . 110
5.4.1 Erkennung von Wartezeiten . . . 111
5.4.2 Erkennung von Fehlern . . . 112
5.4.3 Analyse von Synchronisationsvorgängen . . . 112
5.4.4 Analyse von Seitenzugriffsfehlern . . . 114
5.4.5 Überwachung von Annahmen und Grenzwerten . . . 114
5.4.6 Diskussion . . . 115
5.5 Adaption des Betriebssystems . . . 115
5.6 Online Verarbeitung von Ereignissen - Komplexitätsanalyse . . . 116
5.6.1 Charakterisierung von Ereignisströmen . . . 117
5.6.2 Modellierung der Ereignisstromverarbeitung . . . 119

5.6.3 Diskussion . . . . . . . . . . 123
5.7 Weitere Untersuchungen . . . . . . . . . . 123
5.7.1 Fein-granulare Zuteilung von CPU Zeit . . . . . . . . . . 123
5.7.2 Instrumentierung von Spinlocks . . . . . . . . . . 126
5.8 Zusammenfassung . . . . . . . . . . 130

**6 Verwandte Arbeiten** **131**
6.1 Instrumentierungstechniken . . . . . . . . . . 131
6.1.1 Klassifikation . . . . . . . . . . 131
6.1.2 Implementierungen . . . . . . . . . . 131
6.1.3 Diskussion - Einordnung des WMK . . . . . . . . . . 138
6.2 Verarbeitung von Ereignisströmen . . . . . . . . . . 139
6.2.1 Forschungskontext . . . . . . . . . . 139
6.2.2 Sprachen zur Beschreibung von Ereigniskonstellationen . . . . . . . . . . 141
6.2.3 Diskussion - Einordnung der entwickelten Laufzeitumgebung . . . . . . . . . . 143
6.3 Zusammenfassung . . . . . . . . . . 143

**7 Zusammenfassung und Ausblick** **145**

**Literaturverzeichnis** **149**

**A WMK Ereignistypen** **157**

# Kapitel 1

# Einleitung

*In komplexen Softwaresystemen erlaubt die Aufzeichnung von auftretenden Ereignissen Rückschlüsse auf Vorgänge und Zusammenhänge innerhalb des beobachteten Systems. Grundprobleme sind dabei die bei der Aufzeichnung entstehende große Datenmenge und die starke Systembeeinflussung. Das Ziel der in dieser Arbeit vorgestellten Ansätze ist es, zur Ausführungszeit eines Softwaresystems eine effiziente Aufzeichnung und Analyse von Ereignissen zu ermöglichen.*

*Im weiteren Verlauf dieses Kapitels werden der Kontext der Arbeit und die betrachtete Problemstellung dargestellt. Aufbau und Struktur der Arbeit sowie die Beiträge der Arbeit werden beschrieben. Weiterhin sind grundsätzliche Anmerkungen zu Sprache und Notation der Arbeit enthalten.*

## 1.1 Kontext

Die Beobachtung von komplexen Softwaresystemen durch Aufzeichnung von auftretenden Ereignissen im System (*event tracing*) wird in unterschiedlichen Bereichen eingesetzt:

**Systemanalyse:** Die Untersuchung aufgezeichneter Ereignisfolgen kann zu einem verbesserten Verständnis von Vorgängen in einem System führen, beispielsweise von Vorgängen im Betriebssystemkern [48] oder in einem komplexeren Gesamtsystem [27]. Ganz allgemein können aus dem beobachteten Verhalten Rückschlüsse auf beteiligte Komponenten und Zusammenhänge im System gezogen werden.

**Fehleranalyse:** Einige Problemklassen, beispielsweise Performanzprobleme, lassen sich nur zur Laufzeit des betroffenen Systems analysieren. Die Aufzeichnung von Interaktionen zwischen beteiligten Systemkomponenten [19, 38] kann die Fehlersuche unterstützen. Die Analyse soll also Erkenntnisse über die Ursache von Problemen erbringen.

**Überwachung zur Laufzeit:** Die kontinuierliche Aufzeichnung von Ereignissen während der Systemlaufzeit kann angewandt werden, um beispielsweise die Einhaltung von gewünschtem Antwortzeitverhalten [21] zu überwachen oder um Einbrüche in das System festzustellen [66].

Bei den vorgestellten Ansätzen wird eine konkrete Implementierung eines Systems instrumentiert, das heißt die aufzuzeichnenden Ereignisse werden ausgewählt. Anschließend erfolgt die Systembeobachtung - eine reguläre oder leicht modifizierte Anwendung wird ausgeführt und die angezeigten Ereignisse werden gespeichert. Basierend auf den gespeicherten Ereignissen

werden Rückschlüsse auf das Systemverhalten gezogen und gegebenenfalls die Anwendung aufgrund der Beobachtungsergebnisse angepasst. Dabei steht - im Gegensatz zur Modellprüfung oder der Systemverifikation - die Analyse des tatsächlichen Verhaltens einer vorliegenden Systemimplementierung im Vordergrund und nicht der Vergleich mit einem Modell oder eine Spezifikation.

Je nach Art der durchzuführenden Analyse kann die Auswertung der aufgezeichneten Ereignisse zeitaufwändig und komplex sein. Weitere Einschränkungen ergeben sich aus der Systeminstrumentierung und der zu bewältigenden Datenmenge: das Systemverhalten wird durch die Instrumentierung beeinflusst und zahlreiche auftretende Ereignisse müssen zuverlässig, das heißt möglichst vollständig, aufgezeichnet werden.

Zur Überwachung und Analyse stehen verschiedene Werkzeug-Typen zur Verfügung. Ein Überblick wird in Kapitel 2 gegeben. In der vorliegenden Arbeit werden Werkzeuge zur *dynamischen Analyse* von Systemen betrachtet. Mit statischer Analyse werden Systeme untersucht, die sich nicht in Ausführung befinden - beispielsweise basierend auf dem Quelltext einer Anwendung. Die dynamische Analyse erlaubt die Untersuchung vom Laufzeitverhalten eines Systems. Werkzeuge zur dynamischen Analyse können in *sampling-basiert* und *tracing-basiert* eingeteilt werden. Sampling-basierte Werkzeuge bestimmen den Zustand des analysierten Systems periodisch und analysieren das Verhalten des Systems mit Methoden der Statistik. Tracing-basierte Werkzeuge analysieren, wie beschrieben, (vollständige) Ketten von Ereignissen im System.

Beide Werkzeugtypen haben spezifische Vor- und Nachteile: Tracing-basierte Werkzeuge erfordern die Erzeugung von Ereignissen im zu analysierenden System. Der entsprechende Programmcode muss entweder bereits in der Software vorhanden sein (und bei Bedarf aktiviert werden) oder zur Laufzeit eingefügt werden. Weiterhin ist es im Allgemeinen nicht möglich die Anzahl der im Rahmen der Analyse erzeugten Ereignisse vorherzusagen. Demnach ist auch der Einfluss der Analyse auf das beobachtete System nicht vorhersagbar.

Sampling-basierte Werkzeuge erfordern Messpunkte im analysierten System, die periodisch abgefragt werden können. Der Einfluss der Analyse kann durch Einstellen der Periode bestimmt werden. Sampling von Systemzuständen kann jedoch Änderungen im System nicht erfassen, die zwischen zwei Messzeitpunkten auftreten. Weiterhin gilt, dass manche Vorgänge im System sich nicht durch eine Folge von Messwerten beschreiben oder erklären lassen.

Nach abgeschlossener Analyse wird gegebenenfalls das System verändert, um auf identifizierte Probleme zu reagieren. Entsprechende Systemänderungen sind beispielsweise eine Rekonfiguration der Systembestandteile, das Einspielen von Patches oder der Neustart von Systemteilen.

In der Regel erfolgen die Analyse und die resultierende Anpassung des Systems zeitlich getrennt und werden, beispielsweise in den Bereichen der System- und Fehleranalyse, manuell und nur bei Bedarf ausgeführt.

In der vorliegenden Arbeit werden tracing-basierte Konzepte zur Systembeobachtung und -anpassung untersucht. Es wird eine generische Architektur entwickelt, die es erlaubt tracing-basierte Systemanalyse in Betriebssystemen und Anwendungen zu verwenden. Die Grundidee ist dabei (Werkzeug-unterstützt oder manuell) Beschreibungen von Ereigniskonstellationen zu spezifizieren, die Reaktionen auslösen. Eine solche Reaktion kann beispielsweise ein Eintrag in einer Logdatei, der Neustart einer Systemkomponente oder eine Rekonfiguration des Systems sein.

## 1.2 Problemdefinition

Die Verwaltung komplexer Systeme ist eine zeit- und arbeitsintensive Aufgabe - ständige Änderungen der Rahmenbedingungen einer Anwendung erfordern ständige Anpassung der Konfiguration und Ausführung der Systeme. Konzepte zur Optimierung und Automatisierung des Zyklus aus Analyse und Systemanpassung werden daher wichtiger um selbst-adaptive Anwendungen zu implementieren.

Um die Systemverwaltung zu automatisieren, werden heute vorwiegend Regelkreis-basierte Ansätze verwendet: aus gemessenen Werten werden mit Hilfe eines Systemmodells neue Konfigurationsparameter bestimmt, die dann über Aktuatoren gesetzt werden. Regelkreis-basierte Konzepte haben jedoch Nachteile, die sich aus dem *Sampling* des Systemzustandes ergeben: Tritt eine Systemzustandsänderungen zwischen zwei Messzeitpunkten auf, kann diese unter Umständen nicht erfasst werden. Weiterhin lassen sich nicht alle Aspekte der Systemverwaltung auf die Erfassung und Beeinflussung einzelner Messwerte abbilden.

*Tracing* bietet demgegenüber verschiedene Vorteile: Werden entsprechende Ereignisse generiert, sind Vorgänge wie der Ausfall von Systemkomponenten oder Angriffe einfacher zu erkennen. Auch für die Systemanalyse ist Tracing hilfreich, da Ketten von Ereignissen für das Systemverständnis mehr Informationen enthalten können als die Beobachtung der Entwicklung eines Messwertes im Laufe der Zeit.

Die Verwendung von Tracing zur Überwachung von Systemen zeigt jedoch ebenfalls spezifische Nachteile:

- Im Allgemeinen verursacht Tracing einen hohen Overhead, also zusätzliche Last im System. Bei sehr vielen auftretenden Ereignissen kann das System unter Umständen unbenutzbar werden.
- Die Aufzeichnung von Ereignissen führt (je nach Anzahl und Typ der Ereignisse) zu sehr großen Datenmengen. Diese müssen gespeichert und anschließend analysiert werden.
- Das (automatisierte) Analysieren der Ereignisströme erfordert komplexe Mustererkennung. Je nach konkretem Analyseziel müssen unter Umständen sehr große Datenmengen nach seltenen Ereigniskonstellationen durchsucht werden.
- Viele Werkzeuge führen die Analyse von Ereignisströmen zeitlich getrennt von der Aufzeichnung, also nachträglich, durch. Dies verhindert eine schnelle Reaktion auf erkannte Ereigniskonstellationen.

In der vorliegenden Arbeit sollen diese Tracing-spezifischen Nachteile untersucht und Lösungsmöglichkeiten aufgezeigt werden. Ziel ist es, die Vorteile von Tracing zur automatisierten Systemanalyse verfügbar zu machen.

Dabei wird nicht angestrebt Sampling-basierte Werkzeuge durch Tracing-basierte zu ersetzen. Vielmehr sollen die Tracing-spezifischen Nachteile in Szenarien, in denen der Einsatz Tracing-basierter Werkzeuge sinnvoll ist, verringert werden.

Fragestellungen, die in diesem Zusammenhang untersucht werden, sind beispielsweise: Wie kann die Menge der Daten, die beim Tracing aufgezeichnet werden muss, reduziert werden? Wie kann schnell auf erkannte Muster im Ereignisstrom reagiert werden?

Der Grundgedanke der in dieser Arbeit vorgestellten Ansätze ist dabei, dass fein-granulare Ereignisse im Betriebssystemkern zu komplexeren Ereignissen auf einer höheren Abstraktionsebene zusammengefasst analysiert werden können. Die vorgestellten Konzepte erlauben die Beschreibung und Erkennung von Ereigniskonstellationen, die solche komplexe Ereignisse bilden. Neben der Erkennung dieser Ereigniskonstellationen zur Laufzeit soll auch eine sofortige Reaktion ermöglicht werden. Dies ermöglicht dann Tracing-basierte Ansätze zur automatischen Systemadaption zu verwenden, beispielsweise im Rahmen von selbst-adaptiven Anwendungen.

## 1.3 Anwendungsszenarien

Der Schwerpunkt der vorliegenden Arbeit liegt auf der Entwicklung und Implementierung einer Laufzeitumgebung zur Verarbeitung von Ereignisströmen im Betriebssystemkern. Die Laufzeitumgebung ermöglicht die sofortige Reaktion auf erkannte Ereigniskonstellationen im Ereignisstrom.

Diese Konzepte bilden die Grundlage für zahlreiche Anwendungmöglichkeiten, von denen einige in diesem Abschnitt kurz dargestellt werden. Ausgewählte Szenarien werden im Rahmen von Machbarkeitsstudien im Kapitel 5 untersucht, andere erfordern über diese Arbeit hinausgehende Forschung.

**Softwareprüfstand:** Phänomene wie überlastete Synchronisationsobjekte (*lock contention*) oder Prioritäteninvertierung lassen sich meist erst zur Laufzeit des Systems unter realen Lastbedingungen erkennen. Mit einem Prüfstand, der nach allgemein bekannten, ungünstigen Ereigniskonstellationen zur Laufzeit sucht, kann ein gegebenes System bezüglich solcher Probleme analysiert werden.

**Überwachung von Anwendungen:** Während der Laufzeit eines Systems können verschiedene Ereigniskonstellationen auftreten, die ganz unterschiedliche Aktionen erfordern: Bei einem erkannten Angriff auf das System können Gegenmaßnahmen eingeleitet werden und beispielsweise die verdächtige Aktivität in einen isolierten Bereich verschoben werden. Bei der Anmeldung eines Benutzers und der Ausführung von Funktionen kann beispielsweise geprüft werden, ob der Benutzer diese Funktion tatsächlich ausführen darf oder die Funktionsausführung protokolliert werden.

**Selbst-adaptive Systeme:** Eine Folge von Ereignissen kann als Basis einer Rekonfiguration des Systems dienen, indem beispielsweise eine als fehlerhaft erkannte Systemkomponente ersetzt wird. Eine weitere Möglichkeit ist die Umschaltung zwischen verschiedenen Implementierungsstrategien, wenn die auftretenden Ereignisse im System ineffizientes Verhalten anzeigen oder sich wichtige Umgebungsbedingungen geändert haben.

**Skripting:** Durch die sofortige Ausführung von Aktionen nach der Erkennung von bestimmten Ereigniskonstellationen kann das Gesamtsystemverhalten beeinflusst werden. Auf diese Weise können bestimmte Verhaltensweisen erzwungen werden oder nachträglich Funktionalität im System ergänzt werden.

Mit den dargestellten Anwendungsmöglichkeiten leistet die entwickelte Laufzeitumgebung einen Beitrag zur effizienten Verwaltung und Entwicklung komplexer Systeme auf verschiedenen Ebenen - vom Betriebssystemkern bis zur Anwendung selbst.

## 1.4 Beitrag dieser Arbeit

Im Rahmen dieser Arbeit wurde eine Laufzeitumgebung für die effiziente Verarbeitung von Ereignisströmen auf der Ebene des Betriebssystemkerns entwickelt. Die Laufzeitumgebung ermöglicht dabei:

1. die Erkennung von Konstellationen mehrerer Ereignisse im systemweiten Ereignisstrom
2. die sofortige Verarbeitung und Analyse direkt während des Auftretens der Ereignisse
3. die Ausführung von Aktionen sofort nach der Erkennung bestimmter Ereigniskonstellationen

Die Möglichkeit zur synchronen Reaktion auf erkannte Ereigniskonstellationen schließt die üblicherweise vorhandene Lücke in tracing-basierter Systemüberwachung zwischen Systemanalyse und möglicher Reaktion, also der Anpassung des Systems: die Anpassung erfolgt *nicht* post-mortem, sondern bei weiterhin aktivierter Systemüberwachung. Außerdem kann bei sofortiger Analyse des Ereignisstroms die aufzuzeichnende Datenmenge reduziert werden und auf eine explizite Speicherung aller auftretender Ereignisse verzichtet werden - die Laufzeitumgebung sammelt ausschließlich für die zu erkennenden Ereigniskonstellationen relevante Informationen.

Die Bestandteile der Arbeit sind im Einzelnen:

**Instrumentierung:** Der *Windows Monitoring Kernel* wird vorgestellt, eine Infrastruktur zur Aufzeichnung und Analyse von Ereignissen im Windows Betriebssystemkern.

**Sprachspezifikation:** Es wird eine Sprache zur Spezifikation von Regeln vorgestellt, die es ermöglicht Konstellationen von Ereignissen zu beschreiben und mit Aktivitäten zu verknüpfen, die ausgeführt werden sollen, wenn entsprechende Konstellationen in einem Ereignisstrom erkannt werden.

**Besonderheiten im Umfeld der Betriebssysteme:** Ereignisstromverarbeitung wird bisher vor allem in Geschäftsanwendungen eingesetzt. Die Integration in einen Betriebssystemkern unterscheidet sich davon erheblich. Unterschiede in den Bereichen Ereignisbeschreibung und Implementierungsmöglichkeiten werden identifiziert. Die Besonderheiten werden verwendet, um neue Verarbeitungsmöglichkeiten zu realisieren.

**Implementierung:** Die Konzepte wurden implementiert; die entstandene Laufzeitumgebung zur Ereignisstromverarbeitung im Betriebssystemkern ermöglicht die effiziente Erkennung von Ereigniskonstellationen auch bei hohen Ereignisraten.

**Modellierung und Komplexitätsabschätzung:** Mit Hilfe stochastischer Modelle von Ereignisströmen in einem System und absorbierender Markov-Ketten als Modell für die Ereignisstromverarbeitung wurden Eigenschaften der entwickelten Laufzeitumgebung analysiert.

**Anwendung:** Beispielanwendungen und komplexere Fallstudien werden vorgestellt und damit die praktische Anwendbarkeit der entwickelten Konzepte belegt.

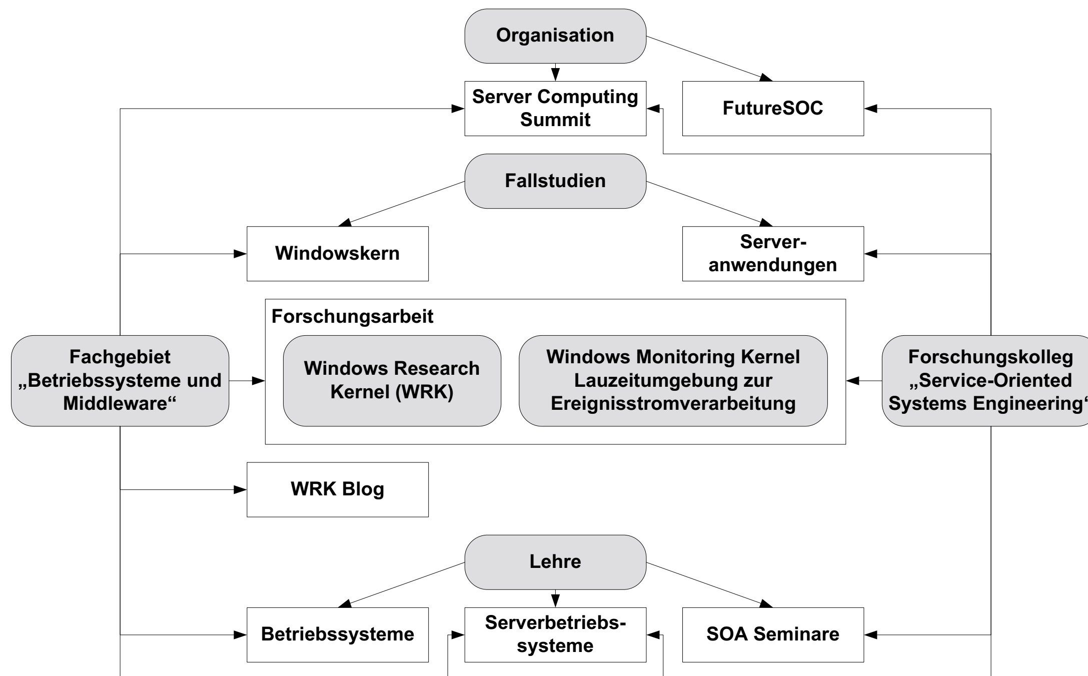



**Abbildung 1.1:** Umfeld der Arbeit

Die Integration der Konzepte in den Betriebssystemkern erlaubt die direkte Verwendung der entwickelten Konzepte in einer Vielzahl von Anwendungen: die Programmsysteme oberhalb des Betriebssystems müssen nicht oder nur geringfügig angepasst werden.

Die Arbeit wurde im Rahmen des HPI Forschungskollegs „Service-Oriented Systems Engineering“ am Fachbereich für Betriebssysteme und Middleware angefertigt. Dementsprechend war der Autor auch an Aktivitäten im Kolleg und am Fachbereich beteiligt. Einen Überblick zeigt Abbildung 1.1. Beispielhaft sei hier die Mitorganisation des jährlichen Forschungskolleg-Symposiums „Future Trends in Service-Oriented Computing“, des „Server-Computing Summits“ am HPI, und die Beteiligung am Lehrbetrieb verschiedener Vorlesungen und Seminare genannt. Weiterhin schrieb der Autor zahlreiche Beiträge zum *Windows Research Kernel (WRK)* Blog „WRK@HPI“ [13], beispielsweise zur Implementierung neuer Systemaufrufe für den Windows Kern.

## 1.5 Struktur der Arbeit

In diesem Kapitel wurde der Kontext der Arbeit dargestellt und die Problemstellung erläutert. Weiterhin wurde der wissenschaftliche Beitrag der Arbeit zusammenfassend dargestellt.

Der verbleibende Teil der Arbeit ist wie folgt strukturiert:

**Kapitel 2** stellt die Grundlagen der Arbeit dar. Konzepte auf dem Gebiet der Systeminstrumentierung und -überwachung werden vorgestellt und definiert.

**Kapitel 3** beschreibt den im Rahmen dieser Arbeit entwickelten *Windows Monitoring Kernel* (WMK), einer Instrumentierungsinfrastruktur für Windows Systeme.

**Kapitel 4** beschreibt die entwickelten Ansätze für die Online-Verarbeitung von Ereignissen durch Mustererkennung und Reaktionen.

**Kapitel 5** zeigt anhand von Fallstudien, wie die entwickelten Konzepte praktisch eingesetzt werden können.

**Kapitel 6** grenzt die Ergebnisse dieser Arbeit gegen verwandte Arbeiten ab und vergleicht die entwickelten Konzepte mit Alternativen.

**Kapitel 7** fasst die Ergebnisse zusammen und bietet einen Ausblick auf weiterführende Fragestellungen.

Ergänzende Informationen zu den entwickelten Konzepten und Werkzeugen werden im Anhang gegeben.

## 1.6 Hinweise zur erstellten Software

Die im Rahmen dieser Arbeit entwickelten Konzepte und Algorithmen wurden innerhalb des *Windows Research Kernels (WRK)* implementiert und praktisch erprobt.

Im Sommer 2006 wurde der Quelltext des Betriebssystemkerns von Windows Server 2003 Enterprise Edition durch Microsoft für universitäre Einrichtungen zugänglich gemacht. Der Einsatz von Windows in Forschung und Lehre sollte dadurch verbessert werden [82].

Die Ergebnisse dieser Arbeit konnten in den Windows Kern integriert werden und dadurch mit praktisch relevanten Serversystemen evaluiert werden. Teile dieser Arbeit wurden durch Drittmittel von Microsoft unterstützt (Grant No. 15899).

Die entwickelten Konzepte und Algorithmen sind jedoch nicht Windows-spezifisch und lassen sich auf andere Betriebssysteme übertragen und in ähnlicher Form realisieren.

Dies gilt auch für das verwendete Instrumentierungssystem. In der Arbeit wurde der *Windows Monitoring Kernel (WMK)* entwickelt und verwendet. Die vorgestellten Konzepte zur Verarbeitung von Ereignissen sind ebenfalls auf andere Instrumentierungssysteme übertragbar.

## 1.7 Hinweise zu Sprache und Notation

Die Sprache der vorliegenden Arbeit ist Deutsch, daher wird versucht deutsche Begriffe zu verwenden. An Stellen, an denen dies nicht möglich ist, sei es, weil der deutsche Begriff missverständlich oder zu ungewöhnlich ist, oder sei es, weil es keine deutsche Entsprechung gibt, wird der englische Fachbegriff verwendet und *kursiv* dargestellt.

Bei Quellcode-Listings und Programmausgaben wurde auf eine Übersetzung verzichtet, das heißt Variablennamen und Kommentare werden auf Englisch dargestellt.

Kapitel 2

# Grundlagen der Überwachung und Instrumentierung von Softwaresystemen

*In diesem Kapitel werden grundlegende Konzepte und Ansätze der Überwachung und Instrumentierung von Softwaresystemen dargestellt. Weiterhin werden in einer Begriffsbestimmung die Konzepte Ereignis und Ereignisstrom definiert und damit die Grundlage für die weiteren Ausführungen der Arbeit dargestellt.*

## 2.1 Klassifizierung

In dieser Arbeit werden Werkzeuge zur Beobachtung von Softwaresystemen analysiert und entwickelt. Als ein Softwaresystem wird dabei (im weitest möglichen Sinne) jedes Programm verstanden, dass auf einem Computer ausgeführt werden kann - von Betriebssystemkernen bis zu Webserver-Implementierungen.

Beobachtung eines Softwaresystems heißt, das Verhalten der Software zu bestimmen - das Verhalten in verschiedenen Kontexten. Ein Kontext wird durch die Umgebung des Softwaresystems vorgegeben und kann beispielsweise die Initialisierung des Systems oder die Bearbeitung einer Anfrage sein.

Das Verhalten einer Software lässt sich mit Hilfe verschiedener Aspekte beschreiben. Eine Möglichkeit ist, den Verlauf von Metriken über die Zeit in einem bestimmten Kontext aufzuzeichnen. Eine andere Möglichkeit ist, Vorgänge im System in Form von Ereignissen aufzuzeichnen. Beide Möglichkeiten zur Beschreibung von Verhalten sind automatisiert durchführbar. Verhaltensbeschreibungen, die nicht automatisiert zu erfassen sind (zum Beispiel informale Erklärungen der Softwareentwickler) sind nicht Gegenstand der vorliegenden Arbeit.

**Definition (Systembeobachtung)** *Systembeobachtung beschreibt den Vorgang, (Teil-) Aspekte des Verhaltens aufzuzeichnen und in einer Form aufzubereiten, die analysiert werden kann.*

Systembeobachtung kann mit unterschiedlichen Zielsetzungen erfolgen: Durch Analyse der Verhaltens kann versucht werden, das System besser zu verstehen; beispielsweise in welchen Kontexten welche Aktivitäten durchgeführt werden, welche Schnittstellen wann verwendet werden oder welche Komponenten im System existieren. Darüber hinaus kann Verhaltensanalyse dazu verwendet werden, Probleme im Softwaresystem zu identifizieren, beispielsweise

durch Analyse fehlerhaften Verhaltens in bestimmten Kontexten. Eine weitere Klasse von Zielsetzungen ist die kontinuierliche Überwachung des Systems zur Laufzeit; beispielsweise können Leistungsvorgaben ständig überwacht werden.

### 2.1.1 Ansatzpunkte für Überwachung und Instrumentierung

Die automatisierte Systembeobachtung erfordert Instrumentierung.

**Definition (Instrumentierung)** *Instrumentierung ist die Bereitstellung/Verwendung von Instrumenten/Messpunkten, an denen Informationen erfasst werden können, die Rückschlüsse auf den aktuellen Systemzustand erlauben.*

Instrumentierung kann dabei auf verschiedenen Ebenen angewendet werden. Abbildung 2.1 zeigt fünf mögliche Ansatzpunkte zur Instrumentierung in einem verallgemeinerten Systemmodell.

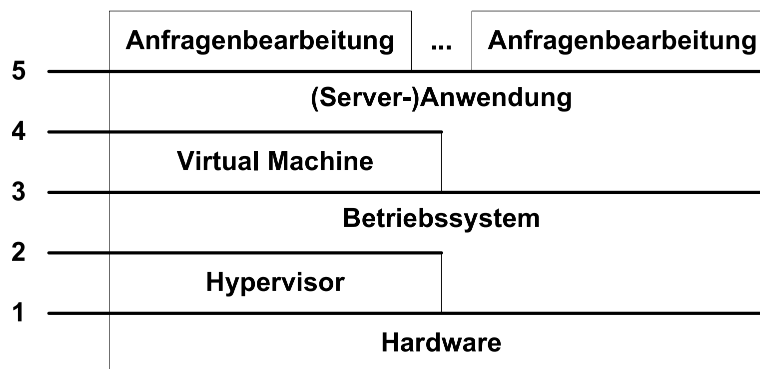



**Abbildung 2.1:** Ansatzpunkte zur Instrumentierung

Das System besteht abstrakt aus den Schichten Hardware, Betriebssystem, Anwendung und Anfrage. Das Betriebssystem kann dabei noch durch einen Hypervisor und die Anwendung durch eine virtuelle Maschine (zum Beispiel zur Ausführung von Java- oder .NET-Code) unterstützt werden. Mit Anfrage ist eine (abgeschlossene) Interaktion mit der Anwendung gemeint.

Jede der Systemschichten kann instrumentiert und jede Systemschicht kann analysiert werden, je nach Ziel der Analyse. Es ist möglich eine Schicht zur Instrumentierung auszuwählen, die abstrakter (= in der Abbildung „weiter unten“) ist als die Schicht, die analysiert werden soll. Dieses Vorgehen hat den Vorteil, dass die darüber liegenden Schichten in der Regel nicht verändert werden müssen. Außerdem ist es in der Regel einfacher Informationen aus höheren Schichten in unteren Schichten zu integrieren als umgekehrt. Je größer jedoch der Abstand der Instrumentierungsschicht von der Analyseschicht ist, umso komplexer ist die Zuordnung von Beobachtungen in der Instrumentierungsschicht zu Vorgängen in der Analyseschicht.

Die Hardware als unterste Schicht eines Computersystems kann instrumentiert werden, um Betriebssysteme oder Virtualisierungssysteme (Hypervisor) zu analysieren. Ansätze zur Hardwareinstrumentierung sind in dieser Arbeit nicht untersucht worden.

Instrumentierung im Betriebssystem kann dazu verwendet werden Anwendungen und Vorgänge in Anwendungen (Anfragenbearbeitung) zu analysieren. Schon bei diesen benachbarten Schichten (Anwendung und Betriebssystem) ist die Zuordnung von Beobachtungen im Betriebssystem (= instrumentierte Schicht) zu Abläufen in der Anwendung (= analysierte Schicht) schwierig.

Aus Sicht eines Anwendungsentwicklers soll Instrumentierung dabei helfen die Anwendung zu verbessern, indem sich beispielsweise Fehler analysieren oder Leistungsengpässe identifizieren lassen. Welche Schichten sich dabei am besten zur Instrumentierung eignen ist demnach abhängig von der konkreten Analysefragestellung.

In der vorliegenden Arbeit wird das Betriebssystem als Instrumentierungsschicht untersucht. Aus der Sicht einer Anwendung stellt das Betriebssystem eine Abstraktion der Hardware bereit und verwaltet die vorhandenen Ressourcen verschiedener Art. Umgekehrt ist es also möglich durch Beobachtung von Vorgängen im Betriebssystem Informationen über das Verhalten von Anwendungen zu erfassen.

### 2.1.2 Sammlung von Analysedaten

Neben der Schicht zur Instrumentierung ist die Verfahrensweise der Systembeobachtung zu bestimmen. Grundsätzlich lassen sich zwei Ansätze zur Systembeobachtung unterscheiden: Sampling und Tracing.

**Definition (Sampling)** *Sampling beschreibt das periodische Abfragen von Sensoren/Instrumentierungspunkten im System. Dazu müssen abfragbare Sensoren im System vorhanden sein oder in das System eingebracht werden. Die Periode kann konfiguriert werden und so der Einfluss des Sampling auf die Gesamtsystem-Leistung bestimmt werden.*

**Definition (Tracing)** *Tracing beschreibt die Aufnahme von Ereignisketten im System. An verschiedenen Punkten im System sind Probes vorhanden, die Ereignisse anzeigen können. Diese Probes könnten schon zur Entwurfzeit des Systems eingerichtet oder zur Laufzeit in das System integriert werden. Eine Kette von auftretenden Ereignissen kann zur Systemanalyse verwendet werden; dazu werden alle generierten Ereignisse aufgezeichnet.*

In Tabelle 2.1 sind beide Ansätze vergleichend gegenübergestellt. Beide Ansätze zur Systembeobachtung haben spezifische Vor- und Nachteile und damit auch spezifische Einsatzgebiete.

Eine Kombination beider Ansätze ist das *Ereignis-basierte Sampling*. Dabei wird keine Zeitbezogene Abtastfrequenz verwendet - auftretende Ereignisse im System lösen das Abfragen von Werten aus. In der vorliegenden Arbeit wird Ereignis-basiertes Sampling nicht explizit untersucht: Betrachtet man die aufgenommenen Werte als Datenwerte des auftretenden Ereignis, so lässt sich Ereignis-basiertes Sampling auf Tracing (im hier definierten Sinn) abbilden.

Prinzipiell können beide Konzepte auf allen in Abbildung 2.1 dargestellten Systemebenen angewendet werden. Beispiele sind verschiedene *performance counter* auf Hardware- oder Betriebssystemebene, und Ereignisprovider in Anwendungen. Entsprechende Sensoren für Sampling und Ereignisprovider für Tracing sind in der Regel vom Hersteller bereits vorgesehen.

Im Vergleich der Vorraussetzungen und der Art der Anwendung von Sampling und Tracing gilt, dass die Sampling-Sensoren *passiv* abgetastet werden während die für Tracing verwendeten Ereignisprovider *aktiv* Ereignisse erzeugen, die gegebenenfalls aufgezeichnet werden.

Aus diesem grundsätzlichen Unterschied ergeben sich jetzt die spezifischen Vor- und Nachteile (Tabelle 2.1):

- Die Zahl der mit Sampling durchgeführten Messungen bestimmt der Analyst durch Regelung der Abtastrate - die Anzahl der bei Tracing auftretenden Ereignisse kann nicht bestimmt werden und ist abhängig von der analysierten Last im System.

| | **Sampling** | **Tracing** |
|---|---|---|
| Vorraussetzung | Sensoren | Ereignisprovider / Probes |
| Messung | Diskrete Werte von Sensoren, bei Bedarf | Aufnahme von Ereignisketten, nach Aktivierung von Tracing |
| Problemanalyse | Statistische Methoden, Vergleich mit Systemmodell | Suche nach besonderen Mustern im Ereignisstrom |
| Reaktion | Anpassung von Systemparametern, basierend auf Systemmodell | Systemanpassung, Vermeidung der erkannten Muster |
| Vorteile | Vorhersagbarer Einfluss auf die Systemlast (regelbar durch Sampling-Periode), Analyse-Reaktion Zyklus automatisierbar (Regelkreis) | Einzelereignisse (Ausfall, Angriff) können einfacher erkannt werden, Hinweise zu Problemlösung evtl. aus Trace ableitbar |
| Nachteile | Verpassen von Ereignissen aufgrund der Sampling-Periode, Ursachenbestimmung komplex oder unmöglich | Umfangreiche Datensammlung, große Logdateien, Analyse-Reaktion Zyklus schwer automatisierbar, Unvorhersagbare Anzahl von Ereignissen |
| Beispiele | Abfrage von *performance counter* (Hardware/Software), WMI | Linux Trace Toolkit, ptrace, DTrace, manuelle Instrumentierung |

**Tabelle 2.1:** Vergleich von Sampling und Tracing

- Werden Messwerte per Sampling aufgenommen, können aus dem zeitlichen Verlauf der Messwerte Rückschlüsse auf den Systemzustand gezogen werden. Der Verlauf kann durch statistischen Methoden mit einem Erwartungswert verglichen werden und notwendige Systemanpassungen abgeleitet werden.

- Durch die periodische Abfrage beim Sampling können unter Umständen wichtige Systemzustandsänderungen verpasst werden. Die Berücksichtigung des Abtasttheorems [94] würde bei vielen Metriken zu sehr kurzen Abtastperioden führen - mit wiederum entsprechend hoher Systembeeinflussung durch Sampling.

- Die Korrelation zwischen mehreren, unabhängig aufgenommenen Sampling-Werten ist aufwändig und die *gleichzeitige* Abtastung mehrerer Sensoren im Allgemeinen nicht realisierbar.

- Tracing erlaubt die Aufnahme (idealerweise) vollständiger Ketten von Ereignissen im System. Diese erlauben ebenfalls Rückschlüsse auf den Systemzustand und bieten bei der Analyse Ansatzpunkte zu eventuell aufgetretenem Fehlverhalten.

- Die Erkennung von besonderen (externen) Einzelereignissen, beispielsweise dem Ausfall einer Systemkomponente oder eines Angriffs auf das System, ist mit Tracing einfacher wenn entsprechende Ereignisse betrachtet werden.

Zusammenfassend lässt sich sagen, dass die beste Methode vom konkreten Einsatzgebiet und Analyseziel abhängt. In der vorliegenden Arbeit werden Tracing-basierte Methoden untersucht. Die gezeigten Nachteile beim Einsatz von Tracing schließen den Einsatz von Tracing-basierten Methoden in vielen Fällen aus. Die in dieser Arbeit vorgestellten Ansätze helfen die Tracing-spezifischen Nachteile abzuschwächen.

### 2.1.3 Auswertung und Analyse

Neben der Position im System und der Art der Messdatenaufnahme sind Art und Zeitpunkt der Datenanalyse ein weiteres Unterscheidungsmerkmal für Ansätze der Systemanalyse.

Bei der Analyse von Messdaten in einem System kann die *Online-* und die *Offline-*Analyse unterschieden werden.

**Definition (Online-Analyse)** *Unter Online-Analyse versteht man die Aufnahme und sofortige Analyse von Messwerten während der Ausführung des zu analysierenden Systems.*

**Definition (Offline-Analyse)** *Bei der Offline-Analyse sind Aufnahme und Analyse zeitlich getrennt: In einem ersten Schritt werden die Messwerte erfasst. Nach Abschluss der Messwerterfassung werden die Daten ausgewertet.*

Einen Sonderfall der Offline-Analyse stellt die *Post-Mortem*-Analyse dar. Bei einer Post-Mortem-Analyse muss das zu analysierende System beendet werden bevor mit der Analyse begonnen werden kann. Ein Grund kann beispielsweise die Art der Instrumentierung sein, die nur mit einem Systemneustart wieder aus dem System entfernt werden kann.

Bei beiden Analyse-Ansätzen werden die aufgenommenen Messwerte oder Ereignisketten analysiert, und es wird versucht eine Vorstellung von Aktivitäten im System zu erhalten.

Nach Abschluss der Systemanalyse und Identifikation eventuell vorhandener Probleme wird im nächsten Schritt das System angepasst. Betrachtet man die zeitliche Abfolge von Analyse und Anpassung, so kann die *synchrone* und die *asynchrone* Adaption unterschieden werden.

**Definition (Synchrone Adaption)** *Synchrone Adaption erfolgt sofort während der Analyse, nachdem erforderliche Systemanpassungsschritte identifiziert wurden.*

**Definition (Asynchrone Adaption)** *Bei der asynchronen Adaption wird erst die Analyse vollständig abgeschlossen und beendet. Anschließend erfolgt die Systemanpassung.*

Das in einem konkreten System verwendete Konzept der Systemanpassung hängt von verschiedenen Aspekten ab. Relevante Fragestellungen sind beispielsweise: Wie aufwändig ist die Datenaufnahme? Ist Online-Analyse überhaupt möglich? Wie aufwändig ist die Analyse? Welche Analysefragestellungen sind zu bearbeiten? Ist synchrone Anpassung des System notwendig?

### 2.1.4 Zusammenfassende Darstellung

Abbildung 2.2 zeigt alle bisher betrachteten Aspekte der Systembeobachtung im Überblick.

Die Kombination *Offline-Analyse/Synchrone-Adaption* kann bei Simulationen des Systems verwendet werden: eine Messwertreihe (Sampling) oder ein Ereignisstrom (Tracing) wird bei einem konkreten System aufgenommen und daraus ein Systemmodell für die Simulation erstellt.

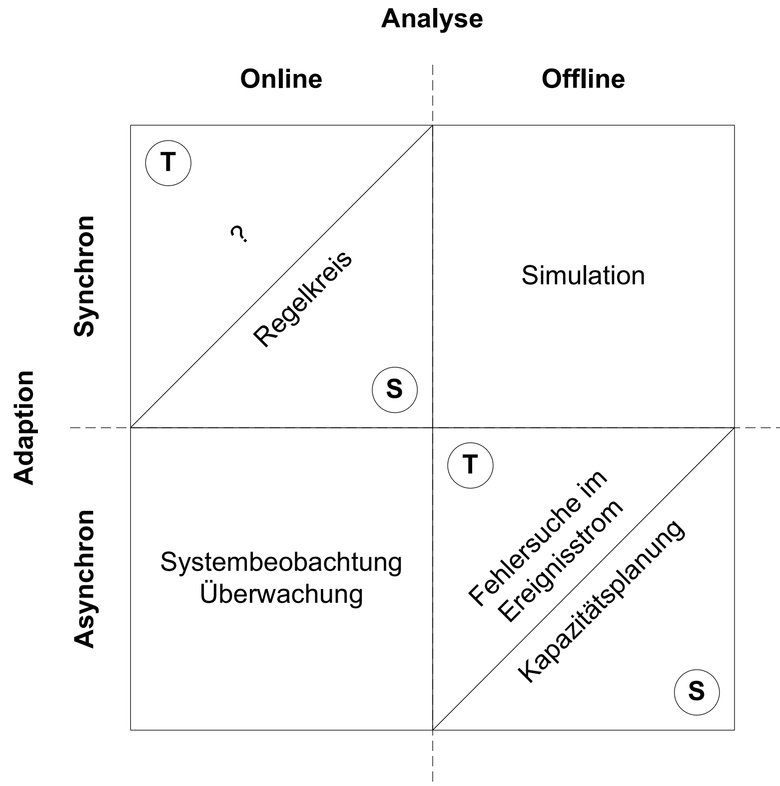



**Abbildung 2.2:** Taxonomie: Systembeobachtung

Während der Simulation werden jetzt basierend auf den simulierten Messwerten Anpassungen an dem Systemmodell vorgenommen. Das Simulationsergebnis kann jetzt zur Evaluierung der getesteten Anpassungen verwendet werden und diese können gegebenenfalls in das tatsächliche System integriert werden. Während der Simulation erfolgt also eine synchrone Anpassung - bezogen auf das tatsächliche System ist die Anpassung natürlich asynchron.

*Online-Analyse/Asynchrone-Adaption* ist gegeben, wenn ein System zur Laufzeit beobachtet und seine Leistung überwacht wird. Wenn dabei Probleme auftreten, werden entsprechende Anpassungen an dem System vorgenommen. Die Systembeobachtung kann sowohl mit Sampling-basierten als auch mit Tracing-basierten Ansätzen implementiert werden.

Die *Offline-Analyse/Asynchrone-Adaption* stellt den Normalfall für Tracing-basierte Analysen dar: Bei aktivem Tracing wird ein Ereignisstrom aufgezeichnet und nach dem Deaktivieren des Tracing wird beispielsweise eine erstellte Logdatei offline analysiert. Identifizierte Probleme werden asynchron behoben.

Die Offline-Analyse bei Sampling-basierter Systemanalyse kann für die Kapazitätsplanung verwendet werden. Mit einem gegebenen zeitlichen Verlauf einer bestimmten Metrik sind Durchschnittswerte und Lastspitzen identifizierbar.

Die Kombination *Online-Analyse/Synchrone-Adaption* kann bei Sampling-basierter Datenerfassung in Form eines Regelkreises erfolgen: Für bestimmte Metriken werden Idealwerte bestimmt, die ständig überwacht werden. Treten Abweichungen auf, so können mit Hilfe eines Systemmodells notwendige Anpassungen errechnet werden. Diese werden dann durchgeführt und beeinflussen die überwachten Werte.

Für Tracing-basierte Systembeobachtung ist die *Online-Analyse/Synchrone-Adaption* bisher selten untersucht worden. Dies liegt vor allem an der im Vergleich zu Sampling hohen System-

beeinflussung durch Tracing und der großen aufzunehmenden Datenmenge - eine Online-Analyse erscheint daher ineffizient.

In der vorliegenden Arbeit werden Konzepte zur Tracing-basierten, Online-Systembeobachtung mit synchroner Systemanpassung untersucht. Die zugrunde liegende Tracing-Infrastruktur wird in Kapitel 3 vorgestellt. Die Laufzeitumgebung und die Konzepte zur Online-Analyse und synchroner Systemanpassung werden in Kapitel 4 beschrieben.

## 2.2 Instrumentierung

Beide Ansätze zur Sammlung von Analysedaten in einem gegebenen System - Sampling und Tracing - erfordern Messpunkte im System, die die Aufnahme von Informationen und Messwerten ermöglichen. Diese Messpunkte werden durch Instrumentierung (siehe Definition 2) in das System integriert.

In diesem Abschnitt werden verschiedene Klassen und Konzepte zur Instrumentierung von Systemen vorgestellt.

### 2.2.1 Instrumentierungspunkte

Mit Hilfe von Instrumentierung werden Instrumentierungspunkte die die Analyse des Systemzustandes erlauben in ein System integriert.

**Definition (Instrumentierungspunkt)** *Ein Instrumentierungspunkt ist Programm- oder Binärcode in einem Softwaresystem, welcher entweder die Abfrage von Systemmetriken erlaubt (Sampling) oder ein auftretendes Ereignis anzeigt (Tracing).*

Wie im tatsächlichen System die Metriken abgefragt werden können oder auf welche Weise auftretende Ereignisse angezeigt werden, hängt von dem Gesamtkonzept der Instrumentierung ab. Möglich ist es beispielsweise eine zusätzliche Instrumentierungskomponente in das System zu integrieren oder als externe Komponente anzubinden, die eine Infrastruktur für die Datensammlung bereitstellt. Am eigentlichen Instrumentierungspunkt werden dann die Schnittstellen dieser Komponente verwendet.

Instrumentierungspunkte können auf Programmcode- oder auf Binärcode-Ebene definiert werden.

Auf Programmcode-Ebene können Instrumentierungspunkte in der verwendeten Programmiersprache definiert werden. Der Entwickler kann entsprechende Konzepte der Sprache oder eine Bibliothek verwenden, die Instrumentierungsfunktionen bereitstellt. Man kann *automatisch generierte* und *manuell eingefügte* Instrumentierung auf Programmcode-Ebene unterscheiden: Automatisch generierte Instrumentierungspunkte können beispielsweise durch entsprechende Compiler-Funktionen oder Pre-Prozessor-Makros während der Übersetzung erzeugt werden. Der GCC [6] kann beispielsweise (mit Hilfe der Compiler-Option `-finstrument-functions`) für Funktionseintritt und Funktionsaustritt automatisch Instrumentierungscode in ein Programm einfügen. Manuell erzeugte Instrumentierungspunkte bieten die Möglichkeit den vollen Umfang der Programmiersprache auszunutzen und genau zu bestimmen, welche Informationen an welchen Stellen im System aufgenommen und zur Verfügung gestellt werden.

Auf der Binärcode-Ebene definierte Instrumentierungspunkte trennen die Spezifikation des Ortes (Wo im System soll der Instrumentierungspunkt eingefügt werden?) von der Funktion

(Was soll am Instrumentierungspunkt tatsächlich ausgeführt werden?). Diese Trennung ist bei der Spezifikation von Instrumentierungspunkten basierend auf dem Programmcode in der Regel nicht der Fall. Die Angabe des Ortes bezieht sich auf eine Binärcode-Position im schon übersetzten System. Ein Instrumentierungspunkt wird an der entsprechenden Stelle eingefügt, indem der Programmcode geändert wird. An der entsprechenden Stelle im System wird dann ein Aufruf der Instrumentierungsfunktion integriert.

Bei der Implementierung von Instrumentierungspunkten können zwei Muster unterschieden werden: (1) Der Instrumentierungspunkt wird in das System integriert und die Daten werden erfasst, sobald der entsprechende Punkt ausgeführt wird. Dies stellt den Normalfall dar, der sowohl beim Tracing als auch beim Sampling des Systemzustandes verwendet werden kann. (2) Der Instrumentierungspunkt stellt eine Schnittstelle zu systeminternen Zuständen bereit. Erst wenn eine externe Komponente den Wert anfragt, wird der entsprechende interne Wert gelesen und zurückgegeben. Dieses Muster kann nur beim Sampling des Systemzustandes für Werte angewendet werden, die schon bei der regulären Ausführung des Systems erfasst werden.

Die Integration von Instrumentierungspunkten in zu analysierende Systeme kann in zwei Klassen unterteilt werden - statische Instrumentierung und dynamische Instrumentierung. Beide Klassen werden in den nachfolgenden Abschnitten genauer erläutert.

### 2.2.2 Statische Instrumentierung

**Definition (Statische Instrumentierung)** *Statische Instrumentierung ist das Einfügen von Instrumentierungspunkten in eine Anwendung, die sich nicht in Ausführung befindet. Im Anschluss an die Instrumentierung kann die instrumentierte Variante der Anwendung ausgeführt und analysiert werden.*

Statische Instrumentierung kann dabei auf dem Quellcode der Anwendung oder auf dem bereits übersetzten Binärcode beruhen.

In den Quellcode integrierte Instrumentierungspunkte sind einfach zu realisieren. Mit der Hilfe von Instrumentierungsbibliotheken (zum Beispiel MPICL [10] für MPI-Programme) oder Debug-Ausgaben (zum Beispiel durch `printf`-Ausgaben) sind Instrumentierungspunkte an beliebigen Stellen im System integrierbar und je nach Leistungsfähigkeit des gewählten Ansatzes zur Laufzeit des Systems aktivierbar und deaktivierbar.

Auf Binärcode basierende statische Implementierung erfordert die Angabe, an welchen Stellen im Code ein Instrumentierungspunkt eingefügt werden soll. Dies kann schon bei der Übersetzung des Quellcode geschehen sein, indem an potentiellen Instrumentierungspunkten besondere Muster von Instruktionen eingefügt wurden. Diese sind dann bei statischer Analyse des Binärcodes einfach zu finden. Ein anderer Weg ist die *Basic block*-Analyse, wenn beispielsweise an jedem Funktionseintritt (`call`-Instruktion im Binärcode) ein Instrumentierungspunkt eingefügt werden soll. Letztere Vorgehensweise ist auch möglich, wenn der Quellcode der Anwendung nicht verfügbar ist.

Binärcode-basierende Instrumentierung ist abhängig von der zugrunde liegenden Ziel-Plattform. Eigenschaften des Binärcode, beispielsweise ein feste Anzahl von Bits pro Instruktion, wirken sich direkt auf die Implementierungskomplexität der notwendigen Binärcode-Analyse aus.

Zusammenfassend lässt sich sagen, dass statische Instrumentierung auf Quellcode-Ebene einfach zu realisieren ist - in der Regel genügen wenige Codezeilen. Auf Binärcode-Ebene ist die

Instrumentierung komplexer - aber auch dann möglich, wenn der Quellcode nicht verfügbar ist.

Die Instrumentierungspunkte werden statisch in das System integriert. Die *Aktivierung* der Datenerfassung ist davon unabhängig und kann auch erst zur Laufzeit erfolgen, der Code für die Instrumentierung ist auf jeden Fall vorhanden.

### 2.2.3 Dynamische Instrumentierung

**Definition (Dynamische Instrumentierung)** *Dynamische Instrumentierung ist das Einfügen von Instrumentierungspunkten in eine Anwendung während ihrer Ausführung. Die Anwendung wird während der Ausführung geändert und kann anschließend analysiert werden.*

Während der Ausführung eines Systems ist in der Regel nur der Binärcode des Systems verfügbar und änderbar. Dynamische Instrumentierung basiert daher auf dem Binärcode. Ausnahmen sind auf dem Gebiet der dynamischen Sprachen zu finden, beispielsweise kann in *Smalltalk*-Laufzeitumgebungen der Programmcode in seiner Quellcoderepräsentation zur Ausführungszeit des Systems geändert werden.

Im Gegensatz zur statischen Instrumentierung bietet die dynamische Instrumentierung den Vorteil, dass sie erst dann Auswirkungen auf das ausgeführte System hat, wenn sie tatsächlich aktiviert wurde. Im Normalfall ist kein Instrumentierungscode im System vorhanden.

Die Aktivierung von Instrumentierungspunkten, das heißt die Änderung von Programmcode zur Laufzeit, ist komplex und stellt besondere Anforderungen an Sicherheit und Stabilität der Implementierung. Insbesondere bei der Berücksichtigung von Aspekten wie Nebenläufigkeit oder Exception-Handling [80] sind Besonderheiten zu beachten.

Zusammenfassend lässt sich feststellen, dass dynamische Instrumentierung auch in Umgebungen möglich ist, in denen das zu analysierende System nicht nach erfolgter Instrumentierung neugestartet werden kann - beispielsweise in produktiv eingesetzten Systemen. Die daraus entstehenden hohen Sicherheits- und Stabilitäts-Anforderungen führen zu Einschränkungen bezüglich der Art und des Ortes definierbarer Instrumentierungspunkte.

Im Allgemeinen gilt, dass bei der Verwendung dynamischer Instrumentierung inaktive Instrumentierungspunkte keinerlei Einfluss auf die Ausführung des Systems haben (*zero probe-effect*).

---

## 2.3 Automatisierte Überwachung

Die durch Instrumentierung gewonnen Messwerte oder Ereignisketten können dazu verwendet werden, den aktuellen Systemzustand zu erkennen und bei Bedarf zu beeinflussen. Die entsprechende Anpassung kann manuell, das heißt von einem Administrator ausgelöst, erfolgen. Alternativ kann die Regelung und damit die Überwachung des Systems automatisiert erfolgen. Der Administrator spezifiziert in einem solchen Fall das gewünschte Systemverhalten in einer angemessenen Form und eine Überwachungskomponente im System regelt die Systemparameter.

In diesem Abschnitt werden zwei Ansätze zur automatisierten Überwachung von Systemen vorgestellt.

### 2.3.1 Regelkreis-basierte Überwachung

Die Regelkreis-basierte Überwachung dynamischer Systeme wird in verschiedenen Anwendungsszenarien zur Beeinflussung physikalischer und virtueller Systeme eingesetzt. In diesem Abschnitt wird nur ein einfacher Überblick über das Gebiet gegeben. Dabei steht die Regelung des Verhaltens von Serversystemen im Vordergrund. Für weitergehende Fragestellungen sei auf entsprechende Literatur verwiesen, beispielsweise [24] oder [50].

Das Verhalten von Implementierungen von Serversystemen ist, da es sich um digitale Systeme handelt, nur Zeit- und Wert-diskret beschreibbar. Näherungsweise können Metriken über das Verhalten von Serversystemn - beispielsweise der Durchsatz oder die durchschnittliche Antwortzeit - als kontinuierlich betrachtet werden. Damit lassen sich die Erkenntnisse über die Regelung kontinuierlicher Größen physikalischer Systeme auch auf die Regelung von Serversystemen anwenden.

Grundlage einer Regelkreis-basierten Überwachung ist die Annahme eines rückgekoppelten Systems, welches von externen Störgrößen beeinflusst wird. Das System besteht aus der zu regelnden Komponente (Regelstrecke), dem Regler und der Rückkopplung. Der allgemeine Systemaufbau ist in Abbildung 2.3 dargestellt.

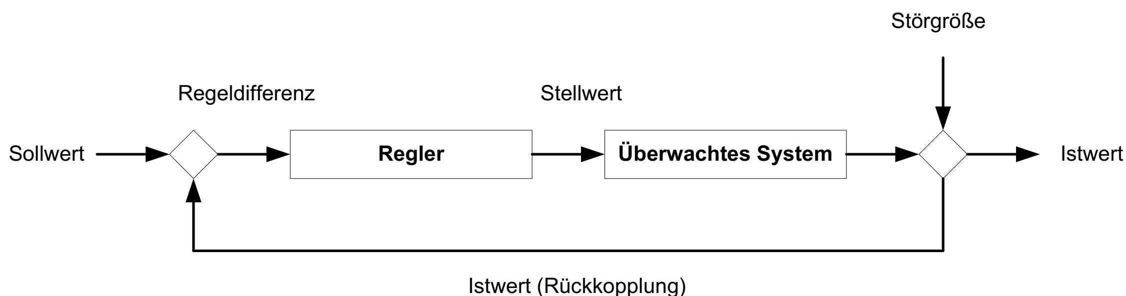



**Abbildung 2.3:** Schematische Darstellung eines Regelkreis

Das System wird bezüglich einer zu regelnden Metrik beobachtet. Der aktuell gemessene *Ist-Wert* wird mit einem *Soll-Wert* verglichen. Aus der *Regeldifferenz*, also der Abweichen von Soll- und Ist-Wert, wird vom Regler der *Stellwert* bestimmt. Mit Hilfe dieses Wertes wird das System rekonfiguriert und ein Zyklus im Regelkreis abgeschlossen. Eine *Störgröße* kann den Ist-Wert beeinflussen und muss vom Regler ausgeglichen werden. Die Beobachtung und Regelung des Systems erfolgt kontinuierlich und sorgt im Idealfall für eine genaue Einhaltung des gewünschten Soll-Wertes.

Das beschriebene abstrakte Konzept eines Regelkreises lässt sich auf Softwaresysteme übertragen: Eine Beobachtungseinrichtung im System nimmt den Ist-Wert bestimmter Systemeigenschaften auf. Eine Reglerkomponente vergleicht diesen Wert mit einem vorgegebenen Soll-Wert und berechnet bei Abweichungen einen Stellwert, der über entsprechende Schnittstellen neu im System eingestellt wird. Zur Berechnung des Stellwertes ist ein mathematisches Modell des Gesamtsystems erforderlich.

Nach [50] erfordert der Entwurf eines Regelungssystems sieben Schritte, die je nach Regelungsproblem bearbeitet werden müssen:

1. *Formulierung der Aufgabenstellung*

   Die Regelung eines Systems dient einer bestimmten Zielstellung; von dieser abhängend kann die konkrete Regelungsaufgabe abgeleitet werden

2. *Wahl geeigneter Stell- und Messeinrichtungen*

   Die zu überwachende Größe muss in geeigneter Weise messbar sein und dem Regler zur Verfügung gestellt werden. Ebenso muss festgelegt werden, auf welche Art der Soll-Wert spezifiziert und verarbeitet werden kann. Weiterhin müssen die Einstellungsmöglichkeiten des Reglers festgelegt werden.

3. *Bildung eines mathematischen Modells der Regelung*

   Das dynamische Verhalten des Gesamtsystems wird (falls möglich) mit einer Menge von Gleichungen beschrieben oder alternativ mit Hilfe experimenteller Modellbildung bestimmt. Das mathematische Modell bildet die Grundlage der Regelung und basiert auf den in Schritt 2 identifizierten Stell- und Messeinrichtungen.

4. *Systemanalyse*

   In diesem Schritt werden Eigenschaften des Regelkreises bestimmt, beispielsweise Stabilitäts-, stationäres, oder Konvergenzverhalten. Die errechneten Eigenschaften bilden die Grundlage für die Stabilisierung im nächsten Schritt.

5. *Stabilisierung/dynamische Korrektur*

   Aus den Eigenschaften der Modells und den allgemeinen Anforderungen an die zu entwickelnde Regelung werden die Gleichungen des Reglers abgeleitet. Zur Stabilisierung können verschiedene Verfahren verwendet werden.

6. *Simulation*

   Die Simulation kann auch schon parallel zu den vorhergehenden Schritten als Hilfsmittel verwendet werden. Die bestimmten Gleichungen des Reglers können im Simulator praktisch erprobt und gegebenenfalls weiter verfeinert werden.

7. *Realisierung der Regelung*

   Im letzten Schritt erfolgt die tatsächliche Erstellung des Regelungssystems. Dazu müssen Implementierungen der Regelungsgleichungen mit den erforderlichen Stell- und Messeinrichtungen des Systems verbunden werden.

Die Schritte 4 bis 6 sind unabhängig vom konkret zu regelnden System mit Hilfsmitteln der Mathematik und theoretischen Erkenntnissen der Regelungstechnik zu bearbeiten - Grundlage bildet das in Schritt 3 erstellte mathematische Modell.

Betrachtet man die vom konkreten System abhängigen Schritte aus der Perspektive der Überwachung von Serveranwendungen, so sind einige Besonderheiten identifizierbar:

Nicht jede Aufgabenstellung beim Betrieb einer Serveranwendung ist regelbar - die zu regelnden Eigenschaften müssen mess- und einstellbar sein. Dazu eignen sich demnach vor allem Metriken, die sich direkt durch Zahlenwerte ausdrücken lassen, beispielsweise Durchsatz oder die Anzahl der gleichzeitig bearbeiteten Anfragen. Eigenschaften wie Sicherheit und Zuverlässigkeit sind nicht direkt über einen Regelkreis steuerbar, da sie nur schwer auf direkt messbare oder einstellbare Metriken abgebildet werden können.

Die Aufstellung eines mathematischen Modells für Serveranwendungen ist ebenfalls komplex, da zahlreiche Randbedingungen das Verhalten unter Umständen stark beeinflussen. Die genaue Berücksichtigung von Caching und anderen Optimierungsheuristiken in einem Modell ist schwer.

Eine weitere Besonderheit der Regelung von Serveranwendungen ist, dass die Einrichtungen zur Messung von Ist-Werten und zur Einstellung von Konfigurationsparametern das Gesamtsystem unter Umständen stark beeinflussen. Eine Regelung muss solche Einflüsse als Störgröße berücksichtigen. Vor allem Sampling-basierte Ansätze zur Messung des aktuellen Systemzustands eignen sich zur Verwendung im Rahmen eines Regelkreises, da die Systembeeinflussung durch die Vorgabe einer Abtastrate konfigurierbar ist.

Im verbleibenden Teil dieses Abschnitts werden anhand verwandter Forschungsarbeiten konkrete Beispiele und Ansätze für die Regelkreis-basierte Überwachung von Serveranwendungen kurz dargestellt.

Ein wichtiger Aspekt beim Regelungsentwurf ist, wie bereits dargestellt, die Aufstellung eines Systemmodells als Grundlage für die Beobachtung und Anpassung des Systems. Das Modell kann einerseits basierend auf dem Wissen über die internen Vorgänge im zu regelnden System aufgestellt werden. Andererseits, wenn entweder kein detailliertes Wissen über Systeminterna vorhanden ist oder dieses nicht in einer brauchbaren Form darstellbar ist, kann ein Systemmodell auch zur Laufzeit aufgrund von Beobachtungen abgeleitet werden.

Ein Beispiel für die Erzeugung eines Modells unter Berücksichtigung der Implementierungsdetails ist in [97] für HTTP-Server dargestellt. Verwendete Parameter für die Modellkonfiguration sind beispielsweise die verfügbare Netzwerkbandbreite und die durchschnittliche Größe der angefragten Daten. Die Verwendung von Durchschnittswerten zeigt sofort, dass die Genauigkeit der Modellierung von der konkreten Last abhängt. Dennoch lassen sich basierend auf dem erstellten Modell verschiedene Charakteristiken ableiten, die in realen System beobachtbar sind - beispielsweise, dass ab einer bestimmten Ankunftsrate die Antwortzeit sehr schnell asymptotisch gegen Unendlich ansteigt. Für eine Regelkreis-basierte Überwachung ist ein solches Modell geeignet, wenn die (zahlreichen) Modellparameter mit Werten konfiguriert werden, die reale Lasten gut abbilden. Die Bestimmung solcher Parameter ist jedoch ebenfalls komplex.

Alternativ kann versucht werden, das Systemmodell erst zur Laufzeit zu bestimmen. Dieses Vorgehen weicht von der klassischen Regelkreistheorie ab, ermöglicht aber die tatsächlich vom System zu verarbeitende Last zu erfassen und auf nicht zur Entwurfszeit bekannte Umgebungsbedingungen zu reagieren. Zwei mögliche Verfahren sind *Trace-based Load Characterization* (TLC) [58] und *Software Architecture and Model Extraction* (SAME) [62]. Beide Ansätze beruhen auf der Idee, aus der Kommunikation zwischen Systemkomponenten Modelle basierend auf der Queueing-Theorie abzuleiten.

Bei TLC werden Ereignisketten für die Bearbeitung einzelner Anfragen im System aufgezeichnet. Dazu werden Zeitstempel, Typ und beteiligte Komponenten der ausgetauschten Nachrichten aufgezeichnet. Weiterhin werden verschiedene Interaktionsmuster unterschieden, beispielsweise einfacher Nachrichtenaustausch (*Remote Procedure Call*) oder asynchroner Nachrichtenaustausch. Aus diesen Informationen wird ein quantitatives Modell der Bearbeitung einer einzelnen Anfrage erstellt. Basierend auf den beteiligten Systemkomponenten können diese Modelle wiederum zu einem Gesamtmodell zusammengefasst werden. Dieses enthält dann Informationen über die Systemstruktur, also den beteiligten Subsystemen, und deren Zeitverhalten.

Ein Problem von TLC ist die effiziente Nachverfolgung von einzelnen Anfragen durch das Gesamtsystem. Der SAME Ansatz verzichtet daher auf diese Anforderung und beschränkt sich auf die Beobachtung von ausgetauschten Nachrichten im System. Aus diesen wird ein Interaktionsgraph aufgestellt, der ebenfalls die beteiligten Systemkomponenten und deren Zeitverhalten enthält. Auch SAME berücksichtigt verschiedene Nachrichtentypen und liefert am Ende ein Queueing-Modell des Gesamtsystems.

Bisher wurden Modelle betrachtet, die interne Informationen des zu regelnden Systems verwenden - seien es detaillierte Implementierungsdetails oder Nachrichten, die zwischen einzelnen Systemkomponenten ausgetauscht werden. Eine andere Sichtweise ist es, das zu überwachende System als *Black Box* zu betrachten und nur Vorgänge und Zeitverhalten an den externen Schnittstellen des Systems zu betrachten. Die Modellierung und Regelung muss dann auf eine andere Art erfolgen.

Für die Implementierung einer *Black Box* Sicht bietet sich das Proxy-Muster an: alle Anfragen an das Gesamtsystem durchlaufen eine vorgeschaltete Komponente, die Anfragen entgegen nimmt, eine Vorverarbeitung durchführt und diese anschließend an das tatsächliche implementierte System weiterleitet. In der Proxy-Implementierung sind Strategien zur Regelung des Gesamtsystemverhaltens umgesetzt.

In [29] und [47] werden entsprechende Proxy-Implementierungen vorgestellt. Die Grundidee ist bei beiden Ansätzen (1) ankommende Anfragen zu klassifizieren und verschiedenen Serviceklassen zuzuordnen, (2) Performance-Vorgaben der einzelnen Serviceklassen mit dem tatsächlichen Verhalten zu vergleichen und (3) nur so viele Anfragen der einzelnen Serviceklassen weiterzuleiten, dass die Vorgaben eingehalten werden können und das Gesamtsystem nicht überlastet wird.

Die Ansätze unterscheiden sich in der Bestimmung der Kapazität des Gesamtsystems und der Strategie der Anfrageweiterleitung:

In [47] werden für jede Serviceklasse Benchmarks durchgeführt, die das System schrittweise an ihre Belastungsgrenze führen. Darauf basierend können Schätzungen für den Aufwand einzelner Anfragen jeder Serviceklasse abgeleitet werden. Mit Hilfe dieser Schätzungen und der aktuellen Auslastung des Systems (geschätzt basierend auf der Menge der schon weitergeleiteten Anfragen) werden nur Anfragen weitergeleitet, die nicht zu einer Überschreitung der Gesamtkapazität führen und die die Performance-Vorgaben der einzelnen Serviceklassen nicht verletzen. Darüberhinaus kann die Schätzung der Last einer Anfrage dazu verwendet werden, eine *Shortest-Job-First*-Strategie zu implementieren. Dies führt zu geringerer (durchschnittlicher) Latenz des Gesamtsystems. Das dargestellte Verfahren stellt genau genommen keine Regelung, sondern eher eine Steuerung dar: die Rückkopplung fehlt und wird durch die Durchführung der Benchmarks ersetzt.

Genau in diesem Punkt unterscheidet sich der in [29] vorgestellte Ansatz: die Anzahl der gleichzeitig zulässigen Anfragen einer Serviceklasse wird schrittweise erhöht (ähnlich dem *Sliding Window* in TCP Implementierungen) bis eine Serviceklasse die Grenze ihrer Performance-Vorgaben erreicht hat. Die Anpassung der Anzahl der Anfragen jeder Serviceklasse stellt hier die Regelungsaufgabe dar - die Rückkopplung erfolgt über die zu beobachtenden Performance-Metriken. Vorteil dieses Ansatzes ist, dass keine Benchmark-Durchläufe notwendig sind und dadurch auch zur Entwurfszeit unbekannte Serviceklassen berücksichtigt werden können.

Nach der Betrachtung einzelner Aspekte des Aufbaus einer Regelung zur Überwachung von Serversystemen sollen jetzt abschließend zwei Beispiele für generische Frameworks zur Implementierung von Regelungen in Serveranwendungen vorgestellt werden.

In [70] wird das *Adaptive Server Framework* (ASF) beschrieben. Das ASF stellt Komponenten bereit, die es ermöglichen sollen, J2EE-Serveranwendungen mit adaptivem Verhalten zu ergänzen. Es werden drei Systemaspekte für die Regelung identifiziert: anwendungsspezifische Parameter, die Serverkonfiguration und Randbedingungen der vorhandenen Ressourcen. Für die Steuerung werden zwei Modelle aufgestellt: Ein Systemmodell beschreibt den Zusammenhang zwischen den vorhandenen Ressourcen und den Parametern der Serverkonfiguration.

Ein Steuerungsmodell regelt mit Hilfe der anwendungsspezifischen Parameter und der Ausgabe des Systemmodells das Systemverhalten. Nicht konkret vorgestellt ist das verwendete Verfahren zur Spezifikation von Performance-Anforderungen auf einer hohen Abstraktionsebene. Steuerungs- und Systemmodell werden in einer Fallstudie manuell hergeleitet. Die Modelle müssen in die bereitgestellten Komponenten integriert werden.

In [15, 111] wird mit *ControlWare* ein weiterer Ansatz für ein generisches Framework zur Regelung von Serversystemen vorgestellt. Schwerpunkt ist dabei die Garantie eines bestimmten Konvergenzverhaltens: bei Störung des Systems wird garantiert, dass die geregelte Performance-Größe (1) einen bestimmten Schwankungsbereich nicht verlässt und (2) innerhalb einer definierten Zeit den gewünschten Vorgabewert wieder erreicht. Weiterhin wird ein generischer Systemaufbau zur Implementierung der Regelung vorgestellt. Bestandteile dieser Architektur sind beispielsweise ein generischer Ressourcenmanager oder der *Softbus* zur Kommunikation zwischen Sensoren, Aktuatoren und weiteren Komponenten. Für das Systemmodell sind Anforderungen vorgegeben: die zu überwachende Größe muss messbar und kontrollierbar sein. Entsprechende Mess- und Einstellungskomponenten müssen implementiert werden.

Zusammenfassend lässt sich feststellen, dass die Überwachung von Anwendungen durch eine Regelung zahlreiche Einsatzgebiete hat und erfolgreich verwendet werden kann. Die Systembeobachtung erfolgt in der Regel Sampling-basiert. Das aufwändigere Tracing kann für die Aufstellung des Systemmodells verwendet werden.

### 2.3.2 Autonomic Computing

Die autonome Überwachung von Softwaresystemen wurde 2001 von IBM vorgeschlagen [59]. Die Vision ist dabei, die Verwaltung und den Betrieb komplexer Softwaresysteme zu automatisieren. Der Administrator spezifiziert Zielvorgaben auf hoher Ebene, die das System selbstständig zu erfüllen versucht. Dabei passt sich das System an veränderte Zielvorgaben und an veränderte Systemkonfigurationen an.

Aspekte, die dabei selbstständig verwaltet werden sollen, sind beispielsweise:

**Selbst-Konfiguration:** Verbindungen und Einstellungen einzelner Systemkomponenten sollen automatisch an veränderte Bedingungen angepasst werden.

**Selbst-Heilung:** Fehler im System sollen automatisch erkannt und behoben werden.

**Selbst-Optimierung:** Basierend auf den Anforderungen sollen die vorhandenen Ressourcen optimal den auszuführenden Diensten zugeteilt werden.

**Selbst-Schutz:** Angriffe auf das System sollen automatisch erkannt und (idealerweise schon im Vorfeld) verhindert werden.

Die Grundlage des Ansatzes von IBM bildet ein Netzwerk von *Autonomen Verwaltern*. Diese verfügen über Sensoren und Aktuatoren, ähnlich einer Regelkreis-basierten Steuerung. Jeder Autonome Verwalter führt (vollautomatisch oder von einem Administrator unterstützt) einen Zyklus aus Beobachtung, Analyse, Planung und Anpassung durch.

In der Beobachtungsphase wird der Systemzustand über die vorhandenen Sensoren erfasst. In der Analysephase werden die Messwerte verarbeitet und mit den Vorgaben verglichen. Anschließend werden in der Planungsphase notwendige Systemänderungen generiert, die anschließend, in der Anpassungsphase, tatsächlich auf das System angewandt werden.

Allen Komponenten im autonomen System liegt eine gemeinsame Wissensbasis zugrunde. Diese enthält das Wissen über den Systemaufbau, über die aktuelle Konfiguration, über vorgegebene Ziele und über geplante Änderungen. Weiterhin sind erfasste Metriken und durchgeführte Anpassungen (Logdatei) Bestandteil der Wissensbasis.

Der Ansatz des Autonomic Computing ist grundsätzlich eine Verallgemeinerung der Regelkreis-basierten Überwachung von Computersystemen: autonome Anpassung eines Systems kann mit Hilfe einer Regelung implementiert werden. Darüber hinaus sind jedoch auch andere Überwachungs- und Anpassungsmechanismen entwickelt worden, die nicht direkt auf Regelungstechnik beruhen und Ansätze, beispielsweise aus den Gebieten der künstlichen Intelligenz oder der Agentensysteme, nutzen.

Nachfolgend werden verschiedene über Regelkreis-basierende Techniken hinaus gehende Möglichkeiten zur Implementierung autonomer Systeme dargestellt. Als Grundlage dienen die in [63] und [85] dargestellten Forschungsgebiete und zentrale Fragestellungen des Autonomic Computing.

Die Realisierung der bereits beschriebenen Selbst-* Eigenschaften ist das oberste Ziel bei der Implementierung autonomer Systeme: Die Systeme sollen sich möglichst selbständig verwalten.

Zur Implementierung von *Selbst-Konfiguration* sind zwei Aspekte unterscheidbar: einerseits das Hinzufügen von Ressourcen zu einer schon vorhandenen Menge von Ressourcen und andererseits das Hinzufügen von Ressourcen-Verbrauchern, also beispielsweise Anwendungen, zu einem System. Der erste Fall ist üblicherweise mit verschiedenen Arten von Verzeichnisdiensten zu lösen. Eine neue Ressource meldet sich bei einer zentralen Registrierungsstelle an und gibt Art und Beschreibung der bereitgestellten Mittel bekannt. Der zweite Fall ist ungleich komplexer, da unter Umständen viele Abhängigkeiten zu berücksichtigen sind. Lösungsansätze beruhen meist auf einer formalisierten Beschreibung von Anforderungen und eines daraus generierten Installationsplanes.

Um *Selbst-Heilung* in einem System implementieren zu können, muss vor allem ein effizienter Weg zur Fehlerlokalisierung gefunden werden. Verschiedene Verfahren zur Zuordnung von Ereignissen im System zu problematischen Komponenten sind auch in anderen Zusammenhängen entwickelt worden. Weitere zu berücksichtigende Aspekte sind beispielsweise eine sich potentiell schnell ändernde Systemkonfiguration und Möglichkeiten zur Durchführung von Aktionen zur Heilung eines Systems.

In [51] wird ein Ansatz vorgestellt, der Methoden aus der Stochastik - maschinelles Lernen und Klassifizieren - mit Methoden für fein-granulares Recovery kombiniert. Die Annahme ist, dass aus der Beobachtung des regulären Systemverhaltens ein Modell gebildet werden kann, das anschließend zur Erkennung von Anomalien dient. Das Modell bildet dabei vor allem die strukturellen Zusammenhänge im System ab, also die vorhandenen Komponenten und das Zeitverhalten bei deren Kommunikation. Als Reaktion auf erkannte Anomalien sollen betroffene Komponenten neu gestartet werden, ein als *Micro-Reboot* bezeichneter Vorgang. Ein Administrator soll benachrichtigt werden, wenn die Anomalien auf diesen Weg nicht beseitigt werden können. Auf diese Art kann das System selbständig auf Fehlersituationen reagieren, die sich durch einen Neustart einzelner Komponenten beseitigen lassen.

In [96] wird eine Architektur für Komponenten vorgestellt, die in der Lage sind sich selbst zu heilen. Die Implementierung einer Komponente wird in eine Dienst- und eine Heilungsschicht unterteilt. Die Dienstschicht enthält die Implementierung der Kernfunktionalität der Komponente. Die Heilungsschicht enthält Überwachungs- und Heilungsfunktionen. Bestandteile der Dienstschicht sind mit speziellen Konnektoren verbunden, die Daten- und Kontrollfluss an

eine Überwachungskomponente in der Heilungsschicht weiterleiten. Die Überwachung basiert auf der Auswertung von Zustandsgraphen - entsprechende Abweichungen können so erkannt werden. Weiterhin überwachen die Konnektoren den Zustand der kommunizierenden Systembestandteile; ein Konnektor kann beispielsweise erkennen, dass eine Nachricht nicht ihren Empfänger erreicht hat. Die Heilungsschicht kann dann entsprechende Maßnahmen ergreifen und beispielsweise den Empfänger neustarten oder die Nachricht an einen anderen Bearbeiter weiterleiten.

Ein Ansatz zur Implementierung von Selbst-Heilung des Betriebssystems ist in [95] basierend auf Sun Solaris dargestellt. Auf der Betriebssystemebene sind Abhängigkeiten zwischen laufenden Prozessen und Threads, sowie den vom Betriebssystemkern jeweils zugewiesenen Ressourcen zu beachten - ganz allgemein muss sowohl auf Hard- als auch auf Softwarefehler reagiert werden können. Ein *Service Manager* im System soll es Diensten erlauben ihre Abhängigkeiten zu spezifizieren und einen *Restarter* anzugeben. Der Restarter wird gerufen wenn ein Problem mit einem Bestandteil des entsprechenden Dienstes erkannt wird. Ein Fazit des Autors ist, dass zur Realisierung von Selbst-Heilung entsprechende Software (1) mit Toleranz für Neustarts entwickelt werden und (2) ihre Abhängigkeiten in einer angemessenen Art spezifizieren muss.

Zur Realisierung von *Selbst-Optimierung* werden in vielen Fällen die bereits dargestellten Ansätze aus der Regelungstechnik verwendet: mit Hilfe eines Systemmodells wird versucht eine optimale Konfiguration des Systems abzuleiten, so dass Performance-Vorgaben und andere Randbedingungen eingehalten werden.

Für den *Selbst-Schutz* eines Systems sind Komponenten notwendig, die fortlaufend den Zustand einzelner Systembestandteile auf Konformität mit den definierten Anforderungen hin überprüfen. Die Anforderungen können beispielsweise durch Sicherheitsrichtlinien definiert werden. Nach der Identifizierung problematischer Komponenten, können diese beispielsweise automatisch aktualisiert werden oder, falls der Verdacht auf einen Angriff besteht, vom Rest des Systems isoliert oder auch neu-gestartet werden.

In [106] wird eine Reihe von Entwurfsmustern für die Implementierung selbst-verwaltender Systeme dargestellt, also Architekturvorschläge für häufig auftretende Fragestellungen: (1) *Resource Reallocation* entspricht einem klassischen Regelkreis. Sensoren erfassen den Ressourcenverbrauch und die zu regelnden Metriken; mit Hilfe eines Systemmodells werden notwendige Einstellungen abgeleitet und durch Aktuatoren in dem System eingestellt. (2) Mit dem *Corruption Resiliency* Muster können Zugriffe auf eine Ressource überwacht werden: Zustandsänderungen werden in einem Cache zwischengespeichert und auf ungewollte Änderungen überprüft. Diese Indirektion erlaubt zum Einen ein schnelles Erkennen von fehlerhaften Zuständen und zum Anderen eine Isolation von fehlerhaften Zugriffen. (3) Im *User Authorization* Muster werden Aktionen nicht sofort ausgeführt, sondern erst als geplant angezeigt und mit Hilfe eines Sicherheitsmodels überprüft. Anschließend werden zulässige Aktionen durchgeführt oder der Benutzer über Probleme benachrichtigt. (4) Ein *Model Comparator* kann dazu verwendet werden, von einer bestimmten Erwartungshaltung abweichendes Systemverhalten zu erkennen. Eingaben in das System werden mit Hilfe eines simulierten Systems in ein erwartetes Verhalten umgewandelt. Dies kann mit den tatsächlichen Vorgängen im System verglichen werden.

Die Kernprobleme des Autonomic Computing - wie kann der aktuelle Zustand erfasst werden und wie muss auf ungewünschte Abweichungen reagiert werden - werden von den dargestellten Mustern offen gelassen.

Einige weitere wichtige Fragestellungen im Bereich der autonomen Verwaltung von Systemen sind hier nicht dargestellt worden, da sie nicht den Kernbereich der Arbeit berühren - beispielsweise die Frage nach Konzepten zur Spezifikation der Ziele, die das System überwachen soll.

Zusammenfassend lässt sich zum Konzept des Autonomic Computing feststellen, dass es in vielen Fällen ähnliche Ansätze wie die Regelkreis-basierte Überwachung verwendet. An manchen Stellen sind jedoch auch stark abweichende Konzepte zu finden, beispielsweise im Bereich Selbst-Heilung. Ein zentrales Problem stellt dabei die Selbstbeobachtung des Systems dar, also die Identifikation von auftretenden Problemen und die Ableitung angemessener Reaktionen.

Die im Rahmen dieser Arbeit entwickelten Konzepte können beispielsweise zur Implementierung von Selbst-Beobachtung und Reaktion auf erkannte Vorgänge im System verwendet werden.

## 2.4 Begriffsbestimmung: Ereignis und Ereignisstrom

Im folgenden Abschnitt werden die Grundbegriffe der Arbeit definiert. Die Formalisierung dient der genauen Spezifikation des Systemmodells auf dem die neu entwickelten Konzepte beruhen. Wie beschrieben befasst sich die vorliegende Arbeit mit der online-Verarbeitung von Ereignisketten, die durch Tracing im System aufgenommen wurden. Der Schwerpunkt der nachfolgend dargestellten Formalisierung liegt auf der genauen Definition von Tracing-Konzepten - beispielsweise der Begriffe *Ereignis* und *Ereignisstrom*.

Ein Ereignis wird durch einen Zeitpunkt und eine Zustandsänderung definiert. Im Kontext eines Betriebssystemkerns sind Ereignisse beispielsweise die Erzeugung eines Threads, der Zugriff auf eine Datei oder die Anmeldung eines Benutzers.

Ereignisse können zueinander in Beziehung stehen. Dies kann beispielsweise durch eine *happened-before* Relation beschrieben werden, also einer Relation, die die Ereignisse nach ihrer Kausalität in eine Halbordnung einordnet. Die entstehende Ordnung ist keine Vollordnung, da bei Berücksichtigung von nebenläufigen Aktivitäten nicht-eindeutige Kausalitätsbeziehungen zwischen Ereignissen auftreten können.

Diese allgemeinen Definitionen sollen nun für den Kontext eines Betriebssystemkerns genauer spezifiziert werden.

### 2.4.1 Definition: Ereignis

Es können zwei unterschiedliche Konzepte des Begriffs „Ereignis“ unterschieden werden. Zum Einen ist es möglich ein Ereignis als etwas anzusehen, was zu einem bestimmten Zeitpunkt eine Zustandsänderung beschreibt (*Zeitpunkt-Semantik*): „Der Baum ist umgefallen.“ Zum Anderen kann ein Ereignis als ein Prozess in einem bestimmten Zeitraum angesehen werden (*Intervall-Semantik*): „Zwei Wochen Winterschlussverkauf.“

Im Bereich des Betriebssystemkerns ist die Annahme von Zeitpunkt-Semantik für auftretende Ereignisse sinnvoll. Bei der Instrumentierung eines Systems können in der Regel nur Zustandsänderungen zu einem bestimmten Zeitpunkt erkannt werden - die Aggregation von Ereignissen zu einem komplexeren Ereignis ist dann schon Aufgabe der Analyse: Die Ereignisse „Datei A wird geöffnet“, „Datensatz X wird in A geschrieben“ und „Datei A wird geschlossen“ könnten Bestandteil des zusammengesetzten Ereignis (mit Intervall-Semantik) „Datei A wird verwendet“ sein. Ereignisse mit Intervall-Semantik können also durch zwei oder mehr Ereignisse mit Zeitpunkt-Semantik modelliert werden - mit einem Ereignis, welches den Startzeitpunkt anzeigt, einem Ereignis, welches den Endzeitpunkt anzeigt und Ereignissen, die Vorgänge innerhalb dieses Zeitraums anzeigen.

Eine sinnvolle Ergänzung ist eine zusätzliche Beschreibung des Umfelds eines auftretenden Ereignis - des gerade aktiven Ausführungskontexts. Ein Ereignis tritt demnach zu einem bestimmten Zeitpunkt im Rahmen eines bestimmten Ausführungskontextes auf.

**Definition (Ereignis)** *Ein Ereignis $E$ wird durch einen Zeitpunkt $T$ (Zeitstempel), einen Ausführungskontext $C$ und einer Menge von Daten $D$ bestimmt, also $E = (T, C, D)$. Die Daten enthalten Informationen über die Zustandsänderung im System, die durch das Ereignis repräsentiert wird.*

Auf einzelne Elemente eines Ereignisses wird mit folgender Notation Bezug genommen: Sei $E$ ein Ereignis, so beschreibt $E.T$ den Zeitstempel, $E.C$ den Ausführungskontext und $E.D$ die Parameter.

Die Art der Erfassung und Representation der Informationen eines Ereignisses (Zeitstempel, Ausführungskontext und Daten) ist abhängig von der konkreten Implementierung des verwendeten Instrumentierungssystems.

Ein auftretendes Ereignis kann durch unterschiedliche Zeitpunkte beschrieben werden. Abbildung 2.4 zeigt eine schematische Darstellung.

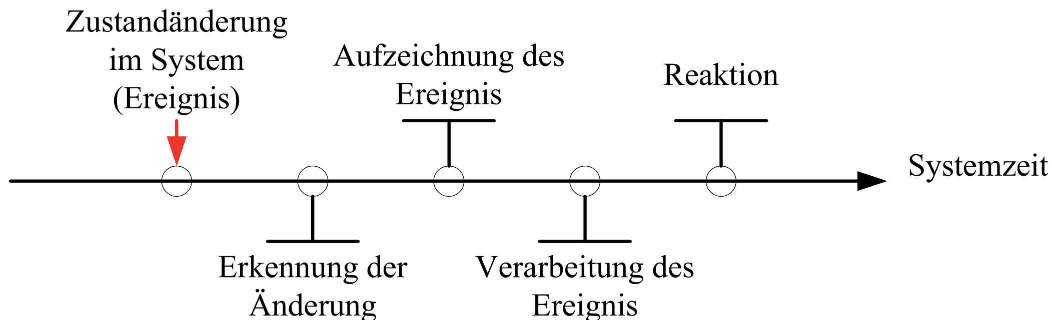



**Abbildung 2.4:** Zeitaspekte eines Ereignisses

Teilweise kann die Bedeutung der dargestellten Zeitpunkte nicht genau definiert werden. Der Zeitpunkt „Zustandsänderung im System“ ist in der Regel gleichbedeutend mit „Abarbeitung eines Instrumentierungspunktes, welcher das entsprechende Ereignis anzeigt“ - für ein Ereignis „Datei wird geöffnet“ hängt dann der Auftrittszeitpunkt von der Instrumentierungsstrategie ab. Welche dieser Zeitpunkte tatsächlich erfassbar sind, hängt ebenso von der konkreten Implementierung des Instrumentierungssystems ab.

Für die vorliegende Arbeit wird angenommen, dass die folgende Definition des Zeitstempels eines Ereignisses ausreichend ist.

**Definition (Zeitstempel)** *Der Zeitstempel $T$ eines Ereignisses beschreibt den Zeitpunkt der Erkennung der durch das Ereignis repräsentierten Zustandsänderung.*

Der Zeitpunkt $T$ kann mit Hilfe unterschiedlicher Zeitquellen bestimmt werden. Die Güte der Zeitquelle bestimmt die mögliche zeitliche Auflösung der Zeitstempel. Bei einer geringen Auflösung könnten nacheinander auftretende Ereignisse den gleichen Zeitstempel erhalten und damit scheinbar gleichzeitig auftreten.

Eine *ausreichend genaue* Zeitquelle garantiert, dass keine zwei Ereignisse den gleichen Zeitstempel erhalten ($\forall E_x, E_y$ mit $E_x \neq E_y$ gilt $E_x.T \neq E_y.T$).

Aktuelle CPUs verfügen in der Regel über ausreichend genaue Zeitquellen, die beispielsweise auf dem internen Prozessortakt beruhen und programmatisch ausgelesen werden können.

Auf der x86-Plattform [61] kann beispielsweise der *Time Stamp Counter* mit der Assembler-Instruktion `rdtsc` ausgelesen und als genaue Zeitquelle verwendet werden.

**Definition (Ausführungskontext)** *Der Ausführungskontext $C$, in dem ein Ereignis auftritt, beschreibt den Rahmen der ausgeführten Aktivität, die zu dem Ereignis geführt hat. Der Wertebereich von $C$ ist daher abhängig vom Abarbeitungsmodell des verwendeten Betriebssystems. In einem Ausführungskontext wird genau eine Aktivität ausgeführt.*

Der Ausführungskontext eines Ereignisses im Betriebssystemkern, wird durch die Art der Aktivität bestimmt, die zu dem Ereignis geführt hat. Je nach Architektur und Abarbeitungsmodell des verwendeten Betriebssystems können verschiedene Ausführungskontexte unterschieden werden.

Der Windows Betriebssystemkern verwendet ein Prozess/Thread Ausführungsmodell. Ein Ereignis kann daher beispielsweise im Kontext eines spezifischen Threads in einem bestimmten Prozess auftreten. Darüberhinaus existieren jedoch, außerhalb von Threads, weitere Möglichkeiten zur Ausführung von Aktivitäten und damit zur Erzeugung von Ereignissen: *Interrupt Service Routinen* (ISR), *Deferred Procedure Calls* (DPC) oder *Asynchronous Procedure Calls* (APC) [84].

### 2.4.2 Definition: Ereignisstrom

Die Menge der in einem System beobachtbaren Ereignisse wird zu einem Ereignisstrom zusammengefasst.

**Definition (Ereignisstrom)** *Ein Ereignisstrom $S$ fasst alle in einem System auftretenden Ereignisse $E$ sequentiell zusammen. Durch den Ereignisstrom werden die Ereignisse geordnet und ihnen eine eindeutige Index-Nummer $n$ zugewiesen: $S = \{E_1...E_n\}$. Die durch die Index-Nummer bestimmte Ordnung ist unabhängig von den Zeitstempeln und unabhängig von der Kausalitätsbeziehungen zwischen den einzelnen Ereignissen. Die Funktion $p_S(E)$ ordnet dem Ereignis $E$ seine Position im Ereignisstrom $S$ zu.*

Die Art und Weise, wie Ereignisse zu einem Ereignisstrom zusammengefasst werden, hängt von der Implementierung des Instrumentierungssystems ab, welches die Ereignisse erzeugt.

Beispielsweise können Ereignisse in unterschiedlichen Ausführungskontexten getrennt aufgezeichnet werden und anschließend zu einem Strom zusammengefasst werden. Die Zusammenfassung kann basierend auf den Zeitstempeln erfolgen. Bei Verwendung dieses Ansatzes muss für jeden Ausführungskontext eine Pufferverwaltung implementiert werden.

Alternativ können Ereignisse sofort (das heißt während ihres Auftretens) in einen Strom eingeordnet werden. Dabei wird nur ein zentraler Puffer für alle Ereignisse benötigt. Die Synchronisation der Pufferzugriffe wirkt sich jedoch auf die Konsistenz des Ereignisstroms aus. Die Folge der Zeitstempel ist unter Umständen nicht monoton steigend.

Allgemein lässt sich der Konsistenzbegriff eines Ereignisstroms wie folgt definieren:

**Definition (Konsistenz)** *Ein Ereignisstrom wird als konsistent bezeichnet, wenn alle Ereignisse im Ereignisstrom bezüglich ihrer Zeitstempel geordnet sind, also für alle Ereignispaare $E_x$, $E_y$ mit $p_S(E_x) < p_S(E_y)$ gilt, $E_x.T \leq E_y.T$.*

Neben der Art der Erzeugung des Ereignisstroms ist die Art der Aufzeichnung der Ereignisse relevant für die Konsistenz eines Ereignisstroms. Prinzipiell existieren zwei Möglichkeiten für die Aufzeichnung auftretender Ereignisse: (1) synchrone Aufzeichnung und (2) asynchrone Aufzeichnung. Unter Aufzeichnung eines Ereignisses ist in diesem Zusammenhang die Ausführung der Funktion $p_S(E)$ zu verstehen, das heißt die Zuweisung der Position im Ereignisstrom $S$ an das Ereignis $E$.

Bei einer synchronen Aufzeichnung wird nach dem Auftreten eines Ereignisses der Wert von $p_S(E)$ bestimmt und die Ausführung im Kontext $E.C$ solange unterbrochen. Bei einer asynchronen Aufzeichnung ist die Reihenfolge der Auswertung von $p_S(E)$ für die Ereignisse in einem bestimmten Kontext nicht bestimmt.

Durch die Verwendung asynchroner Ereignis-Aufzeichnung kann der Einfluss der Instrumentierung auf den Programmablauf minimiert werden. Bei synchroner Aufzeichnung wird die Ausführung blockiert, bis das Ereignis aufgezeichnet ist.

**Theorem** *Bei der synchronen Aufzeichnung von Ereignissen sind die Ereignisströme $S_C$, die alle Ereignisse eines Stroms $S$, die im Kontext $C$ auftreten, zusammenfassen, konsistent.*

*Dabei gilt $\forall E_x, E_y : p_S(E_x) < p_S(E_y) \Rightarrow p_{S_C}(E_x) < p_{S_C}(E_y)$ mit $E_x.C = E_y.C$.*

**Beweis** *Der Beweis ergibt sich direkt aus der Definition der synchronen Ereignisaufzeichnung: die Ausführung im Kontext $C$ wird erst fortgesetzt, wenn ein auftretendes Ereignis $E_i = (T_i, C, D_i)$ aufgezeichnet wurde. Gleichzeitig können keine weiteren Ereignisse im gleichen Ausführungskontext auftreten. Dem nächsten auftretenden Ereignis $E_{i+1} = (T_{i+1}, C, D_{i+1})$ wird daher ein späterer Zeitstempel zugewiesen, das heißt $T_i < T_{i+1}$. Da die Ausführung im Kontext $C$ erst fortgesetzt wird, wenn das Ereignis $E_i$ aufgezeichnet wurde, gilt: $p_S(E) < p_S(E)$. Daraus folgt $p_S(E_i) < p_S(E_{i+1}) \Leftrightarrow T_i < T_{i+1}$ und damit die Konsistenz der Ereignisströme im gleichen Ausführungskontext.*

Aus dieser Eigenschaft von Ereignisströmen eines Ausführungskontexts bei synchroner Ereignisaufzeichnung ergibt sich ein weiteres Merkmal: Die Ereignisse im gleichen Ausführungskontext sind durch den Ereignisstrom *happened-before* geordnet. Im Allgemeinen kann nicht angenommen werden, dass der Gesamtereignisstrom eines Systems konsistent ist.

## 2.5 Zusammenfassung

In diesem Kapitel wurden grundlegende Ansätze aus dem Arbeitsgebiet dargestellt und in einer Begriffsbestimmung die Konzepte „Ereignis“ und „Ereignisstrom“ im Betriebssystemkern definiert. Die Aufzeichnung von Ereignissen ist Gegenstand von Kapitel 3, die Verarbeitung von Ereignisströmen wird in Kapitel 4 beschrieben.

Ein Diskussion verwandter Forschungsarbeiten erfolgt in Kapitel 6.

Kapitel 3

# Der Windows Monitoring Kernel

*In diesem Kapitel wird der entwickelte* Windows Monitoring Kernel *(WMK) beschrieben. Der WMK bietet eine Infrastruktur zur Instrumentierung im Windows Kern, die eine effiziente Aufzeichnung von Ereignissen erlaubt. Der Windows Monitoring Kernel ist das erste Forschungsprojekt, welches außerhalb von Microsoft basierend auf dem Windows Research Kernel [82] entworfen und implementiert [90, 91] wurde.*

## 3.1 Zielstellung der Implementierung

Der *Windows Monitoring Kernel* (WMK) soll die Analyse von Softwaresystem unterstützen; dabei steht in erster Linie der Einsatz als Testumgebung oder „Prüfstand" im Vordergrund. Insbesondere war es kein Ziel, einen Betriebssystemkern für den Einsatz in produktiven, geschäftskritischen Anwendungen zu entwickeln.

Diese Einschränkung des Einsatzgebietes ist aus zwei Gründen wichtig: (1) Allein die Tatsache, dass eine neue Variante des Betriebssystemkern erstellt wird, schließt den Einsatz in Produktivsystemen aus, da dieser neue Kern auf den laufenden Systemen installiert werden müsste und nicht vom Hersteller unterstützt wird. (2) Die Einschränkung erlaubt es Konzepte auszuprobieren, die eventuell in zukünftigen Betriebssystemen in den Kern integriert werden *könnten*, um eine bessere Laufzeitumgebung für Anwendungen bereit zu stellen.

### 3.1.1 Anforderungen

Ganz allgemein soll der WMK zwei Aufgaben erfüllen. Einerseits soll er als eigenständiges Werkzeug zur flexiblen Aufzeichnung von Ereignissen im (Windows) Betriebssystemkern dienen. Andererseits soll er als Grundlage für die Implementierung der sofortigen Verarbeitung von Ereignisströmen verwendet werden können.

Aus diesen beiden Einsatzgebieten und grundsätzlichen Überlegungen zur Gestaltung einer Infrastruktur zur Aufzeichnung von Betriebssystemereignissen lassen sich die folgenden Anforderungen ableiten.

**Zuverlässige Ereignisaufzeichnung:** Die Ereignisse werden zur Analyse des Systemverhaltens verwendet, daher sollten möglichst keine auftretenden Ereignisse verloren gehen. Bei sehr hohem Ereignisaufkommen und überlaufenden Ereignispuffern soll ein Zähler der nicht gespeicherten Ereignisse erhöht werden. Auf diese Weise ist eine Interpretation der aufgezeichneten Daten unter Berücksichtigung der Anzahl der verlorenen Ereignisse möglich.

**Geringe Systembeeinflussung:** Die Aufzeichnung von Ereignissen erzeugt Last auf dem zu untersuchenden System. Die Last, die der WMK für die Ereignisverarbeitung erzeugt, soll möglichst gering gehalten werden. Bei sehr vielen aufzuzeichnenden Ereignissen kann eine starke Systembeeinflussung nicht verhindert werden.

**Aufzeichnung beliebiger Ereignisse:** Das Programmiermodell des WMK soll es prinzipiell ermöglichen, Ereignisse an beliebigen Stellen im Betriebssystemkern aufzuzeichnen, unabhängig von Einschränkungen wie beispielsweise des aktuellen Prozessor-Modus oder des Interrupt Request Levels.

**Nicht-blockierende Synchronisation:** Die Aufzeichnung von Ereignissen erfordert synchronisierte Puffer-Zugriffe. Die Synchronisation muss auf nicht-blockierende Art geschehen, da in einigen Ausführungskontexten (beispielsweise bei der Ausführung einer Interrupt Service Routine) blockierendes Warten nicht möglich ist.

**Ereignisse variabler Größe:** Der WMK soll keine Einschränkungen bezüglich der Größe der Ereignisse vorgeben. Das heißt, dass (1) Ereignisse unterschiedlicher Größe und (2) Ereignisse mit zur (Kernel-)Kompilierzeit unbekannter Größe aufgezeichnet werden können. Letzteres ist notwendig, um beispielsweise Dateinamen variabler Länge speichern zu können.

**Schnittstelle für externe Anwendungen:** Neben Ereignissen innerhalb des Kerns soll es ergänzend möglich sein, auch anwendungsspezifische Ereignisse mit Hilfe der WMK Infrastruktur aufzuzeichnen.

Diese Anforderungen werden mit der im nächsten Abschnitt beschriebenen Architektur umgesetzt.

### 3.1.2 Architektur

Die Abbildung 3.1 zeigt die Architekturkomponenten des WMK.

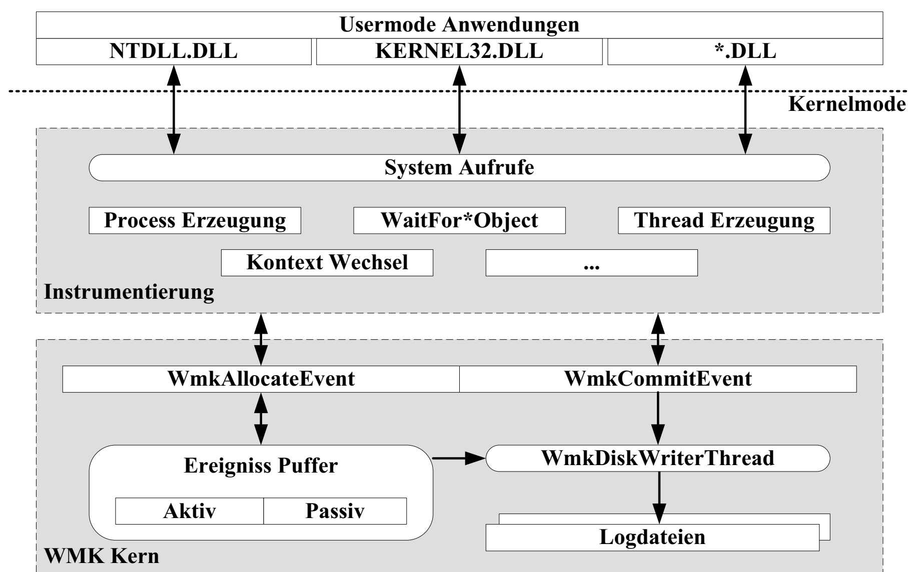



**Abbildung 3.1:** Architektur des Windows Monitoring Kernels

Anwendungen im Usermode verwenden die üblichen Systembibliotheken (zum Beispiel `ntdll.dll`) und können ohne Anpassungen auf einem WMK-System ausgeführt werden. Die Kommunikation mit dem Betriebssystemkernel erfolgt über Systemaufrufe.

Der Kernel enthält an verschiedenen Stellen Instrumentierungspunkte - Beispiele sind in der Abbildung dargestellt. Die Instrumentierungspunkte verwenden die WMK Schnittstellen zur Aufzeichnung von Ereignissen.

Der WMK Kern ist die zentrale Komponente des Windows Monitoring Kernel. Er ist in den Windows Betriebssystemkern integriert, damit er Ereignisse aufzeichnen kann, die von Kernel-Komponenten erzeugt werden. Der Kern wird von den Ereignispuffern, einem Systemthread zum Schreiben der Puffer, sowie der API zur Verwendung des WMK gebildet.

Die Implementierung der Instrumentierungspunkte wird im Abschnitt 3.2 beschrieben, der WMK Kern im Abschnitt 3.3.

## 3.2 Ereigniserzeugung

Die verschiedensten Vorgänge in einem Softwaresystem können als Ereignis interpretiert werden. Um eine Analyse von Ereignisketten zu ermöglichen, müssen Repräsentationen der Ereignisse protokolliert werden. Dies erfordert eine explizite Erzeugung und Aufzeichnung der Ereignisse, das heißt ein Vorgang (zum Beispiel das Öffnen einer Datei) muss zu einem beobachtbaren Ereignis führen.

Die explizite Erzeugung von Ereignissen kann in einer Anwendung bereits integriert und über entsprechende Schnittstellen zugänglich sein; beispielsweise Ausgaben auf einer Debug-Konsole. Eine andere Möglichkeit ist die (dynamische oder statische) Instrumentierung der Software, das heißt Code zur Ereigniserzeugung wird in ein bestehendes Programm eingefügt.

Der WMK verwendet den Ansatz der statischen Instrumentierung. Nachfolgend werden die im WMK verwendeten Konzepte zur Erzeugung von Ereignissen dargestellt.

### 3.2.1 Grundlagen

Der WMK setzt die in Abschnitt 2.4 definierten Konzepte um. Ein Ereignis wird demnach durch einen Zeitstempel, einen Ausführungskontext und Daten beschrieben. Darüber hinaus hat jedes Ereignis einen Typ, gehört also zu einer bestimmten Klasse von Ereignissen.

Die Informationen über ein bestimmtes Ereignis können unterteilt werden in Informationen, die für alle Ereignisse erfasst werden (Zeitstempel, Typ, Ausführungskontext) und in Informationen, die spezifisch für das konkrete Ereignis sind.

Im WMK sind die allgemeinen Informationen über ein Ereignis in einer *Header*-Datenstruktur zusammengefasst. Jedes Ereignis ist aus einem solchen Header und einem Speicherbereich für die ereignisspezifischen Daten zusammengesetzt.

**Listing 3.1:** WMK Datenstrukturen für Ereignisse

```
1 typedef struct _WMK_EVENT_HEADER {
2
3   // general information
4   ULARGE_INTEGER    TimeStamp;
5   UCHAR             Type;
6
```

```
7    // execution context
8    UCHAR              CpuId;
9    HANDLE             ProcessId;
10   HANDLE             ThreadId;
11
12   // event data
13   USHORT             BodySize;
14
15 } WMK_EVENT_HEADER, *PWMK_EVENT_HEADER;
16
17 typedef struct _WMK_E_PAGE_FAULT {
18
19   PVOID              VirtualAddress;
20   USHORT             Type;
21
22 } WMK_E_PAGE_FAULT, *PWMK_E_PAGE_FAULT;
```

In Listing 3.1 ist die WMK Header Datenstruktur und ein Beispielereignis (zur Repräsentation eines Seitenfehlers) dargestellt[1]. Das Feld `BodySize` im Ereignisheader gibt an, welche Größe die ereignisspezifischen Daten haben.

Das Feld `Type` im Ereignisheader definiert den Typ des Ereignisses. Für die Analyse muss der Typ eines Ereignis eindeutig bestimmbar sein. Der WMK unterscheidet Ereignisse im Betriebssystemkern und Ereignisse, die in einer anderen Systemkomponente auftreten. Das `Type`-Feld enthält eindeutige Nummern für verschiedene Betriebssystemereignisse. Eine spezielle Nummer ist für Ereignisse anderer Komponenten reserviert.

**Listing 3.2:** WMK Datenstruktur für Ereignisse außerhalb des Kerns

```
1 typedef struct _WMK_E_CUSTOM_EVENT {
2
3   WMKCUSTOMEVENTHANDLE CustomEventHandle;
4   USHORT CustomEventId;
5   USHORT DataLength;
6
7 } WMK_E_CUSTOM_EVENT, *PWMK_E_CUSTOM_EVENT;
```

Listing 3.2 zeigt die aus WMK Sicht gespeicherten Informationen über Ereignisse in externen Komponenten: Das Feld `CustomEventHandle` bestimmt die Komponente, die das Ereignis erzeugt hat. In `CustomEventId` wird das Ereignis genau bestimmt; der tatsächliche Wert ist von der ereigniserzeugenden Komponente abhängig. Die tatsächliche Größe der Ereignisdaten wird in `DataLength` gespeichert.

Der Datentyp `WMKCUSTOMEVENTHANDLE` hat einen 32 Bit Wertebereich. Entsprechend viele externe Komponenten können den WMK verwenden. Für die im Rahmen dieser Arbeit vorgenommenen Fallstudien war das verwendete Schema zur Bestimmung von Ereignisidentifikatoren keine Einschränkung: in vielen Fällen war die Aufzeichnung von Kernel-Ereignissen für die Analyse ausreichend - in den verbleibenden Fällen mussten höchstens drei externe Komponenten Ereignisse anzeigen.

Ereignisse der unterschiedlichen Typen werden an entsprechenden Instrumentierungspunkten im System erzeugt. Der WMK erlaubt die statische Instrumentierung des Windows Research

[1] Die Datenstrukturen für weitere Ereignisse im Betriebssystemkern sind in Anhang A dargestellt.

Kernels, das heißt Instrumentierungscode wird in den Quelltext des WRK eingefügt und dieser dann neu übersetzt.

In den nachfolgenden Abschnitten wird detailliert beschrieben, wie Instrumentierungspunkte für den WMK in den Betriebssystemkern, in Treiber und in Anwendungen (ausgeführt im Usermode) eingefügt werden können.

Bei allen drei Möglichkeiten muss bei jedem Instrumentierungspunkt geprüft werden, ob das Ereignis „aktiv“ ist, also tatsächlich aufgezeichnet werden soll, und wo die Daten des Ereignisses gespeichert werden sollen.

### 3.2.2 Betriebssystemkernereignisse

Instrumentierungspunkte werden im WMK durch eine Reihe von Makros definiert. Listing 3.3 enthält eine Übersicht.

**Listing 3.3:** WMK Makros zur Ereigniserzeugung

```
1  //
2  // allocate/commit macros
3  //
4
5  #define WmkAllocateEvent( EventType ) \
6    if (EventType##_FLAG & WmkEventSelector) { \
7      EventType* WmkEvent = _WmkAllocateEvent( sizeof( EventType ), ↩
           EventType##_ID, 0 ); \
8      if (WmkEvent != NULL) {
9
10 #define WmkAllocateEventEx( EventType, AdditionalBufferSize ) \
11   if (EventType##_FLAG & WmkEventSelector) { \
12     EventType* WmkEvent = _WmkAllocateEvent( sizeof( EventType ), ↩
           EventType##_ID, AdditionalBufferSize ); \
13     if (WmkEvent != NULL) {
14
15 #define WmkCommitEvent() \
16         _WmkCommitEvent(WmkEvent); \
17     } \
18   }
19
20 #define WmkBeginEvent( EventType ) \
21   if (EventType##_FLAG & WmkEventSelector) {
22
23 #define WmkEndEvent() \
24   }
```

Das Makro `WmkAllocateEvent(EventType)` fügt den dargestellten Programmcode am Instrumentierungspunkt ein. Zuerst wird geprüft, ob das spezifizierte Ereignis tatsächlich erfasst werden soll. Dazu wird der `WmkEventSelector` mit dem Ereignis-Flag verglichen. Soll das Ereignis erfasst werden, wird mit `_WmkAllocateEvent` Pufferspeicher reserviert. Wenn die Reservierung erfolgreich war (Rückgabewert ungleich `NULL`), wird Code ausgeführt, der nach dem Makro, direkt beim Instrumentierungspunkt definiert wird. In diesem Programmteil kann dann auf `WmkEvent` zugegriffen und die Ereignisdaten können gespeichert werden.

Sollen Daten mit variabler Länge gespeichert werden, wird WmkAllocateEventEx verwendet. Dieses Makro erhält einen weiteren Parameter AdditionalBufferSize, der die Größe des variablen Speicherbereichs angibt.

Beide WmkAllocate*-Makros erfordern ein WmkCommitEvent-Makro am Ende des Instrumentierungscodes. WmkCommitEvent schließt die Ereignisdatenerfassung ab und markiert den Speicherbereich im Puffer entsprechend als abgeschlossen.

Die zusätzlichen Makros WmkBeginEvent und WmkEndEvent dienen als Rahmen für Programmcode, der nur ausgeführt werden soll, wenn ein bestimmter Ereignistyp aufgezeichnet werden soll. Es wird daher nur ein Vergleich des gewünschten Typen mit dem WmkEventSelector eingefügt, ohne Speicher zu reservieren.

Mit Hilfe dieser Makros lassen sich Instrumentierungspunkte wie im Listing 3.4 dargestellt definieren.

**Listing 3.4:** Beispiel: Instrumentierung Dateizugriff

```
1  WmkAllocateEventEx(WMK_E_CREATE_FILE,
2                     ObjectAttributes->ObjectName->Length)
3
4    PCHAR pStringData = WmkGetDataSection(WmkEvent);
5
6    WmkEvent->Status = status;
7
8    WmkEvent->FileNameLength = ObjectAttributes->ObjectName->Length;
9    RtlCopyMemory(
10     pStringData,
11     ObjectAttributes->ObjectName->Buffer,
12     ObjectAttributes->ObjectName->Length );
13
14 WmkCommitEvent()
```

Die dargestellte Instrumentierung ist am Ende der Funktion IoCreateFile in der Datei base\ntos\io\iomgr\iosubs.c im Windows Research Kernel eingefügt - beim Öffnen von Dateien sollen entsprechende Ereignisse aufgezeichnet werden.

Zuerst wird ein Speicherbereich im Puffer reserviert. Da der Name der Datei eine Information variabler Länge ist, muss WmkAllocateEventEx verwendet werden. Als Parameter wird die tatsächliche Länge des Dateinamen angegeben (Zeile 2). Anschließend werden die Felder von WmkEvent mit Informationen beschrieben: Es werden der Status des Dateizugriffs (Zeile 6) und der Dateiname (Zeilen 8-12). Der Status kann bei der Analyse verwendet werden um beispielsweise festzustellen, ob die Datei erfolgreich geöffnet wurde.

Das Makro WmkGetDataSection (Zeile 4) liefert eine Referenz auf den Speicherbereich variabler Länge, der dann für den Dateinamen verwendet wird. Am Ende (Zeile 14) wird mit WmkCommitEvent die Aufzeichnung abgeschlossen.

Das dargestellte Vorgehen zur Definition von Instrumentierungspunkten kann in C-Quellcodedateien verwendet werden. Der WMK soll aber auch die Instrumentierung von Assembler-Funktionen ermöglichen. Zu diesem Zweck können C-Funktionen verwendet werden, die aus einer Assembler-Funktion gerufen werden. In Listing 3.5 ist ein entsprechender Aufruf dargestellt. Es sollen Systemaufrufe aufgezeichnet werden.

**Listing 3.5:** Beispiel: Instrumentierung Systemaufruf

```
...
; (eax) = Service number
; (edx) = Callers stack pointer
; (esi) = Current thread address

_KiSystemServiceRepeat:

  push eax
  push edx
  push esi
  stdCall _WmkLogSystemCallEntry,<eax,esi>
  pop esi
  pop edx
  pop eax

  mov edi, eax  ; copy system service number
...
```

Die Systemaufrufnummer (`eax`-Register) und die Information über den aktuellen Thread (`esi`-Register) werden an die C-Funktion `WmkLogSystemCallEntry` weitergegeben. In dieser Funktion wird dann ein entsprechendes Systemaufruf-Ereignis erzeugt und gespeichert.

Die Instrumentierung von Assembler-Funktionen erfordert genaues Wissen über den Speicherort von Informationen, das heißt darüber, welche Register an die Instrumentierungsfunktion übergeben und welche gesichert werden müssen. Entsprechende `push`- und `pop`-Instruktionen müssen vor und nach dem Aufruf der C-Funktion eingefügt werden.

Durch den Aufruf einer C-Funktion wird die Komplexität der Instrumentierung von Assembler-Funktionen verringert - die `stdCall`-Instruktion und die Stack-Operationen führen allerdings zu einem geringen Mehraufwand. Dieser Mehraufwand kann reduziert werden, wenn für die gezeigten Instrumentierungsmakros (siehe Listing 3.3) entsprechende Assembler-Versionen verwendet werden. Die notwendige Verwaltung der in der zu instrumentierenden Assembler-Funktion verwendeten Register führt jedoch zu komplexen Makros. Im Rahmen der vorliegenden Arbeit wurde auf eine entsprechende Implementierung verzichtet - die wenigen instrumentierten Assembler-Funktionen wurden manuell optimiert.

Die Erstellung von Instrumentierungspunkten ist mit den vorgestellten Techniken durch wenige Zeilen Programmcode möglich. Eine Auswahl von Kernel-Ereignissen, die mit dem WMK aufgezeichnet werden können, ist in Tabelle 3.1 und in Anhang A dargestellt.

Die Laufzeitumgebung des Windows Monitoring Kernels stellt sicher, dass die Aufzeichnung der Ereignisse unabhängig von weiteren Randbedingungen tatsächlich erfolgen kann. Das heißt insbesondere, dass Ereignisse unabhängig vom aktuellen Ausführungsmodus (*Dispatch-Level*) des Windows Kerns erfasst werden können. Die dafür notwendigen Implementierungstechniken sind im Abschnitt 3.3 dargestellt.

### 3.2.3 Anwendungsspezifische Ereignisse

Der WMK ist in erster Linie für die Aufzeichnung von Ereignissen im Betriebssystemkern geeignet. Für die Analyse von Anwendungen müssen Ereignisse im Kern zu Abläufen in der Anwendung in Beziehung gesetzt werden.

| Ereignis | Beschreibung |
|---|---|
| ProcessCreation | ein neuer Prozess wird erzeugt |
| ProcessTermination | ein Prozess wird beendet |
| ThreadCreation | ein neuer Thread wird erzeugt |
| ThreadTermination | ein Thread wird beendet |
| WaitEvent | ein Thread wartet auf ein oder mehrere Synchronisationsobjekte |
| WaitRelease | ein Thread gibt ein Synchronisationsobjekt frei |
| Syscall | ein Thread führt einen Systemaufruf durch |
| SyscallExit | ein Systemaufruf verlässt den Kernel-Mode |
| ContextSwitch | der Scheduler aktiviert einen neuen Thread |
| QuantumEnd | ein Thread hat sein Quantum verbraucht |
| CreateObject | eine Objekt-Referenz wird erzeugt |
| CreateFile | eine Datei-Referenz wird erzeugt |
| TimerExpiration | ein Kernel-Timer läuft ab |

**Tabelle 3.1:** WMK Ereignistypen (Auswahl)

Ohne Anpassung der zu analysierenden Anwendungen können Übergänge an der Schnittstelle zwischen Anwendung und Betriebssystemkern - also Systemaufrufe - betrachtet werden. Die Aktivität im Betriebssystemkern von Threads, in deren Kontext Systemaufrufe durchgeführt werden, können mit Hilfe der aufgezeichneten Kernelereignisse genauer analysiert werden. Aktivitäten im Usermode bleiben aus Betriebssystemsicht jedoch verborgen.

Soll die Anwendung als Ganzes betrachtet werden, sind Kernereignisse eventuell ausreichend: Kontextwechsel (beziehungsweise CPU-Nutzung) und Ein-/Ausgabe-Verhalten einer Anwendung können beispielsweise problemlos analysiert werden. Analysen einzelner Aspekte einer Anwendung sind durch kontrollierte Ausführung möglich: Durch gezielte und isolierte Ausführung der zu analysierenden Funktion eines Programms können die aufgezeichneten Ereignisse dann dieser Funktion zugeordnet werden.

Komplex wird die Zuordnung, wenn ein Feature durch mehrere kooperierende Threads realisiert wird. Teilweise kann man Zusammenhänge nachvollziehen, indem man die Synchronisationsobjekte des Kernels betrachtet: Durch Analyse von *wait*- und *signal*-Operationen auf Objekten kann der Kontrollfluss nachvollzogen werden.

Diese Ansätze haben jedoch auch ihre Grenzen, wenn indirekte Kontrollflussübergänge über gemeinsam genutzte Speicherbereiche oder Usermode-Threads koordiniert werden. In diesen Fällen ist die Zuordnung von Systemaufrufen zu konkreten Abläufen in der Anwendung sehr komplex und eventuell unmöglich.

In jedem Fall sind für eine genaue *Black Box*-Analyse einer Anwendung mit Hilfe von Kernelereignissen genaue Kenntnisse über die Interna der Anwendung notwendig. Die skizzierten Probleme sind auf einfache Weise zu umgehen, wenn der Quellcode der Anwendung verfügbar ist und auch im Usermode-Programm Instrumentierungscode eingefügt werden kann.

Der WMK stellt für Anwendungen eine Schnittstelle zur Verfügung, die es erlaubt auch anwendungsspezifische Ereignisse im WMK aufzuzeichnen. Die Korrelation zwischen dem aufgezeichneten Ereignisstrom und Abläufen in der Anwendung ist dann einfacher möglich. Die

Implementierung muss dabei eine eindeutige Identifizierung der anwendungsspezifischen Ereignisse ermöglichen. Weiterhin sollten die Konsistenz-Garantien der aufgezeichneten Ereignisse nicht verletzt werden.

Listing 3.6 zeigt die Schnittstelle, die von einer Usermode-DLL des WMK für Anwendungen zur Verfügung gestellt wird.

**Listing 3.6:** Interface zur Aufzeichnung anwendungsspezifischer Ereignisse

```
NTSTATUS
RegisterCustomEventprovider(
  LPSTR EventproviderName,
  USHORT* Handle
);

NTSTATUS
UnregisterCustomEventprovider(
  USHORT Handle
);

NTSTATUS
LogCustomEvent(
  USHORT Handle,
  USHORT EventId,
  USHORT DataLength,
  PVOID Data
);
```

Mit `RegisterCustomEventprovider` signalisiert eine Anwendung, dass sie Ereignisse an den WMK übermitteln möchte. Der Parameter `EventproviderName` beschreibt die Anwendung. Mit `Handle` wird ein Identifikator für den Ereignisprovider vom WMK zurückgegeben. Dieser Identifikator muss bei jedem aufzuzeichnenden Ereignis angegeben werden.

Ein Aufruf der Registrierung führt zu der Aufzeichnung eines `WMK_CUSTOM_EVENT` (siehe Listing 3.2) mit einer `CustomEventId` von 0 und dem Ereignisprovidernamen als Ereignisdaten, sowie dem zugewiesenen `Handle`.

Durch Aufrufen von `UnregisterCustomEventprovider` meldet sich eine Anwendung beim WMK ab und kann anschließend keine weiteren Ereignisse aufzeichnen.

Die eigentliche Aufzeichnung von Ereignissen erfolgt durch die Funktion `LogCustomEvent`. Als Parameter sind zu übergeben: der Identifikator des Ereignisprovider, eine ID des aufzuzeichnenden Ereignis, die Größe der Ereignisdaten und eine Referenz auf die Daten.

Die Ereignisdaten werden als `WMK_CUSTOM_EVENT` in die WMK Logdatei geschrieben. Der Ereignis-Header und Ausführungskontext des Ereignisses ergeben sich aus dem Thread, der `LogCustomEvent` aufgerufen hat.

Die `EventId` kann zur Unterscheidung verschiedener Ereignisse verwendet werden, die vom gleichen Ereignisprovider erzeugt wurden. Bei der Logdatei-Analyse sind jetzt alle Informationen vorhanden um die anwendungsspezifischen Ereignisse in den Kontext der im Betriebssystemkern aufgetretenen Ereignisse einordnen zu können.

### 3.2.4 Ereignisse in DLLs

Die mit Windows ausgelieferten dynamisch ladbaren Bibliotheken stellen Funktionen bereit, die von Usermode-Anwendungen verwendet werden können. Eine Instrumentierung dieser Funktionen kann in verschiedenen Anwendungen verwendet werden und ermöglicht eine Usermode-Instrumentierung *ohne* Änderung der eigentlichen Anwendung.

Im Rahmen der vorliegenden Arbeit wurden beispielsweise `wsock32.dll` und `ws2_32.dll`, also die Netzwerkschnittstelle für Windows Anwendungen, instrumentiert. Dies ermöglichte die Analyse, welche Threads welche Verbindungen zu welchen Rechnern aufbauen, oder welche Daten gesendet werden.

Prinzipiell wird die im letzten Abschnitt beschriebene Usermode-Schnittstelle des WMK verwendet: der DLL-Ereignisprovider wird registriert, wenn die DLL geladen und initialisiert wird.

Ereignisse werden geschrieben, wenn bestimmte Funktionen der instrumentierten DLL ausgeführt werden. Zu diesem Zweck wird für eine DLL ein Proxy generiert, der die gleichen Funktionen exportiert. Jede der Proxy-Funktionen ruft die ursprüngliche Implementierung auf und kann bei Bedarf (vorher oder hinterher) noch Ereignisse in die WMK Logdatei schreiben.

### 3.2.5 Ereignisse in Treibern

Treiber sind dynamisch zur Laufzeit ladbare Erweiterungen des Betriebssystemkerns. Sie werden im Kernelmodus ausgeführt und haben damit vollen Zugriff auf den Speicher und sonstige vom Betriebssystemkern verwalteten Ressourcen. Die häufigste Klasse von Treibern sind Gerätetreiber, die eine Schnittstelle zu Erweiterungshardware bereitstellen.

Die Programmierschnittstelle des WMK ist als Kernel API exportiert. Implementierungen von Treibern werden gegen die Kernelbibliothek gelinkt und können anschließend die WMK Funktionen verwenden. Die exportierten Funktionen sind analog zu denen in Listing 3.6 gestaltet.

Treiber folgen daher dem gleichen Schema bei der Aufzeichnung von Ereignissen: (1) Registrieren des Treibers als Ereignisprovider, (2) Aufzeichnen von Ereignissen, und (3) Abmelden des Providers. Die Schritte (1) und (3) erfolgen im Allgemeinen beim Laden beziehungsweise Entladen des Treibers.

## 3.3 Ereignisaufzeichnung

Der WMK Kern ist die zentrale Komponente des Windows Monitoring Kernel. Er ist in den Windows Betriebssystemkern integriert, damit er Ereignisse aufzeichnen kann, die von Kernel-Komponenten erzeugt werden. Der Kern wird von den Ereignispuffern, einem Systemthread zum Schreiben der Puffer, sowie der API zur Verwendung des WMK gebildet (siehe Abbildung 3.1).

Die einzelnen Komponenten werden in den folgenden Abschnitten beschrieben.

### 3.3.1 Pufferverwaltung

Die anfallenden Daten über auftretende Ereignisse werden in einem Puffer gespeichert. Enthält dieser Puffer eine bestimmte Anzahl von Ereignissen, wird der Pufferinhalt in einer Logdatei gesichert.

Der Ereignispuffer ist in zwei unabhängige Teile unterteilt, welche entweder eine *aktive* oder eine *passive* Rolle einnehmen. Neue Ereignisse werden in den aktiven Puffer geschrieben. Ist in diesem kein Speicherplatz mehr vorhanden, wird der passive Puffer aktiviert: der aktive Puffer wird zum passiven Puffer und umgekehrt. Jetzt kann der (volle) passive Puffer auf die Festplatte geschrieben werden.

Die WMK-Puffer werden zu einem sehr frühen Zeitpunkt im Windows Bootprozess angelegt. Auf diese Weise ist es möglich, Ereignisse schon während des Bootvorgangs aufzuzeichnen.

Die zur Aufzeichnung der Ereignisse verwendeten Puffer werden in dem Teil des Speichers angelegt, der nicht vom Betriebssystem ausgelagert wird (= *non-paged pool*). Dies ist notwendig, um zu jeder Zeit und für jeden möglichen Ausführungskontext garantieren zu können, dass der Pufferspeicher verfügbar ist und keine Seitenzugriffsfehler verursacht. Die Bearbeitung von Seitenzugriffsfehlern ist in manchen Ausführungskontexten nicht möglich und führt zu einem Systemfehler.

Zu beachten ist weiterhin, dass die Größe des non-paged Pool begrenzt ist. Der Windows Kern erfasst beim Start des Systems die Größe des verfügbaren Speichers und reserviert einen bestimmten Anteil. Ist der entsprechende Speicherbereich verbraucht und wird weiterer Speicher aus diesem Pool angefordert, führt dies ebenfalls zu einem Systemfehler. Es ist daher vorteilhaft, die Größe der Puffer zur Ereignisaufzeichnung möglichst klein zu halten. Da auch andere Anwendungen Speicher aus dem nicht-auslagerbaren Speicher anfordern, wird durch einen kleinen WMK-Puffer auch der Einfluss auf solche Anwendungen minimiert.

Die Puffer-Größe muss so gewählt werden, dass durch die beschriebene Doppel-Pufferung keine Ereignisse verloren gehen. Es muss also möglich sein, den passiven Puffer „schnell genug“ auf die Festplatte zu schreiben, solange noch Speicherplatz im aktiven Puffer verfügbar ist. Ob eine solche Größe tatsächlich bestimmt werden kann, hängt direkt von der Ereignisrate ab, also davon, wie viele Ereignisse pro Zeiteinheit im System auftreten. Da diese Rate im allgemeinen nicht vorher bekannt ist, kann die erforderliche Puffergröße nur abgeschätzt werden.

Die gewählte Puffergröße beeinflusst weiterhin die Häufigkeit, mit der die aufgezeichneten Ereignisse in die Logdateien geschrieben werden: Ist der Puffer klein, muss häufiger zwischen aktivem und passivem Puffer umgeschaltet werden und dementsprechend häufiger müssen Daten auf die Festplatte geschrieben werden. Ist der Puffer größer, muss seltener geschrieben werden.

Die optimale Puffergröße kann also nur experimentell bestimmt werden, da sie einerseits von der zu erwartenden Ereignisrate abhängt und andererseits von der Geschwindigkeit der Speichermedien für Logdateien bestimmt wird. Weiterhin ist die Puffergröße vom non-paged Pool Speicherbedarf des zu untersuchenden Systems begrenzt.

Der WMK soll die Ereignisaufzeichnung in beliebigen Ausführungskontexten erlauben, daher sind für die Synchronisation der Pufferzugriffe Konzepte wie Semaphoren oder Mutex nicht anwendbar - in manchen Ausführungskontexten (zum Beispiel im Kontext von *Deferred Procedure Calls*) kann nicht blockierend auf Synchronisationsobjekte gewartet werden. Daher sind im WMK nicht-blockierende Synchronisationskonzepte verwendet worden, die im nächsten Abschnitt beschrieben werden.

### 3.3.2 Synchronisation

Aus dem beschriebenen Konzept der Pufferung von Ereignisdaten ergibt sich, dass die folgenden Aktivitäten synchronisiert werden müssen:

**Anforderung von Speicherplatz im Ereignispuffer:** Parallele Aktivitäten, die gleichzeitig Speicherplatz im Ereignispuffer anfordern, müssen synchronisiert werden, so dass niemals Speicherbereiche für die Aufzeichnung von mehr als einem Ereignis verwendet werden.

**Umschalten von aktivem und passivem Puffer:** Kann eine Speicheranforderung nicht mehr aus dem aktiven Puffer erfüllt werden, müssen die Puffer umgeschaltet werden. Dies muss synchronisiert erfolgen, so dass die Umschaltung nur einmal durchgeführt wird und während der Umschaltung keine Ereignisse verloren werden.

**Aufforderung zum Schreiben von vollen Puffern:** Gefüllte Puffer müssen in eine Logdatei geschrieben werden; dazu muss ein Systemthread aktiviert werden, der diese Aufgabe übernimmt. Durch Synchronisierung der Aktivierung muss sichergestellt werden, dass (1) nur volle Puffer und (2) jeder Puffer nur genau einmal geschrieben wird.

Zur Unterstützung beliebiger Ausführungskontexte für Ereignisse bietet sich die Verwendung nicht-blockierender Synchronisationskonzepte an. Im Windows Kern sind verschiedene Funktionen (`Interlocked*`) verfügbar, die, basierend auf der CPU Instruktion `cmpxchg` [61] für x86-CPUs, atomar eine Vergleichs- und eine Zuweisungsoperation durchführen.

Um die `cmpxchg`-basierten Funktionen für die Pufferverwaltung des WMK verwenden zu können, muss die Zugriffssynchronisation auf die Änderung *eines einzelnen Wertes* abgebildet werden - die Verwendung zur Synchronisation erfolgt dann nach dem folgenden Muster: (1) ursprünglichen Wert lesen, (2) neu zu setzenden Wert bestimmen (abhängig vom ursprünglichen Wert), (3) versuchen, den neuen Wert zu setzen, (4) falls nicht erfolgreich, wieder bei (1) beginnen.

Dieses Verfahren verwendet also *polling*, das heißt es wird solange versucht die Wertänderung durchzuführen, bis dies erfolgreich ist. Die entsprechende Schleife lastet die CPU (theoretisch) voll aus. Auf Ein-Prozessor-Systemen ist dies unproblematisch, da jeweils nur ein Thread aktiv ist, der dann erfolgreich die `cmpxchg`-Instruktion ausführen kann. Die Schleife wird in der Regel nur einmal durchlaufen. Auf Mehr-Prozessor-Systemen können parallele Aktivitäten gleichzeitig die `cmpxchg`-Instruktionen aufrufen. Kollidiert die Ausführung von `cmpxchg` auf unterschiedlichen Kernen, so wird genau ein Thread die Instruktion erfolgreich ausführen und anschließend andere Aktionen durchführen. Das heißt insbesondere, dass er keine erneute `cmpxchg`-Instruktion auf der gleichen Speicheradresse ausführen wird.

Es ist plausibel anzunehmen, dass der *worst-case* für die Synchronisierung mit Hilfe von `cmpxchg` in der Praxis nicht konstruiert werden kann: (1) In der WMK-Implementierung sind nach der Ausführung von `cmpxchg` zahlreiche Instruktionen notwendig, um eine weitere `cmpxchg`-Instruktion bezogen auf die gleiche Speicheradresse zu erreichen. (2) Wird ein Thread direkt nach der Ausführung von `cmpxchg` unterbrochen, so sind für den Wechsel zum nächsten Thread (der im ungünstigsten Fall direkt vor einer entsprechenden `cmpxchg`-Instruktion unterbrochen wurde) ebenfalls zahlreiche Instruktionen auszuführen. In beiden Fällen wird daher ein wartender Thread auf einer anderen CPU die `cmpxchg`-Instruktion erfolgreich ausführen können.

Sind so viele CPUs und eine entsprechende Last vorhanden, dass die kürzeste Instruktionssequenz zwischen zwei `cmpxchg`-Instruktionen permanent ausgeführt wird, so ist theoretisch das Verhungern (*starvation*) eines Threads möglich. In der Praxis trat dieses Problem beim Einsatz des WMKs auf Mehr-Prozessor-Systemen nicht auf.

Die tatsächliche Synchronisierung der WMK Pufferzugriffe geschieht folgendermaßen: Der WMK Puffer besteht, wie bereits beschrieben, aus zwei Teilen - einem aktiven und einem passiven. Beide Pufferteile lassen sich durch eine Basisspeicheradresse und die Puffergröße beschreiben. Zusätzlich wird in der globalen WMK-Verwaltungsdatenstruktur ein Pointer für die aktuelle Schreibposition innerhalb des gerade aktiven Puffers gespeichert. Die Synchronisation nebenläufiger Zugriffe basiert auf Modifikationen dieses Pointers.

Bei einem Instrumentierungspunkt wird `WmkAllocateEvent` verwendet, um den notwendigen Speicherplatz für Ereignisdaten anzufordern. Bekannt sind an dieser Stelle die aktuelle Schreibposition und die Größe der Ereignisdaten. Aus diesen Informationen kann errechnet werden, um wie viele Bytes der Schreibpointer erhöht werden muss. Im Normalfall, das heißt wenn keine Kollision dieser Operation auftritt und ausreichend Platz im Puffer vorhanden ist, wird der Schreibpointer auf seine neue Position gesetzt (mit Hilfe von `cmpxchg`) und die ursprüngliche Position zurückgegeben. An dieser Stelle können jetzt im Instrumentierungspunkt die Ereignisdaten gespeichert werden.

Tritt eine Kollision auf, das heißt konnte der Schreibpointer nicht auf die neue Position gesetzt werden, wird die jetzt neue Position des Pointers gelesen und basierend darauf erneut versucht Speicherplatz zu reservieren.

Ein Sonderfall tritt auf, wenn der aktuelle Puffer voll ist. Ist auch der passive Puffer voll, also noch nicht in die Logdatei geschrieben, so kann das Ereignis nicht gespeichert werden und geht verloren - der entsprechende Zähler für verlorene Ereignisse wird erhöht. Ist der passive Puffer verfügbar, müssen die Puffer umgeschaltet werden. Dazu wird versucht den aktuellen Schreibpointer auf den Anfang des passiven Puffers zu setzen. Da auch diese Operation mit Hilfe von `cmpxchg` durchgeführt wird, setzt genau ein anderer Thread die neue Schreibposition. Parallel ausgeführte Threads lesen entweder schon die neue Schreibposition und versuchen zur Speicherplatzanforderung diese umzusetzen oder warten auf die erfolgte Umschaltung und fordern anschließend den benötigten Speicherplatz erneut an.

Auf die beschriebene Art und Weise werden ohne blockierende Synchronisationskonstrukte die Speicheranforderungen im WMK Puffer verwaltet und zwischen aktivem und passivem Puffer umgeschaltet. Eine weitere Aktivität, die Synchronisierung erfordert, ist das Schreiben gefüllter Puffer, beziehungsweise die Aktivierung des entsprechenden Systemthreads.

Zu diesem Zweck wird Referenzzählung verwendet: für jeden Puffer wird ein Zähler für erfolgreiche Speicherplatzanforderungen verwaltet. Dieser Zähler wird bei `WmkAllocateEvent` erhöht und bei `WmkCommitEvent` verringert. Die entsprechende Wertänderung wird ebenfalls mit `cmpxchg`, beziehungsweise mit den Funktionen `InterlockedIncrement` und `InterlockedDecrement` des Windows Kerns, durchgeführt.

Nach der Änderung des Referenzzählers wird der neu gesetzte Wert ausgewertet. Durch die atomare Ausführung der Operation wird genau ein Thread den Referenzzähler auf 0 verringern. Dieser Thread prüft jetzt den Status des Puffers, auf den sich der Zähler bezieht: Ist der Puffer im Moment passiv, wurde vom aktuellen Thread das letzte offene Ereignis in diesem Puffer abgeschlossen und der Systemthread zum Schreiben des Puffers kann aktiviert werden. Ist der Puffer im Moment aktiv, wird keine Aktion ausgelöst.

Die Prüfung, ob ein Puffer aktiv oder passiv ist, muss mit der Puffer-Umschaltung in `WmkAllocateEvent` synchronisiert werden. Dazu wird ein Flag verwendet, welches anzeigt, dass der Systemthread bereits signalisiert wurde. Dieses Flag wird ebenfalls mit `cmpxchg` manipuliert. Wird beim Abschließen eines Ereignis der Referenzzähler auf 0 reduziert, wird das entsprechende Flag gesetzt und (falls erfolgreich) der Thread aktiviert. Wird nach dem Umschalten der Puffer festgestellt, dass der Referenzzähler für den passiven Puffer 0 beträgt, wird ebenfalls versucht, erst das Flag zu setzen und (falls erfolgreich) der Thread aktiviert.

Diese Struktur stellt sicher, dass der Thread (1) tatsächlich aktiviert wird, wenn ein voller Puffer vorhanden ist und (2) die Aktivierung nur genau einmal erfolgt.

### 3.3.3 Logdateiverwaltung

Bestandteil des WMK-Kerns ist ein Systemthread, der gefüllte Puffer in Logdateien auf die Festplatte schreibt. Der entsprechende Thread wird beim Systemstart gestartet und initialisiert.

Der Thread wird gestartet, sobald das Windows Ein-/Ausgabesystem initialisiert wurde (Funktion `IoInitSystem` in `base\ntos\io\iomgr\ioinit.c`). Ab diesem Zeitpunkt ist es prinzipiell möglich Daten auf die Festplatte zu schreiben. Die WMK Puffer selbst sind zu diesem Zeitpunkt bereits initialisiert und bereit zur Aufzeichnung von Ereignissen. Sind beide Puffer gefüllt bevor der Systemthread initialisiert ist und treten weitere Ereignisse auf, so gehen diese verloren.

Der Thread initialisiert die Logdatei und wartet anschließend auf ein `FlushEvent` welches signalisiert, dass ein Ereignispuffer geschrieben werden kann. Das `FlushEvent` wird in den Funktionen `WmkAllocateEvent` oder `WmkCommitEvent` gesetzt, wenn (1) alle Ereignisse im passiven Puffer abgeschlossen (*committed*) sind und (2) das aktuelle IRQL-Level die Signalisierung von Ereignissen erlaubt. Ist die zweite Bedingung nicht erfüllt, wird in der globalen WMK Verwaltungsdatenstruktur ein entsprechendes Flag gesetzt. Bei der nächsten Gelegenheit (= Aufruf von `WmkAllocateEvent` mit passendem IRQL Level) wird dann der Logdatei-Thread aktiviert.

Nach der Threadaktivierung akquiriert der Thread zuerst ein `DiskWriterLock`. Über dieses Lock werden gleichzeitige Aufrufe des Systemaufrufs `NtWmkFlushLogfile` synchronisiert. Mit Hilfe dieses Systemaufrufes könnte ein Usermode-Programm ebenfalls das Schreiben des aktuellen Puffers auslösen.

Der Thread schreibt zuerst einen Checkpoint, der Informationen über den Pufferinhalt enthält. Die Informationen sind dabei ein Zeitstempel, die Datengröße und die Anzahl der verlorenen Ereignisse. Anschließend werden die Daten geschrieben. Abschließend gibt der Thread das `DiskWriterLock` wieder frei und wartet auf das nächste `FlushEvent`.

Beim Start des Systemthreads wird der Pfad der Logdatei mit `\SystemRoot\wmk00000.log` initialisiert. Dabei ist garantiert, dass das Verzeichnis `SystemRoot` existiert und verwendet werden kann. Im Allgemeinen sind die Logdateien dann im Verzeichnis `c:\windows\system32\` zu finden.

Zur Kompilierzeit des WMK wird eine maximale Größe der Logdateien festgelegt. Diese Größe kann beispielsweise verhindern, dass die maximale Dateigröße des verwendeten Dateisystems überschritten wird. Würde die aktuelle Logdatei die zulässige Größe überschreiten, wenn der aktuelle Puffer zusätzlich geschrieben wird, so wird sie geschlossen und eine neue Datei angelegt. Die Logdateien werden dabei fortlaufend durchnummeriert, beginnend mit

`wmk00000.log`. Die Werkzeuge zur Bearbeitung der Logdateien können Informationen aus verschiedenen Dateien wieder kombinieren.

## 3.4 Werkzeuge

Zusätzlich zu den Kernkomponenten des WMK wurde eine Reihe von Werkzeugen entwickelt. In diesem Abschnitt werden Werkzeuge aus den Bereichen (1) Steuerung der Ereignisaufzeichnung und (2) Auswertung von Logdateien kurz vorgestellt.

### 3.4.1 Steuerung der Ereignisaufzeichnung

Der WMK verwendet einen statischen Ansatz zur Instrumentierung, das heißt die erzeugten Ereignisse werden zur Kompilierzeit des Kernels definiert. Verschiedene Aspekte der Ereignisaufzeichnung können jedoch zur Laufzeit des WMK konfiguriert werden.

Zu diesem Zweck wurden zwei Steuerungsprogramme entwickelt: `flushlogfile` und `procmonitor`.

Der WMK verwendet *Doublebuffering* für die Aufzeichnung von Ereignissen. Die Umschaltung zwischen den beiden Puffern erfolgt, wenn der aktive Puffer gefüllt ist. Mit `flushlogfile` kann die Umschaltung zwischen aktivem und passiven Puffer ausgelöst werden, unabhängig vom Füllstand des aktiven Puffers. Die aufgezeichneten Ereignisse werden nach der Umschaltung in die Logdatei geschrieben.

Mit `procmonitor` kann bestimmt werden, welche Aktivitäten überwacht und welche Ereignisse aufgezeichnet werden sollen.

Die grundsätzlichen Randbedingungen werden statisch bei der Kompilierung des WMK vorgegeben: globale Ereignisaufzeichnung oder prozess-bezogene Aufzeichnung und vorhandene Instrumentierungspunkte. Diese Konfigurationen lassen sich nicht ändern.

Ist die globale Ereignisaufzeichnung aktiviert, kann mit der Hilfe von `procmonitor` die Ereignismaske, also die Auswahl der aufgezeichneten Ereignisse, modifiziert werden. Diese Einstellung gilt dann für alle Aktivitäten im System.

Ist die prozess-bezogene Aufzeichnung aktiviert, kann ebenfalls die Ereignismaske definiert werden. Zusätzlich bietet `procmonitor` zwei Möglichkeiten zur Auswahl von zu beobachtenden Aktivitäten:

***Attach*-Mode:** ein bereits laufender Prozess kann zur Aufzeichnung ausgewählt werden. Durch einen Systemaufruf wird das `MonitorProcess`-Feld der `EPROCESS`-Datenstruktur auf `TRUE` gesetzt.

**Prozesserzeugung:** Mit der Hilfe von `procmonitor` kann einer neuer Prozess sofort unter Beobachtung gestartet werden. Der so gestartete Prozess (und alle von ihm erzeugten Prozesse) können auf diese Weise analysiert werden.

### 3.4.2 Auswertung von Logdateien

Für die Auswertung von Logdateien ist ein generisches Werkzeug entwickelt worden, das durch Erweiterungsmodule ergänzt werden kann. Als Implementierungssprache wurde dabei C# verwendet.

Grundsätzlich erfolgt die Auswertung ereignisstrom-basiert, das heißt die geschriebenen Ereignisse werden in der Reihenfolge, in der sie in der Logdatei gespeichert sind, in das Werkzeug eingelesen. Das Werkzeug analysiert diesen Ereignisstrom. Jedes Werkzeugmodul muss das in Listing 3.7 dargestellte Interface implementieren.

**Listing 3.7:** Interface von WMK Werkzeugen zur Logdateianalyse

```
1 public interface IEventTool {
2
3   void processEvent(WmkEventHeader weh, WmkEvent we);
4
5   void startProcessing(string resultFileName);
6   void finishedProcessing();
7 }
```

Die Funktion `startProcessing` wird am Anfang der Analyse genau einmal aufgerufen. Das Werkzeug kann dann die Analyse initialisieren und beispielsweise Ergebnisdateien anlegen.

Nach Abschluss der Analyse wird `finishedProcessing` gerufen und das Werkzeug schließt die Bearbeitung ab. Dabei sollen beispielsweise eventuell geöffnete Dateien geschlossen und verwendete Ressourcen freigegeben werden.

Der Kern eines jeden Werkzeugs ist die Funktion `processEvent`. Diese Funktion erhält als Parameter Informationen zum aktuell zu analysierenden Ereignis: den `WmkEventHeader`, ein Objekt, das die Ereignisheader-Daten kapselt, und das eigentliche Ereignis als `WmkEvent`-Objekt. Mit diesen Informationen kann dann die Analyse durchgeführt werden.

Durch das beschriebene Werkzeugmodul-Modell sind Analysewerkzeuge mit relativ geringem Aufwand implementierbar. Ein Rahmenprogramm lädt, initialisiert und ruft Module. Konfigurationsparameter erlauben dabei die Auswahl einer Menge von Modulen und die Spezifikation von Filtern für die zu analysierende Logdatei.

Die Logdatei kann dabei bezüglich der folgenden Aspekte gefiltert werden:

**Sequenznummer:** Ein Bereich der Logdatei kann durch einen Bereich von Ereignissequenznummern angegeben werden. Beispielsweise „analysiere die ersten x Ereignisse“ oder „analysiere ab dem x. Ereignis bis zum y. Ereignis“.

**Zeitstempel:** Alternativ zur Sequenznummer können auch Zeitstempel zur Auswahl von Bereichen in der Logdatei verwendet werden.

**Ereignistypen:** Sollen nur bestimmte Ereignistypen analysiert werden, oder sollen bestimmte Ereignistypen explizit von der Analyse ausgeschlossen werden, kann ein entsprechender Filter definiert werden.

**Thread- und Prozess-ID:** Analog zu den den Ereignistypen können auch bestimmte Threads und Prozesse zur Analyse selektiert oder explizit von der Analyse ausgeschlossen werden.

Geladene Module analysieren dann nur den Teil-Ereignisstrom, der aus der Anwendung der spezifizierten Filterbedingungen resultiert. Die Filterung kann gleichzeitig (sofern sinnvoll) bezüglich mehrerer der dargestellten Aspekte erfolgen.

Nachfolgend werden die Werkzeugmodule `DumpLogfile` und `EventStats` kurz vorgestellt.

Mit Hilfe von `DumpLogfile` kann die im Binärformat gespeicherte Logdatei in eine einfach lesbare Textform umgewandelt werden. Der Auszug einer umgewandelten Logdatei ist in Listing 3.8 dargestellt.

**Listing 3.8:** Beispiel: In Textform umgewandelte WMK Logdatei

```
[..]
0 22510305  CREATE_OBJECT 04 08 0x823A5370 Callback \Callback\PowerState
0 22544856        SYSCALL 04 08 27 NtClose
0 22554270   SYSCALL_EXIT 04 08
0 23417955        SYSCALL 04 08 289 NtCreateKeyedEvent
0 23514984  CREATE_OBJECT 04 08 0xE10027B8 KeyedEvent \KernelObjects\
   CritSecOutOfMemoryEvent
0 23679504   SYSCALL_EXIT 04 08
0 23722074        SYSCALL 04 08 27 NtClose
0 23733333   SYSCALL_EXIT 04 08
0 34958421 CONTEXT_SWITCH 00 00 0 127
0 60677082 CONTEXT_SWITCH 04 08 31 36
[..]
```

Die erste Spalte zeigt die Nummer der CPU, auf der das Ereignis auftrat. Die nächsten Spalten enthalten den Zeitstempel, den Ereignistyp, die Thread-ID und die Prozess-ID. Die Informationen in der restlichen Zeile sind vom konkreten Ereignistyp abhängig. Für ein `CREATE_OBJECT`-Ereignis sind beispielsweise die Objektadresse, der Objekttyp und der Objektname dargestellt.

Das Werkzeugmodul `EventStats` gibt verschiedene Statistiken über eine gegebene Logdatei aus. Listing 3.9 zeigt ein Beispiel für die Ausgabe des Werkzeugs.

**Listing 3.9:** Beispiel: Statistische Auswertung einer WMK Logdatei

```
open file wmk00000.log
size = 311889044

events  =   10887236
dropped =          0 [0.0%]

- Event Type -------- Count --------- Size ------------ Avg -

              UNKNOWN       0 ( 0.0%)         0 ( 0.0%)
     PROCESS_CREATION      86 ( 0.0%)      3440 ( 0.0%)  40
  PROCESS_TERMINATION      72 ( 0.0%)      1440 ( 0.0%)  20
      THREAD_CREATION    7585 ( 0.1%)    485880 ( 0.2%)  64
   THREAD_TERMINATION    7369 ( 0.1%)    176856 ( 0.1%)  24
           WAIT_EVENT  990964 ( 9.1%)  42036824 (13.5%)  42
              SYSCALL 4453178 (40.9%) 124688984 (40.0%)  28
       CONTEXT_SWITCH  449445 ( 4.1%)  12584460 ( 4.0%)  28
         WAIT_RELEASE  168778 ( 1.6%)   6076008 ( 2.0%)  36
         SYSCALL_EXIT 4448376 (40.9%) 106761024 (34.2%)  24
          QUANTUM_END     841 ( 0.0%)     20184 ( 0.0%)  24
        CREATE_OBJECT  166822 ( 1.5%)   7196760 ( 2.3%)  43
     TIMER_EXPIRATION  158389 ( 1.5%)   6335560 ( 2.0%)  40
          CREATE_FILE   35331 ( 0.3%)   5521384 ( 1.8%) 156

-------------------------------------------------------- 28 -
```

Ausgegeben werden der Name der Logdatei und allgemeine Informationen über die Größe der Datei sowie die Anzahl der aufgezeichneten und verlorenen Ereignisse. Die Tabelle zeigt dann Informationen über die einzelnen Ereignistypen: Häufigkeit (absolut und prozentual bezogen auf die Gesamtzahl der Ereignisse), Datenmenge für den entsprechenden Ereignistyp und durchschnittliche Größe eines einzelnen Ereignis.

Mit Hilfe dieser generischen Werkzeuge können grundlegende Analysen der WMK-Logdatei durchgeführt werden.

## 3.5 Evaluation

In diesem Abschnitt werden Evaluierungsergebnisse für den WMK angegeben. Dabei standen Micro-Benchmarks im Vordergrund. Die Anwendbarkeit des WMK im Rahmen komplexerer Fallstudien wird im Kapitel 5 dargestellt.

### 3.5.1 Testumgebung und Methodologie

Der WMK Overhead wird im Wesentlichen durch drei Faktoren bestimmt: der Anzahl der aufzuzeichnenden Ereignisse, der Datenmenge und der Puffergröße.

1. Die Anzahl hängt von Art und Eigenschaft der ausgeführten Testlast ab. Eine Ein-/Ausgabeintensive Anwendung führt zu einer großen Zahl von Systemaufrufen, eine rechenintensive Anwendung wird deutlich weniger Systemaufrufe durchführen, dafür aber eventuell mehr Kontextwechselereignisse erzeugen.
2. Die Datenmenge hängt direkt von den Typen der auftretenden Ereignisse ab: bei einem `CreateFile`-Ereignis wird beispielsweise der Dateiname in Unicode-Representation gespeichert, bei einem `QuantumEnd`-Ereignis nur die entsprechenden Thread-Informationen.
3. Die Puffergröße bestimmt die Frequenz, mit der gefüllte Puffer in die Logdatei geschrieben werden müssen. Bei der gleichen Testlast müssen kleinere Puffer häufiger geschrieben werden. Größere Puffer erhöhen jedoch die benötigte Zeit pro Schreibvorgang.

Diese drei Faktoren wurden mit dem nachfolgend beschriebenen Testverfahren untersucht.

Als Testsystem wurde eine Workstation verwendet, die über eine Intel Pentium 4 (2.66 GHz) CPU und 768 MByte RAM verfügt. Die Tests wurden mit einem nativ gebooteten Windows Betriebssystem ausgeführt, das heißt insbesondere, dass zur Evaluation keine virtuelle Maschine verwendet wurde. Der original-Kernel `ntoskrnl.exe` (Windows Server 2003 Enterprise Edition SP1) wurde durch eine Testversion des WMK ersetzt.

Als Testlast wurde der Windows Research Kernel auf dem Testsystem übersetzt und anschließend der Sourcecode-Baum wieder bereinigt. Diese Testlast erzeugt zahlreiche Dateioperationen und Systemaufrufe. Weiterhin wird auch die CPU belastet, so dass die Dauer für einen Zyklus aus Übersetzen und Bereinigen eine gute Maßzahl für die Systemleistung darstellt.

Wenn nicht anders angegeben, beziehen sich die Messwerte auf jeweils 10 Zeitmessungen für jeweils 10 Zyklen aus Übersetzen und Bereinigen.

### 3.5.2 Messungen und Analysen

Die Synchronisation der Pufferzugriffe im WMK ist nicht-blockierend implementiert worden. Dabei wird ein *Allocate-Commit* Schema zur Lock-freien Synchronization verwendet. Als erstes wurde die Dauer der Allocate- und Commit-Funktionsaufrufe untersucht. Es ergab sich eine Verteilung wie in Abbildung 3.2 dargestellt.

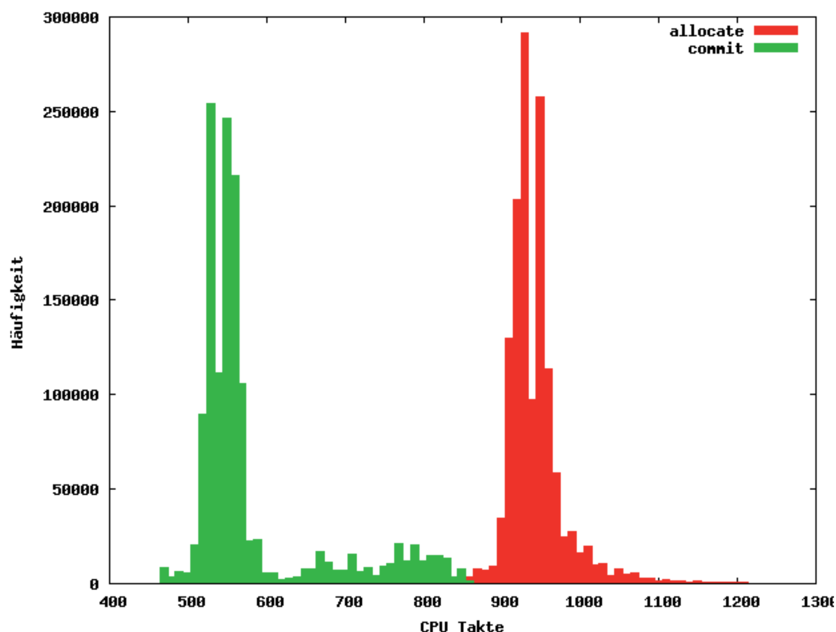



**Abbildung 3.2:** Analyse: Dauer von Allocate- und Commit-Aufrufen

Gemessen wurde die Anzahl von CPU Zyklen die für einen Allocate- oder Commit-Funktionsaufruf benötigt wurden. Das Histogramm stellt die Verteilung der Messwerte für 1,3 Mio. aufgezeichnete Ereignisse dar. Für Allocate-Aufrufe werden mehr Zyklen benötigt da das aufzuzeichnende Ereignis initialisiert werden muss. Die Streuung der Messwerte ist durch den verwendeten Synchronisationsmechanismus erklärbar: die `cmpxchg`-Aufrufe können kollidieren und müssen in einem solchen Fall (mehrfach) wiederholt werden.

Unter Berücksichtigung der Varianz ergibt sich, dass 90% der Aufrufe von Allocate im Bereich $942 \pm 111$ Zyklen und der Aufrufe von Commit im Bereich $578 \pm 255$ Zyklen liegen. Bei einer CPU Taktfrequenz von 2.66 GHz ergeben sich näherungsweise[2] Zeiten von $377 \pm 44$ ns für Allocate- und $231 \pm 102$ ns für Commit-Aufrufe.

Neben den dargestellten Werten für einzelne Allocate- und Commit-Aufrufe ist die Auswirkung der Instrumentierung auf das Gesamtsystem interessant. Bei Verwendung des originalen Windows Kerns dauerte ein Compilerdurchlauf des WRK 165.7 Sekunden. Dieser Basiswert wurde durch 250 Durchläufe bestimmt.

Die Systemverlangsamung durch den WMK muss in Bezug auf die Ereignisrate analysiert werden. Alle in Tabelle 3.1 dargestellten Ereignisse waren während der Testdurchläufe aktiviert. Diese Konfiguration führte zu ca. 13 Millionen Ereignissen bei einem Compilerdurchlauf. Jedes dieser Ereignisse hatte eine durchschnittliche Größe von 32 Bytes. Die resultierende Logdatei für 10 Durchläufe war demnach 3.3 GByte groß. Bei einer Zeit von etwa 170 Sekunden pro

[2] Auf x86-Systemen kann zwar ein prozessortakt-genauer Zeitstempel abgefragt werden; die tatsächliche Taktfrequenz kann jedoch nur mit Hilfe anderer Zeitquellen näherungsweise bestimmt werden.

| Puffergröße [MB] | Overhead / Verzögerung [%] | Ereignisverlust [%] |
|---|---|---|
| 2 | 6.5 | 14.3 |
| 4 | 6.3 | 7.2 |
| 8 | 6.3 | _ |
| 16 | 6.1 | _ |
| 32 | 6.4 | _ |
| 64 | 6.3 | _ |
| 96 | 6.2 | _ |

**Tabelle 3.2:** WMK Overhead nach Puffergröße

Compilerdurchlauf musste der WMK in den durchgeführten Tests ca. 64000 Ereignisse pro Sekunde aufzeichnen.

Tabelle 3.2 zeigt die Systemverlangsamung in Abhängigkeit von der Puffergröße. Zwei Beobachtungen lassen sich aus den Ergebnissen ableiten:

1. Der WMK Overhead ist annähernd konstant und unabhängig von der Puffergröße. Die Abwägung zwischen Puffergröße und Häufigkeit der Logdatei-Schreibvorgänge scheint (in der zur Evaluierung verwendeten Testumgebung) keine Rolle zu spielen.

2. Mit einer Puffergröße $\geq$ 8 MByte werden keine Ereignisse verloren. Ereignisverlust ist nicht akzeptabel für eine sinnvolle Systemanalyse, 8 MByte bilden daher die minimal zu verwendende Puffergröße.

Die tatsächlich notwendige Puffergröße wird auch durch die Geschwindigkeit des verwendeten Logdatei-Speichermediums bestimmt. Dieser Zusammenhang wurde nicht untersucht, sollte aber nur geringen Einfluss haben.

Für die maximal mögliche Puffergröße konnte bei den Messungen festgestellt werden, dass bei einer Puffergröße $>$ 128 MByte das Testsystem nicht mehr erfolgreich booten konnte. Die Reservierung von mehr als 128 MByte des verfügbaren Hauptspeichers im non-paged Pool-Bereich scheint zu viel dieser knappen Ressource zu verbrauchen. Die Größe des non-paged Pools wird beim Systemstart durch eine Heuristik basierend auf der Gesamtspeichergröße bestimmt. Der WMK könnte mit einem ähnlichen Ansatz auch die Puffergröße erst dynamisch beim Systemstart festlegen.

Zusammenfassend gilt: die WMK Puffergröße sollte so gewählt werden, dass keine Ereignisse verloren gehen. Mit einer Ereignisrate von ca. 64000 Ereignissen pro Sekunde verursacht der WMK eine Systemverlangsamung von ca. 6%.

Tabelle 3.3 zeigt den Overhead des WMK in Abhängigkeit von den verschiedenen Ereignistypen. Zur Bestimmung wurde die Testlast ausgeführt und ausschließlich die zu untersuchenden Ereignisse aufgezeichnet. Spalte 2 der Tabelle zeigt die Systemverlangsamung bei Aufzeichnung bestimmter Ereignisse. Spalte 3 enthält einen Näherungswert für die Ereignisrate der entsprechenden Ereignisse.

| Ereignistyp | Overhead / Verzögerung [%] | Ereignisrate [$\approx \frac{1}{s}$] |
|---|---|---|
| *keine aktivierten Ereignisse* | 1.1 | _ |
| Process* | 1.1 | 3 |
| Thread* | 1.3 | 4 |
| Wait* | 1.2 | 580 |
| Syscall* | 5.2 | 53604 |
| ContextSwitch | 1.1 | 395 |
| CreateObject | 1.8 | 5159 |
| CreateFile | 1.8 | 4901 |
| TimerExpiration | 1.1 | 97 |

**Tabelle 3.3:** WMK Overhead nach Ereignistyp

Aus den Messergebnissen lassen sich die folgenden Regeln ableiten:

1. Der WMK verlangsamt das Gesamtsystem bei einer Ereignisrate von 64000 Ereignissen pro Sekunde um etwa 1.1%, wenn keine Ereignisse tatsächlich aufgezeichnet werden. Bei einem inaktiven WMK muss bei jedem Instrumentierungspunkt getestet werden, ob das entsprechende Ereignis tatsächlich aufgezeichnet werden muss. Diese Grundbelastung durch den WMK führt zu 1.1% Overhead.
2. Das Auftreten von etwa 13000 Ereignissen pro Sekunde führt zu einer weiteren Systemverlangsamung von 1%. Mit Hilfe dieser Regel kann bei bekannter Ereignisrate die Systembeeinflussung abgeschätzt werden.
3. Bei der gewählten Testlast sind mehr als 80% der auftretenden Ereignisse Systemaufrufereignisse. Diese Ereignisse sind allein für etwa 5.2% der Systemverlangsamung die Ursache.

Zusammenfassend lässt sich feststellen, dass der WMK das zu analysierende System und die ausgeführten Anwendungen beeinflusst. Der Einfluss setzt sich aus den folgenden Komponenten zusammen: (1) dem Test, ob bei einem Instrumentierungspunkt das Ereignis tatsächlich aktiviert ist und aufgezeichnet werden soll und (2) der eigentlichen Ereignisdatenerfassung. Entscheidender Faktor für die Systembeeinflussung ist die Ereignisrate, also wie viele Ereignisse pro Sekunde verarbeitet werden müssen.

Die Hardwareplattform (CPU, Arbeitsspeicher und Festplatte) bestimmt den tatsächlich bei Tests auftretenden Overhead. Der Zusammenhang zwischen der Plattform und der WMK Systembeeinflussung ist nicht weiter als oben dargestellt untersucht worden.

Die bestimmte Systemverlangsamung durch den WMK ist gering und verhindert nicht die sinnvolle Ausführung des Betriebssystemkerns. Daher ermöglicht der WMK die dynamische Analyse von Anwendungen zur Laufzeit.

Es ist jedoch möglich, durch bestimmte Lastszenarien bei aktiviertem WMK das System zu stören und unter Umständen unbenutzbar zu machen: (1) durch Auslösen sehr vieler Ereignisse in sehr kurzer Zeit, beispielsweise durch wiederholte (kurze) Systemaufrufe in einem Thread auf Realtime-Priorität, (2) durch Instrumentierung von Punkten im System, die potentiell sehr viele Ereignisse erzeugen, beispielsweise Spin-Locks.

### 3.5.3 Vergleich mit alternativen Systemen

Der Windows Monitoring Kernel wurde als Instrumentierungsframework für die Forschung entwickelt. Dabei standen eine möglichst einfache Verwendbarkeit und eine einfache Erweiterbarkeit im Vordergrund: Ereignisse und Instrumentierungspunkte sollten mit minimalem Aufwand verwaltet und implementiert werden können.

Dieses Ziel wurde durch die vorgestellte WMK-Implementierung erreicht: Instrumentierungspunkte sind mit zwei Quellcodezeilen definiert; zusätzlich ist noch Code erforderlich, der die Ereignisdaten aufnimmt. Neue Ereignistypen sind mit einer Strukturdefinition in C und der Vergabe einer Ereignis ID-Nummer ebenfalls schnell definiert.

Als Vorbild bei Design und Implementierung des WMK diente KLogger [48] welches ebenfalls eine fein-granulare Infrastruktur zur Aufzeichnung von Ereignissen bereitstellt - allerdings für Linux Systeme. Die wichtigsten Gemeinsamkeiten und Unterschiede sind:

**Konfiguration und Ereignistypen:** KLogger verwendet eine C-ähnliche Spezifikationssprache zur Definition von Ereignistypen. Im WMK wurden alle Konfigurationsparameter und Ereignisdefinitionen in einer Headerdatei (`wmk.h`) zusammengefasst, so dass die Konfiguration und Erweiterung direkt in der C-Syntax erfolgt. Dies stellt keine Einschränkung in der Verwendbarkeit dar, da bei beiden Systemen der Betriebssystemkern neu kompiliert werden muss.

**Definition von Instrumentierungspunkten:** KLogger stellt ein zentrales Preprozessor-Makro `klogger` zur Anzeige von Ereignissen bereit. Diesem Makro werden Ereignistyp und Ereignisparameter übergeben. Der WMK verwendet ein anderes Schema für Instrumentierungspunkte: explizite *allocate*- und *commit*-Aufrufe. Sind komplexere Operationen bei der Sammlung von Ereignisdaten notwendig, ermöglicht das WMK-Schema eine explizite Trennung von Instrumentierungs- und Kernelcode, sowie einen (subjektiv empfundenen) übersichtlicheren Quellcode. Darüber hinaus sichert das WMK-Schema die (typ-)sichere Verwendung der Felder der Ereignisdatenstrukturen: die Parameterübergabe beim KLogger Makro basiert auf der definierten Feldreihenfolge.

**Pufferverwaltung:** KLogger verwendet einen Speicherbereich, der unterteilt ist einen regulären Pufferbereich und einen Reservebereich. Wird der Reservebereich erstmalig beschrieben, also zur Aufzeichnung von Ereignisdaten verwendet, sind vier Schritte erforderlich, um den Puffer in die Logdatei zu schreiben: (1) der Systemthread zum Schreiben des Puffers wird aktiviert, (2) der reguläre Pufferbereich wird in die Logdatei geschrieben und weiterhin auftretende Ereignisse werden im Reservebereich gespeichert, (3) die Schreibposition im Puffer wird auf den Anfang des Puffers gesetzt, und (4) die im Reservebereich gespeicherten Ereignisse werden in die Logdatei geschrieben. Der WMK verwendet *Doublebuffering* mit zwei gleich großen Puffern und kann daher auf Schritt (4) im oben skizzierten Schema verzichten. Weiterhin vereinfacht sich die Implementierung der Pufferverwaltung, da der Sonderfall beim Zugriff auf den Reservebereich nicht auftritt[3].

[3] Im Rahmen der WMK-Entwicklung sind beide Varianten implementiert und ausprobiert worden. Die jetzt gewählte und in dieser Arbeit beschriebene Variante erschien aus den genannten Gründen besser.

Nachfolgend werden alternative Systeme zur Instrumentierung und Analyse des Windows Betriebssystemkerns mit dem WMK verglichen.

Das umfassendste Werkzeug zur Systemanalyse im Windows Umfeld ist der Perfomance-Monitor (*perfmon*) [84]. Mit Perfmon lassen sich die in Windows eingebauten *Performance Counter* (= Messwerte) abfragen und graphisch darstellen. Perfmon verwendet einen statischen sampling-basierten Ansatz. Es ist daher nicht möglich beispielsweise jeden Kontext-Wechsel oder ganze Ketten von Ereignissen zu erfassen.

Erweiterte Untersuchungsmöglichkeiten bieten die Werkzeuge *Windows Management Instrumentation* (WMI) und *Event Tracing for Windows* (ETW). WMI basiert ebenfalls auf dem Konzept der Trennung von Ereignisprovider und Analysewerkzeug. Der ETW *NT Kernel Logger* ist als WMI Ereignisprovider implementiert.

ETW ist Teil der Windows Treiber Entwicklungsumgebung (DDK). ETW erlaubt die Aufzeichnung von Ereignissen im Betriebssystemkern und in Usermode-Anwendungen.

Für einen Vergleich von ETW mit dem WMK wurden die im letzten Abschnitt beschriebenen Messungen auf dem original Windows Kernel mit aktiviertem ETW wiederholt. Bei 50 Tests benötigte ein Zyklus aus Übersetzen und Bereinigen auf dem ETW System 167.17 Sekunden. Dies entspricht einem Overhead von 0.86% bei Verwendung des ETW Werkzeuges `tracelog` im „NT Kernel Logger“ Modus. Weitere Kommandozeilenparamter waren: `-nonet` um Netzwerkereignisse abzuschalten und `-UseCPUCycle` für die Verwendung von präziseren Zeitstempeln.

ETW und der entwickelte Windows Monitoring Kernel haben jedoch unterschiedliche Einsatzgebiete und lassen sich daher nicht direkt vergleichen:

1. ETW dient in erster Linie Treiberentwicklern zur Optimierung ihrer Implementierung und Systemadministratoren zur Analyse von Problemen zur Laufzeit. Der WMK ist primär für den Einsatz in der Forschung vorgesehen.

2. Der Einsatz von ETW in Produktivumgebungen führt dazu, dass ETW generische Konzepte zur Instrumentierung (WMI Integration) verwendet und dadurch schwerer zu programmieren und einzusetzen ist. Die WMK Schnittstelle ist wesentlich einfacher gestaltet.

3. Die Menge der Ereignistypen, die durch ETW *im Betriebssystemkern* aufgezeichnet werden kann, ist nicht erweiterbar: die Menge der vorgegebenen Ereignisse umfasst beispielsweise Kontextwechsel, Threaderzeugung und Dateizugriffe. Die Analyse von Synchronisationsbeziehungen zwischen Threads oder Systemaufrufe lassen sich nicht mit ETW analysieren.

4. Der ETW Overhead von 0.86% wurde bei einer relativ geringen Ereignisrate von 4700 Ereignissen pro Sekunde gemessen - es waren jedoch alle vom ETW Kernel Logger bereitgestellten Kernelereignisse aktiviert. Der WMK kann bei einem Overhead von 1% etwa 13000 Ereignisse pro Sekunde verarbeiten. Beide Instrumentierungssysteme zeigen also vergleichbare Leistungseigenschaften.

Weiterhin ist festzustellen, dass die Produktdokumentation von ETW unvollständig ist und nur wenig Hilfestellung bei der Implementierung zusätzlicher Ereignistypen im Betriebssystemkern bietet.

Zusammenfassend lässt sich feststellen, dass der WMK nicht in Produktivumgebungen eingesetzt werden kann, da er auf nicht vom Hersteller unterstützten Betriebssystemkernänderungen beruht und einen Systemneustart zur Installation oder bei wesentlichen Änderungen

erfordert. Als Forschungswerkzeug bietet der WMK zwei Vorteile: (1) bessere Performanz beziehungsweise geringerer Overhead durch leichtgewichtige Integration in den Windows Kern und (2) wesentlich einfachere Erweiterbarkeit um neue Instrumentierungspunkte.

In Abschnitt 6.1 werden der WMK und seine Verwendung in den Kontext verwandter Forschungsarbeiten eingeordnet.

## 3.6 Zusammenfassung

Die vom WMK bereitgestellte Infrastruktur erlaubt die effiziente Aufzeichnung von Ereignissen im Betriebssystemkern. Sie kann als Grundlage für die Verarbeitung von Kernel-Ereignisströmen verwendet werden.

Für die vorgesehenen Einsatzfälle ist der WMK sehr gut geeignet. Die Testfälle und Fallstudien (siehe Kapitel 5) zeigen sowohl Leistungsfähigkeit als auch Einschränkungen des WMK.

Um noch weitere Einsatzgebiete abzudecken sind an einigen Stellen Verbesserungen und Erweiterungen notwendig:

**Dynamische Aktivierung von Ereignissen:** In der vorgestellten Implementierung des WMK sind die Instrumentierungspunkte auf Ebene des Quellcode festgelegt und werden beim Übersetzen des Kerns eingefügt. Zur Laufzeit ist dann bei jedem Instrumentierungspunkt mindestens ein Vergleich notwendig um festzustellen, ob das entsprechende Ereignis tatsächlich aufgezeichnet werden soll.

Bei der Verwendung dynamischer Instrumentierungstechniken verursachen nicht aktivierte Instrumentierungspunkte keine Systemlast. Der Instrumentierungscode wird dann nur bei Bedarf dynamisch aktiviert.

**Verbesserung des Puffermanagements:** Das beschriebene Vorgehen bei der Aufzeichnung eines Ereignisses - (1) Reservieren von Speicherplatz, (2) Schreiben der Ereignisdaten, und (3) Abschließen des Ereignisses - wird problematisch, wenn nicht jedes Ereignis abgeschlossen wird. In einem solchen Fall können die Puffer nicht auf die Festplatte geschrieben werden, da noch offene Ereignisse existieren.

Dieses Problem existiert, ist jedoch in keinem der zahlreichen Tests und Anwendungen des WMK tatsächlich aufgetreten. Eine Lösung kann in der Verwendung von feingranularen Puffern, die in kürzeren Abständen freigegeben werden, gefunden werden.

Für beide Verbesserungsbereiche existieren Ansätze in verwandten Arbeiten, die in den WMK integriert werden können.

Der WMK bildet die Grundlage für die weiteren im Rahmen der vorliegenden Arbeit entwickelten Konzepte zur Verarbeitung von Ereignisströmen im Betriebssystemkern.

Kapitel 4

# Online Verarbeitung von Ereignissen

*In diesem Kapitel wird eine Sprache zur Spezifikation von Ereignismustern und -regeln, sowie eine Laufzeitumgebung zur dynamischen Verarbeitung von Ereignisströmen beschrieben. Weiterhin werden eine Programmierschnittstelle und Werkzeuge vorgestellt, die auf den entwickelten Konzepten aufbauen und es Anwendungen ermöglichen auf Muster innerhalb des im System auftretenden Ereignisstroms zu reagieren.*

*Das beschriebene Konzept stellt einen neuen Ansatz zur Verarbeitung von Ereignisströmen direkt im Betriebssystemkern dar: Die Analyse und die Reaktion auf erkannte Ereigniskonstellationen erfolgen synchron im Ausführungskontext der Aktivität, die die Konstellation verursacht hat.*

## 4.1 Zielstellung der Implementierung

In Kapitel 2 wurde das Gebiet der Überwachung und Instrumentierung von Softwaresystemen dargestellt. Dabei sind zwei Konzepte zur Erfassung des Zustands eines Softwaresystem unterschieden worden - Sampling und Tracing.

Das Ziel der im Rahmen dieser Arbeit entwickelten Laufzeitumgebung zur online Verarbeitung von Ereignissen [87] ist, die Tracing-spezifischen Nachteile abzuschwächen - in Szenarien, in denen der Einsatz von Tracing zur Systemanalyse sinnvoll ist.

Die bearbeiteten Nachteile von Tracing sind:

- Große Datenmengen bei der Analyse - Der Einsatz von Tracing führt in der Regel zu Logdateien, die nicht selten Größen von mehreren Gigabyte erreichen. Der Analyst muss jetzt diese Daten verarbeiten und nach auffälligen Mustern durchsuchen.

  In dieser Arbeit wird vorgeschlagen, die Ereignisse online (das heißt sofort nach ihrem Auftreten) zu verarbeiten und so idealerweise auf eine explizite Speicherung in einer Logdatei zu verzichten.

- Bei der Analyse von Logdateien werden spezifische Muster gesucht. Diese Aufgabe kann automatisiert werden, wenn die zu suchende Muster beziehungsweise das zu erkennende Systemverhalten entsprechend spezifiziert werden.

  Nachfolgend wird eine Sprache vorgestellt, die die Beschreibung von Mustern beziehungsweise Ereigniskonstellationen[1] in Ereignisströmen ermöglicht.

[1] Die Begriffe Muster und Ereigniskonstellation werden im nachfolgenden Text synonym verwendet.

- Sampling-basierte Überwachung kann mit Hilfe von Konzepten aus der Regelungstechnik automatisiert werden - bei Tracing-basierten Konzepten entsteht in der Regel ein zeitlicher Bruch zwischen der Analyse und der Reaktion.

  Die in der vorliegenden Arbeit entwickelte Laufzeitumgebung ermöglicht die automatische Reaktion auf erkannte Konstellationen durch den Aufruf von Callbacks. Die Callbacks können entweder direkt im Kernel oder im Kontext einer Usermode-Anwendung ausgeführt werden.

In den nachfolgenden Abschnitten werden die entwickelten Konzepte detailliert dargestellt.

## 4.2 Beschreibung von Mustern in Ereignisströmen

Die Beschreibung von zu erkennenden Mustern in einer Folge von Objekten ist ein Standardproblem im Kontext der Zeichenkettenverarbeitung - reguläre Ausdrücke bieten die Möglichkeit zur flexiblen Spezifikation beliebiger Subzeichenketten. Ein wichtiges Anwendungsfeld ist der Compilerbau [22].

Die Erkennung von Mustern in einem Ereignisstrom unterscheidet sich jedoch in einigen Bereichen von der Zeichenkettenverarbeitung:

**Eingabealphabet:** Bei der Verarbeitung von Zeichenketten wird im Allgemeinen angenommen, dass diese über einem bekannten und endlichen Eingabealphabet definiert sind. Werden die potentiell zu verarbeitenden Ereignistypen als Elemente des Eingabealphabets betrachtet, so ist bei der Spezifikation eines Musters unter Umständen nicht klar, welche Gesamtmenge von Ereignistypen im System auftreten können.

**Zeichenkette:** Reguläre Ausdrücke werden in den meisten Werkzeugen auf einzelne, endliche Zeichenketten angewendet. Ein Ereignisstrom ist dagegen kontinuierlich, das heißt insbesondere, dass kein eindeutiger Anfang und kein eindeutiges Ende einer zu verarbeitenden Ereignisfolge identifizierbar ist.

**Verarbeitung:** Auch das Verarbeitungsmodell unterscheidet sich, wenn keine diskreten Eingabewerte sondern ein kontinuierlicher Eingabestrom geprüft werden muss: Das Ergebnis einer Zeichenkettenverarbeitung ist im Allgemeinen „akzeptiert“, wenn das beschriebene Muster erkannt wurde, oder eben „nicht akzeptiert“. Im Ereignistrom soll das Auftreten eines definierten Musters sofort signalisiert werden.

Die gezeigten Besonderheiten müssen sowohl bei der Spezifikation der Sprache zur Beschreibung von Ereigniskonstellationen, als auch bei der Implementierung der eigentlichen Laufzeitumgebung berücksichtigt werden.

Im Abschnitt 2.4 wurden die Begriffe „Ereignis“ und „Ereignisstrom“ im Kontext der Verarbeitung im Betriebssystemkern definiert. Eine Sprache zur Beschreibung von Mustern in Ereignisströmen kann jetzt durch die folgenden Merkmale charakterisiert werden.

**Ereignisspezifikation:** Datenelemente eines Ereignisses

Wie bereits definiert, wird ein Ereignis durch die folgenden Elemente definiert: Zeitstempel, Ausführungskontext, und Ereignisdaten.

**Musterspezifikation:** Beschreibung von Musterstrukturen

Ganz abstrakt betrachtet, sind die Sprachkonstrukte Sequenz und Alternative ausreichend um alle möglichen (zusammenhängenden) Teilmengen eines Ereignisstroms zu beschreiben: alle gewünschten Sequenzen werden als Alternativen betrachtet (und explizit aufgezählt).

Existieren jedoch unbekannte Ereignistypen, also Typen, die bei der Musterdefinition nicht bekannt sind, so ist das oben angedeutete Verfahren nicht mehr durchführbar. Wird das folgende Muster betrachtet: „ein Ereignis A soll von einem Ereignis B gefolgt werden, ohne dass dazwischen ein Ereignis C auftritt", kann jede zutreffende Sequenz von Ereignissen aufgezählt werden, wenn alle Ereignistypen bekannt sind. Existieren unbekannte Typen, ist dies nicht möglich.

**Regelspezifikation:** Beschreibung von Beziehungen zwischen Ereignissen

Neben den Kausalitäts- oder Ordnungsbeziehungen zwischen Ereignissen, sind für die Musterbeschreibung eventuell auch andere Relationen relevant. Regeln für Ereignisse ermöglichen die Spezifikation von Relationen zwischen Ereignisdatenfeldern zu anderen Ereignisdatenfeldern, zu spezifizierten konstanten Werten oder zu während der Mustererkennung errechneten Werten. Nur wenn alle vorgegebenen Regeln erfüllt sind, wird das entsprechende Muster erkannt.

Die im Rahmen dieser Arbeit entwickelte prototypische Sprache zur Beschreibung von Mustern in Ereignisströmen ist in EBN Form in Tabelle 4.1 dargestellt. Die Sprache setzt die identifizierten Anforderungen an Ereignisstromverarbeitung im Betriebssystemkern um.

In Listing 4.1 ist ein Beispiel für eine Musterbeschreibung dargestellt.

**Listing 4.1:** Beispiel: Beschreibung eines Musters

```
1 EVENTS "wmkevents.h"
2
3 // time between syscall and syscall exit is greater than 1 second
4 ASYNCHRONOUS RULE longsyscalls
5   STRICTPARTITION PATTERN { [syscall:a, syscallexit:b] }
6   WHERE { [ProcessId],
7           [ThreadId],
8           b.TimeStamp - a.TimeStamp > 2000000 }
9   RETURN { a.SyscallNr }
```

Das Muster erkennt aufeinander folgende Paare von Systemaufruf-Anfang und Systemaufruf-Ende Ereignisse, die im gleichen Thread, innerhalb des gleichen Prozesses auftreten und deren zeitlicher Abstand größer als eine Sekunde ist. Das Muster dient also der Erkennung von lange dauernden Systemaufrufen. Als Ergebnis wird die Nummer des Systemaufrufs zurückgegeben.

In den folgenden Abschnitten werden die einzelnen Elemente einer Regelspezifikation mit Hilfe dieses Beispiels genauer beschrieben.

### 4.2.1 Ereignistypen

In **Zeile 1** des Beispielmusters wird mit dem Schlüsselwort `EVENTS` auf eine Datei verwiesen, die Informationen zu Ereignistypen und deren Datenfeldern enthält. Im Beispiel wird auf die

<table>
<tr><th>Konzept</th><th></th><th>Beschreibung</th></tr>
<tr><td>EPC</td><td>=</td><td>epcstatement { epcstatement }</td></tr>
<tr><td>epcstatement</td><td>=</td><td>( eventdefinition | ruledefinition )</td></tr>
<tr><td>eventdefinition</td><td>=</td><td>"EVENTS" <file name></td></tr>
<tr><td>ruledefinition</td><td>=</td><td>[( "SYNCHRONOUS" | "ASYNCHRONOUS" )] "RULE" <rule name><br>patterndefinition<br>{ rulemodifier }</td></tr>
<tr><td>rulemodifier</td><td>=</td><td>( wheredefinition | whithindefinition | resultdefinition )</td></tr>
<tr><td>patterndefinition</td><td>=</td><td>[( "STRICTSEQUENCE" | "STRICTPARTITION" |<br>"SKIPTILLNEXT" | "SKIPTILLANY" )]<br>"PATTERN" "{" eventstructure "}"</td></tr>
<tr><td>eventstructure</td><td>=</td><td>( eventspecification | eventsequence |<br>eventalternative | eventnegation )</td></tr>
<tr><td>eventsequence</td><td>=</td><td>"[" eventstructure { "," eventstructure } "]"</td></tr>
<tr><td>eventalternative</td><td>=</td><td>"(" eventstructure "|" eventstructure {"|" eventstructure} ")"</td></tr>
<tr><td>eventnegation</td><td>=</td><td>"~" ( eventspecification | eventsequence | eventalternative )</td></tr>
<tr><td>eventspecification</td><td>=</td><td><event type> [ eventlength ] [":" <event name> ]</td></tr>
<tr><td>eventlength</td><td>=</td><td>"[" [ ( <start> ".." <end> | ("<"|"="|">") <number> ) ] "]"</td></tr>
<tr><td>wheredefinition</td><td>=</td><td>"WHERE" "{" wherecondition { "," wherecondition } "}"</td></tr>
<tr><td>wherecondition</td><td>=</td><td>( fieldjoin | relation )</td></tr>
<tr><td>fieldjoin</td><td>=</td><td>"[" <event field name> "]"</td></tr>
<tr><td>relation</td><td>=</td><td>value relationop value</td></tr>
<tr><td>value</td><td>=</td><td>fieldspec [ operation fieldspec ]</td></tr>
</table>

... Fortsetzung auf der nächsten Seite

<table>
<tr><th>Konzept</th><th></th><th>Beschreibung</th></tr>
<tr><td>fieldspec</td><td>=</td><td>( <integer const> | <float const> |<br><event name> ( fieldspecevent | fieldspecarray ) )</td></tr>
<tr><td>fieldspecevent</td><td>=</td><td>"." <event field name></td></tr>
<tr><td>fieldspecarray</td><td>=</td><td>"[" "]" "." ( "len" | ( "avg" | "max" | "min" )<br>"." <event field name> )</td></tr>
<tr><td>operation</td><td>=</td><td>( "+" | "−" | "*" | "/" | "&" | "|" )</td></tr>
<tr><td>relationop</td><td>=</td><td>( "<" | "<=" | "==" | ">=" | ">" | "!=" )</td></tr>
<tr><td>whithindefinition</td><td>=</td><td>"WITHIN" <time></td></tr>
<tr><td>resultdefinition</td><td>=</td><td>"RETURN" ( "SEQUENCE" | "{" value { "," value } "}" )</td></tr>
</table>

**Tabelle 4.1:** Sprache zur Beschreibung von Mustern in Ereignisströmen

C-Datei `wmkevents.h` verwiesen. Wie Ereignisse im Windows Monitoring Kernel (WMK) repräsentiert werden, ist im Kapitel 3 beschrieben.

Aus der angegebenen Datei werden die Informationen extrahiert, die der Compiler für die Verarbeitung der Musterspezifikation benötigt. Dies sind (1) Namen und Typen von Datenelementen, die bei jedem Ereignistyp vorhanden sind (Felder des Ereignisheader), (2) Identifikatoren und Namen von Ereignistypen und (3) Namen und Typen der Datenelemente eines jeden Ereignistyps.

Der Compiler überprüft die typsichere Verwendung der Datenelemente von Ereignissen: Wird der Name eines entsprechenden Feldes verwendet, beispielsweise in einem Vergleich, werden die weiteren verwendeten Felder auf Kompatibilität getestet.

Zur Laufzeit des WMK werden Ereignisse über einen Zahlenwert identifiziert. Auch diese ID wird aus der WMK-Headerdatei entnommen. Bei der Übersetzung von Regeln wird die ID vom Compiler zur Generierung des Binärcodes benötigt.

Die Ereignistypen und der Header werden im WMK durch C-Strukturen definiert. Ereignis-IDs sind durch `#define`-Anweisungen definiert. Eine Komponente des erstellten Compilers verarbeitet die Header-Datei und extrahiert Strukturdefinitionen und ID-Definitionen.

Die benötigten Informationen über Ereignistypen könnten auch auf andere Art bestimmt werden. Möglich wäre es die Sprache um Sprachelemente zu expliziten Deklaration von Ereignistypen zu erweitern. Im Rahmen der vorliegenden Arbeit sollte die vorhanden Instrumentierungsinfrastruktur des Windows Monitoring Kernels verwendet werden. Die Möglichkeit Ereignistypinformationen aus der Datei `wmkevents.h` einzulesen vermindert den Aufwand für Musterspezifikationen.

Im erstellten Prototyp ist der WMK auch die einzige unterstützte Instrumentierungsinfrastruktur. Für die Unterstützung anderer Ansätze sind demnach Anpassungen am Compiler notwendig.

### 4.2.2 Konstellation

In **Zeile 4** werden allgemeine Angaben zur definierten Regel spezifiziert:

```
ASYNCHRONOUS RULE longsyscalls
```

*Modus der Regelauswertung* - Die Auswertung einer Regel kann (wie die Aufzeichnung von Ereignissen auch) synchron oder asynchron erfolgen. Bei der asynchronen Auswertung werden aufgezeichnete Ereignisse unabhängig vom Kontrollfluss der Aktivität bearbeitet, die die Ereignisse ausgelöst hat. Eine parallel ablaufende Aktivität prüft, ob die Ereignisse die definierten Regeln erfüllen. Bei der synchronen Auswertung erfolgt die Regel-Prüfung im Kontrollfluss der ereignisauslösenden Aktivität. Die Abarbeitung wird erst fortgesetzt, wenn die Verarbeitung des aufgetretenen Ereignis abgeschlossen ist.

Die synchrone Auswertung bietet die Möglichkeit, sofort auf erkannte Muster reagieren zu können. Wird beispielsweise eine unerlaubte Operation erkannt, kann die entsprechende Anwendung sofort beendet werden. Die asynchrone Auswertung hingegen kann erst reagieren, wenn die Mustererkennung tatsächlich durchgeführt wurde. Die Beeinflussung der ablaufenden Anwendungen kann jedoch reduziert werden.

Der Modus der Auswertung kann durch die Schlüsselworte `SYNCHRONOUS` beziehungsweise `ASYNCHRONOUS` gewählt werden.

*Name der Regel* - Nach dem Schlüsselwort `RULE` folgt ein Name für die definierte Regel. Über den Namen kann die Regel zur Laufzeit verwaltet werden. Beispielsweise kann die Regel geladen, entladen oder ihr Zustand abgefragt werden.

In der **Zeile 5** wird das zu erkennende Muster definiert.

```
STRICTPARTITION PATTERN { [syscall:a, syscallexit:b] }
```

Vor dem Schlüsselwort `PATTERN` wird die Semantik der Musterbeschreibung spezifiziert, nach dem Schlüsselwort folgt die eigentliche Musterbeschreibung.

*Musterbeschreibung* - Zur Musterbeschreibung werden Konzepte verwendet, die aus regulären Ausdrücken bekannt sind. Als Strukturelemente können Sequenzen, Alternativen und Negationen verwendet werden.

- Eine Sequenz wird durch die Zeichen '`[`' und '`]`' begrenzt. Einzelne Elemente werden durch '`,`' getrennt. Die Musterspezifikation `[A,B]` beschreibt ein Ereignismuster, in dem ein Ereignis vom Typ A gefolgt von einem Ereignis von Typ B auftritt.
- Eine Alternative wird durch die Zeichen '`(`' und '`)`' begrenzt. Einzelne Elemente werden durch '`|`' getrennt. Die Musterspezifikation `(A|B)` beschreibt demnach ein Ereignismuster, in dem entweder ein Ereignis vom Typ A oder ein Ereignis von Typ B auftritt.
- Negationen können verwendet werden, um einzelne Ereignistypen, Sequenzen oder Alternativen zu negieren. Das Zeichen '`~`' wird dazu einer entsprechenden Struktur vorangestellt. `~A` beschreibt, dass an dieser Stelle kein Ereignis vom Typ A im Ereignisstrom vorkommen soll. Entsprechend beschreibt `[A,~B,C]`, dass ein Ereignis vom Typ A gefolgt von einem Ereignis von Typ C auftritt, ohne dass dazwischen ein Ereignis vom Typ B auftrat.

Die dargestellten Strukturelemente einer Musterbeschreibung können verschachtelt werden. Die folgende Beschreibung definiert ein solches zusammengesetztes Muster:

```
[ A , ( B | [ C , D ] ) , ~E , F ]
```

Einem Ereignis vom Typ A soll entweder ein Ereignis vom Typ B oder eine Sequenz von einem Ereignis vom Typ C und einem Ereignis vom Typ D folgen. Anschließend soll ein Ereignis vom Typ F auftreten, ohne dass vorher noch eine Ereignis vom Typ E auftrat.

In den gezeigten Beispielen sind bisher nur Ereignistypen angegeben worden. Durch einen '`:`' abgetrennt können bestimmten Ereignissen im Muster Namen zugewiesen werden. Im Beispiel erhält das `syscall`-Ereignis den Namen `a` und das `syscallexit`-Ereignis den Namen `b`. Relationen zwischen Ereignissen werden über deren Namen definiert.

Welche Ereignisse in einem gegebenen Ereignisstrom tatsächlich auf eine solche Musterbeschreibung zutreffen, wird zusätzlich durch die Semantik der Musterbeschreibung bestimmt. Die Semantik definiert, wie mit unbekannten Ereignistypen umgegangen werden soll.

*Semantik der Musterbeschreibung* - Es stehen vier Modi für die Mustersemantik zur Verfügung: `STRICTSEQUENCE`, `STRICTPARTITION`, `SKIPTILLNEXT` und `SKIPTILLANY`.

Im Modus STRICTSEQUENCE müssen alle Ereignisse, die durch ein gegebenes Muster beschrieben sind, im Ereignisstrom direkt aufeinander folgen. Eventuell autretende unbekannte Ereignistypen führen also dazu, dass das beschriebene Muster nicht erkannt wird.

STRICTPARTITION berücksichtigt eine Untermenge des Ereignisstroms: durch die Verwendung von Join-Feldern (siehe Abschnitt 4.2.3) wird eine Partition des Ereignisstroms definiert. Im Modus STRICTPARTITION müssen passende Ereignisse innerhalb der definierten Partition direkt aufeinander folgen.

Mit SKIPTILLNEXT wird definiert, dass alle Ereignisse übersprungen werden, die nicht zur Musterbeschreibung passen. Das nächste passende Ereignis wird dann in die Ergebnismenge aufgenommen.

Der letzte Modus - SKIPTILLANY - erlaubt es zusätzlich auf die Musterbeschreibung passende Ereignisse zu überspringen. Dieses Verhalten ermöglicht, genau die Exemplare von Ereignissen im Ereignisstrom auszuwählen, die bestimmte Bedingungen erfüllen.

Zur Verdeutlichung der Unterschiede zeigt Tabelle 4.2 einen Beispiel-Ereignisstrom und die Resultate, die unter den verschiedenen Semantiken zurückgegeben werden. Das Verhalten für folgende Regel ist dargestellt:

```
RULE Beispiel <Semantik> PATTERN { [ A, B] } WHERE { [x] }
```

Anstelle von <Semantik> wird die Mustersemantik definiert. Die Bezeichnung der Semantiken sind in der Tabelle abgekürzt angegeben: STRICTSEQUENCE entspricht dabei *SS*, *SP* entspricht STRICTPARTITION, *SN* entspricht SKIPTILLNEXT und *SA* entspricht SKIPTILLANY.

Die Anweisung WHERE { [x] } bedeutet in dem vorliegenden Fall, dass sowohl der Ereignistyp A als auch der Ereignistyp B ein Datenfeld mit dem Namen x enthalten. Außerdem soll für ein erkanntes Muster gelten, dass die konkret erkannten Ereignisse den gleichen Wert im Datenfeld x haben. Die WHERE-Anweisung und der Join-Operator [] werden im Abschnitt 4.2.3 genauer beschrieben.

Jede Zeile in der Tabelle steht für ein mögliches Ergebnis der Mustererkennung. Dabei werden erstmal nur die Ereignistypen betrachtet - ein Ereignis vom Typ A soll von einem Ereignis vom Typ B gefolgt werden. Betrachtet man die Partitionierungsbedingung WHERE {[x]} fallen die Zeilen 1-3, 5, 7, 10-12 und 15 aus der potentiellen Ergebnismenge heraus, da in diesen Fällen die Werte der Felder x von beiden Ereignissen unterschiedlich sind. Bei den übrigen Paaren von Ereignissen bestimmt die definierte Semantik die Gültigkeit als Ergebnis:

STRICTSEQUENCE - Nur im Muster in Zeile 13 folgt einem Ereignis vom Typ A direkt ein Ereignis vom Typ B.

STRICTPARTITION - Partitionen werden durch den Wert des Feldes x im Ereignis vom Typ A bestimmt, das heißt Ereignisse vom Typ B müssen den gleichen Wert für x enthalten, damit das spezifizierte Muster erkannt wird. Bestimmt x einen Teil eines Ausführungskontext, werden auftretende Ereignisse zusätzlich geprüft: das beschriebene Muster muss im (durch x definierten) Ausführungskontext ohne Unterbrechung auftreten. Diese Prüfung ist für alle Ereignisse möglich, da angenommen wird, dass ein Ausführungskontext durch Felder, die in *jedem* möglichen Ereignistyp vorhanden sind, bestimmt wird.

Bestimmt das Feld x im Beispiel einen Ausführungskontext, so sind die Zeilen 6 und 9 keine gültigen Muster, da zwischen den dargestellten Ereignissen A und B andere Ereignisse auftreten, die den gleichen Wert x besitzen.

| | Ereignisstrom | | | | | | | | | | Semantik | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ereignis | A | A | A | C | B | A | B | C | B | B | | | | |
| Feld x | 3 | 1 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 3 | SS | SP | SN | SA |
| Nr. | | | | | | | | | | | | | | |
| 1 | A | | | | B | | | | | | - | - | - | - |
| 2 | A | | | | | | B | | | | - | - | - | - |
| 3 | A | | | | | | | | B | | - | - | - | - |
| 4 | A | | | | | | | | | B | - | X | X | X |
| 5 | | A | | | B | | | | | | - | - | - | - |
| 6 | | A | | | | | B | | | | - | (X) | X | X |
| 7 | | A | | | | | | | B | | - | - | - | X |
| 8 | | A | | | | | | | | B | - | - | - | - |
| 9 | | | A | | B | | | | | | - | (X) | X | X |
| 10 | | | A | | | | B | | | | - | - | - | - |
| 11 | | | A | | | | | | B | | - | - | - | - |
| 12 | | | A | | | | | | | B | - | - | - | - |
| 13 | | | | | | A | B | | | | X | X | X | X |
| 14 | | | | | | A | | | B | | - | - | - | X |
| 15 | | | | | | A | | | | B | - | - | - | - |
| | | | | | | | | | Anzahl | | 1 | 2 | 4 | 6 |

**Tabelle 4.2:** Beispiel: Semantik von Musterbeschreibungen

`SKIPTILLNEXT` - Bei dieser Semantik werden nicht zum Muster passende Ereignisse übersprungen. Das erste passende Ereignis wird ausgewählt. Im Beispiel ist dies für die Ereignisse in den Zeilen 4, 6, 9 und 13 der Fall.

`SKIPTILLANY` - Diese Semantik stellt den allgemeinsten Fall dar - auf die Musterbeschreibung im Beispiel passen alle Paare von Ereignissen vom Typ A gefolgt von Ereignissen vom Typ B, die die `WHERE`-Einschränkungen erfüllen. Im Beispiel sind das die Muster in den Zeilen 4, 6-7, 9 und 13-14.

Es gilt, dass alle Ergebnisse, die mit `STRICTSEQUENCE` zurückgegeben werden, auch gültig im Rahmen der `STRICTPARTITION`-Semantik sind. Ebenso gilt, dass die Ergebnisse der `SKIPTILLNEXT`-Semantik in den Ergebnissen von `SKIPTILLANY` enthalten sind.

Ganz allgemein gilt für einen gegebenen Ereignisstrom der folgende Zusammenhang bei den Ergebnismengen der unterschiedlichen Semantiken der Musterbeschreibungen:

$$STRICTSEQUENCE \subseteq STRICTPARTITION \subseteq SKIPTILLNEXT \subseteq SKIPTILLANY$$

Die Möglichkeit zur Angabe der Mustersemantik führt zu einer größeren Flexibilität beim Einsatz der Sprache: Ohne Spezifikation der Semantik muss ein Standardverhalten bei der Mustersuche definiert werden. Durch den dargestellten Zusammenhang bei den Ergebnismengen

kann nur SKIPTILLANY garantieren, dass alle möglichen Muster im Ereignisstrom erkennbar sind. Soll jetzt beispielsweise nur das Muster in Zeile 13 tatsächlich zurückgegeben werden, da es für die Mustererkennung wichtig ist, dass die Ereignisse direkt aufeinander folgen, so müssen Relationen (siehe Abschnitt 4.2.3) verwendet werden um alle anderen potentiellen Muster auszuschließen. Dies kann unter Umständen nicht möglich sein (im Beispiel sind alle Elemente der beiden einzelnen Ereignisse identisch) und erfordert dann zusätzliche Operatoren, die sich beispielsweise auf den Abstand zwischen Ereignissen beziehen.

Der Bezug auf den Abstand zweier Ereignisse erfordert jedoch Garantien bezüglich der Abbildung des beobachteten Ereignisstroms auf die Reihenfolge (und damit der Kausalitätsbeziehung) der tatsächlich aufgetretenen Ereignisse. Es könnte sonst der Fall auftreten, dass zwei Ereignisse in der spezifizierten Art auftraten, jedoch im verarbeiteten Ereignisstrom in einer nicht zum Muster passenden Reihenfolge auftreten. Dies auszuschließen erfordert einen hohen Aufwand bei der Implementierung des entsprechenden Instrumentierungssystems.

Das dargestellte Konzept der Spezifikation einer Mustersemantik ermöglicht einen Kompromiss: Wie in der Formalisierung (siehe Abschnitt 2.4) dargestellt ist eine strenge Kausalordnung der Ereignisse innerhalb eines Ausführungskontextes einfach zu realisieren - mit STRICTPARTITION kann dies auch bei der Musterspezifikation ausgenutzt werden.

Die Spezifikation der Semantik ermöglicht daher die einfachere Beschreibung bestimmter Sonderfälle bei der Beschreibung von Beziehungen zwischen Ereignissen im Ereignisstrom. Darüber hinaus kann die Ereignisstromverarbeitung mit Hilfe der Semantik optimiert werden.

### 4.2.3 Relationen

Nach der Beschreibung des Musters werden in den **Zeilen 6 bis 8** Eigenschaften definiert, die passende Ereignisse haben müssen.

```
WHERE { [ProcessId],
        [ThreadId],
        b.TimeStamp - a.TimeStamp > 2000000 }
```

Eingeleitet wird die Spezifikation von Eigenschaften mit dem Schlüsselwort WHERE. In den Zeilen 6 und 7 werden Join-Felder angegeben, in der letzten Zeile wird eine Relation definiert.

*Join-Felder* - Mit der Hilfe von Join-Felder kann beschrieben werden, dass alle Ereignisse bezüglich eines bestimmten Datenfelds die gleichen Werte enthalten sollen. Im Beispiel wird in Zeile 6 spezifiziert, dass alle in der PATTERN-Definition angegebenen Ereignisse den gleichen Wert im Feld ProcessId aufweisen sollen.

Join-Felder werden innerhalb eckiger Klammern angegeben. Jeder im Muster verwendete Ereignistyp muss ein Datenfeld mit dem entsprechenden Namen enthalten. Vorhandene Felder und Datentypen werden bei Auswertung einer EVENTS Angabe erfasst. Demnach ist sowohl eine Namens- als auch eine Typprüfung der Join-Felder möglich: in jedem Ereignistyp müssen kompatible Datentypen verwendet werden.

Im konkreten Beispiel ist ..

```
PATTERN { [syscall:a, syscallexit:b] }
WHERE { [ProcessId],
        [ThreadId],
        b.TimeStamp - a.TimeStamp > 2000000 }
```

.. eine Kurzschreibweise für ..

```
PATTERN { [syscall:a, syscallexit:b] }
WHERE { a.ProcessId == b.ProcessId,
        a.ThreadId == b.ThreadId,
        b.TimeStamp - a.TimeStamp > 2000000 }
```

Die Verwendung von Join-Feldern ermöglicht eine kompaktere Regel-Spezifikation. Join-Felder vereinfachen die Spezifikation von Relationen zwischen *allen* zu erkennenden Ereignissen, insbesondere wenn Beziehungen zwischen mehr als zwei Ereignissen definiert werden müssen.

Außerdem werden die Join-Felder für die Bestimmung passender Ereignisse bei der Muster-Semantik `STRICTPARTITION` verwendet - die Join-Felder bestimmen die Partition des Ereignisstroms. Da die Partitionierungsinformation verloren geht, sind die oben gezeigten Alternativen nur dann äquivalent, wenn die Muster-Semantik nicht `STRICTPARTITION` ist.

*Relationen* - Neben den Join-Feldern können allgemeinere Relationen definiert werden, die Einschränkungen bezüglich der Ereignisse in einem Muster definieren. Dabei kann nur auf Ereignisse Bezug genommen werden, denen bei der `PATTERN`-Spezifikation ein Name zugewiesen wurde.

Im Beispiel wird eine Relation der Zeitstempel definiert:

```
b.TimeStamp - a.TimeStamp > 2000000
```

Die Differenz der Zeitstempel des `syscallexit`-Ereignis und des `syscall`-Ereignis soll größer als `2000000` sein.

Auf Datenfelder eines Ereignisses wird mit Hilfe des '`.`'-Operators zugegriffen. Verfügbare Feldnamen und dazugehörige Felddatenypen sind durch eine `EVENTS`-Anweisung eingelesen worden. Demnach ist es möglich ein typsichere Verwendung zu überprüfen.

Als Relationsoperatoren stehen die üblichen Vergleichsoperatoren zur Verfügung: '<', '<=', '==', '>=', '>' und '`!=`'.

Für Operationen auf Ereignisfeldern stehen die folgenden arithmetischen Operatoren zur Verfügung: '`+`', '`-`', '`*`', '`/`', '`&`' und '`|`'. Die letzten beiden Operatoren symbolisieren dabei die bitweise-Und-Operation und die bitweise-Oder-Operation. Der Compiler prüft, ob die Operatoren auf die verwendeten Datentypen sinnvoll angewendet werden können.

Abschließend enthält das Beispiel-Muster in **Zeile 9** eine `RETURN`-Spezifikation.

```
RETURN { a.SyscallNr }
```

Diese gibt an, das als Resultat eines gefundenen Musters das Feld SyscallNr des Ereignisses a zurückgegeben werden soll. Weitere Rückgabewerte können mit Kommata getrennt angegeben werden. Die oben angegebenen Operatoren können auch dazu verwendet werden innerhalb einer RETURN-Anweisung Berechnungen mit Ereignisdatenfeldern durchzuführen.

Weiterhin kann mit RETURN SEQUENCE angezeigt werden, dass die Ereignisdaten aller zum Muster gehörenden Ereignisse zurückgegeben werden sollen.

Eine Anwendung, die auf erkannte Muster reagiert, kann die zurückgegebenen Information zu den Ereignissen verwenden um ihre Reaktion zu konfigurieren. Abgeleitete Werte, beispielsweise der zeitliche Abstand zwischen zwei Ereignissen, können direkt als Parameter für eine Rekonfiguration des Systems verwendet werden.

### 4.2.4 Weitere Sprachelemente

Neben den im gezeigten Beispiel verwendeten Sprachelementen existieren noch zusätzliche Konstrukte, die in Musterbeschreibungen verwendet werden können.

*Arrays von Ereignissen* - In einer Musterbeschreibung können auch Felder von Ereignissen (des gleichen Typs) spezifiziert werden. Felder können ebenfalls benannt und außerdem in ihrer Größe beschränkt werden.

Es gibt folgende Möglichkeiten die Anzahl der Feldelemente zu beschränken:

- A[] - spezifiziert eine unbeschränkte Anzahl von Ereignissen vom Typ A
- A[<x] - das Feld soll weniger als x Elemente enthalten
- A[=x] - das Feld soll genau x Elemente enthalten
- A[>x] - das Feld soll mehr als x Elemente enthalten
- A[x..y] - die Anzahl $a$ der Feldelemente soll $x < a < y$ sein.

Ein Name kann einem Feld nach dem folgenden Schema zugewiesen werden: A[]:b. Das (unbeschränkte) Feld mit Ereignissen vom Typ A erhält den Namen b.

In Relationen können Funktionen verwendet werden, die alle bereits gelesenen Elemente eines Feldes verarbeiten. Sei a der Name eines in der Musterbeschreibung definierten Feldes, so kann mit

- a.len auf die Anzahl der Elemente,
- a.min.<Feldname> auf den kleinsten Wert des Elements Feldname,
- a.max.<Feldname> auf den größten Wert des Elements Feldname,
- a.avg.<Feldname> auf den Durchschnitt der Werte der Elemente Feldname,

zugegriffen werden. Die abgeleiteten Eigenschaften von Feldern können auch in Relationen und Rückgabewerten verwendet werden. Dabei hat der Rückgabewert der Funktionen min und max den gleichen Datentyp wie das entsprechende Feld. len liefert eine Ganzzahl und avg eine Gleitkommazahl. Der Compiler kann so auch die Verwendung von Feldfunktionen auf Typsicherheit hin überprüfen.

Während der Ereignisstromverarbeitung werden Felder in Musterbeschreibungen *sofort wenn möglich* abgeschlossen. Ein (künstliches) Beispiel:

```
[a[>2],a,a[<4],b]
```

Es soll eine Folge von Ereignissen vom Typ `a` erkannt werden, zuerst eine Folge von mehr als zwei Ereignissen, anschließend ein einzelnes Ereignis, danach eine Folge von weniger als vier Ereignissen und abschließend ein einzelnes Ereignis vom Typ `b`.

Bei der Verarbeitung eines Ereignisstroms wird zuerst versucht die Bedingungen für das erste Feld von `a`-Ereignissen zu erfüllen: drei Ereignisse werden diesem Feld zugeordnet. Danach ist die Bedingung erfüllt. Ein viertes `a`-Ereignis würde dem einzelnen Ereignis im Muster zugeordnet werden. Danach wird versucht das zweite Feld mit `a`-Ereignissen zu füllen: tritt ein Ereignis `b` auf solange weniger als vier `a`-Ereignisse verarbeitet wurden, ist das Muster erfolgreich erkannt worden. Treten vier `a`-Ereignisse auf ist das Muster nicht im Ereignisstrom vorhanden.

Das dargestellte Vorgehen ermöglicht die Umsetzung der Musterbeschreibungen in einen deterministischen Automaten und eine effiziente Mustererkennung.

*Spezifikation einer Zeitschranke* - Mit dem Schlüsselwort `WITHIN` kann eine Zeitschranke für das gesamte Muster spezifiziert werden. Das heißt, dass das komplette Muster innerhalb des definierten Zeitraums erkannt werden muss.

Im Beispiel könnte die definierte Relation

```
b.TimeStamp - a.TimeStamp > 2000000
```

auch durch `WITHIN 2000000` ersetzt werden - die Semantik ist die gleiche. Allerdings hat die Variante mit `WITHIN` einen Vorteil: die Laufzeitumgebung hat die Möglichkeit jederzeit die Zeitstempel aktueller Ereignisse mit dem Zeitstempel des ersten erkannten Ereignis zu vergleichen. Bei Überschreitung der Zeitschranke kann die Mustererkennung abgebrochen werden. Die Relation hingegen kann erst ausgewertet werden, wenn auch das zweite Ereignis erkannt wurde.

Ergänzend zu den hier vorgestellten Sprachmitteln wird in Abschnitt 4.5.1 eine mögliche Spracherweiterung beschrieben: mit Hilfe einer einfachen Skriptsprache können Aktionen definiert werden die ausgeführt werden, sobald ein Muster im Ereignisstrom erkannt wurde.

## 4.3 Automaten zur Mustererkennung

Für eine effiziente Mustererkennung werden Regeln, die mit Hilfe der im Abschnitt 4.2 vorgestellten Sprache definiert worden sind, in deterministische, endliche Automaten (DEA) umgewandelt. In diesem Abschnitt werden zunächst Grundlagen zu DEA und der Umwandlung der Muster beschrieben. Anschließend wird der implementierte Compiler vorgestellt.

### 4.3.1 Grundlagen

Grundsätzlich ist ein DEA durch fünf Elemente $(\Sigma, S, s_0, \delta, F)$ gekennzeichnet:

- $\Sigma$ ist die durch den Automaten verarbeitete Eingabesprache - eine endliche, nicht leere Menge von Symbolen. Im Kontext der Mustererkennung in Ereignisströmen ist $\Sigma$ gleich der Menge der auftretenden Ereignistypen.

- $S$ ist eine endliche, nicht leere Menge von Zuständen des Automaten. Diese Zustandsmenge wird im vorliegenden Fall vom Compiler aus der Regelbeschreibung abgeleitet.
- $s_0$ ist der Startzustand des Automaten, mit $s_0 \in S$.
- $\delta$ ist die Zustandsübergangsfunktion, die aus dem aktuellen Zustand des Automaten und einem Eingabeelement den nächsten Zustand bestimmt, also $\delta : S \times \Sigma \rightarrow S$. Diese Funktion wird ebenfalls vom Compiler aus der Regelbeschreibung abgeleitet und ist durch die Struktur (also Zuständen und Transitionen) des Automaten gegeben.
- $F$ ist die Menge der Endzustände des Automaten. In diesen Zuständen ist ein Muster entweder vollständig erkannt worden (Zustand $r$) oder es kann ausgeschlossen werden, dass es im aktuellen Zustand noch auftreten wird (Zustand $i$). Es gilt: $F = \{r, i\}$ und $F \subset S$.

Aus der definierten Musterbeschreibungssprache und der Möglichkeit zur Definition von Bedingungen ergibt sich, dass die Zustandsübergangsfunktion im vorliegenden Fall komplexer gestaltet werden muss: der nächste Automatenzustand ergibt sich aus dem aktuellen Zustand, dem Eingabeelement (also dem aufgetretenen Ereignis) und zusätzlich aus Informationen, die aus den bisher verarbeiteten Ereignissen entnommen wurden. Die Zustandsübergangsfunktion ist also *zustandsbehaftet* - ähnlich zur Zustandsübergangsfunktion eines Mealy-Automaten, allerdings mit leerer Ausgabemenge.

Zur Integration der Zustandsbehandlung in das DEA-Modell wird das Konzept des *Laufzeitzustands* eingeführt. Zu jedem Automaten ist eine endliche Menge $Z$ von Zustandsinformationen $\{z_0..z_n\}$ gespeichert.

Die Zustandsübergangsfunktion $\delta$ bestimmt dann in Abhängigkeit vom aktuellen Zustand $s$, dem Eingabelement $\sigma$ und dem Laufzeitzustand $Z$ den neuen Automatenzustand $s'$ und den neuen Laufzeitzustand $Z'$. Die Funktion $\delta$ wird durch die folgenden beiden Teilfunktionen $\delta_S$ und $\delta_Z$ gebildet:

- Die Funktion $\delta_S$ bestimmt den Folgezustand im Automaten: $s' = \delta_S(s, \sigma, Z)$.

  In der Funktion $\delta_S$ werden eine Reihe von Bedingungen ausgewertet, die bei einem Zustandsübergang erfüllt sein müssen. Dazu können beispielsweise Datenfelder von $\sigma$ mit Elementen aus $Z$ verglichen werden.

  Die Funktion muss für alle Eingabeparameter eindeutig definiert sein, damit der resultierende Automat weiterhin deterministisch ist.
- Die Funktion $\delta_Z$ aktualisiert den Laufzeitzustand: $Z' = \delta_Z(s, s', \sigma, Z)$, beziehungsweise $Z' = \delta_Z(s, \delta_S(s, \sigma, Z), \sigma, Z)$.

  Die Funktion $\delta_Z$ aktualisiert den Laufzeitzustand in Abhängigkeit des aktuellen Eingabeelements sowie des tatsächlich auszuführenden Automatenzustandübergangs.

  Aktionen, die bei der Durchführung des Übergangs durch die Funktion $\delta_Z$ ausgeführt werden können, sind beispielsweise das Kopieren von Datenfeldern des aktuellen Ereignis in den Laufzeitzustand oder die Aktualisierung eines Durchschnittswertes für ein Array.

Regelbeschreibungen werden mit Hilfe eines Compilers in einen DEA nach dem beschriebenen Modell umgewandelt. Die theoretischen Grundlagen dieser Umwandlung werden im Abschnitt 4.3.2 dargestellt. Die Umsetzung dieser Grundlagen in einen Compiler wird im Abschnitt 4.3.3 beschrieben.

### 4.3.2 Automatenerzeugung

Bei der Umwandlung einer gegebenen Regelbeschreibung in einen DEA sind drei verschiedene Aspekte zu beachten: (1) Erkennung der Musterstruktur, also einer gültigen Folge von Ereignissen im Ereignisstrom, (2) Berücksichtigung der Mustersemantik, und (3) Auswertung von Bedingungen, die von einer gültigen Ereignisfolge erfüllt sein müssen.

#### Musterstrukturerkennung

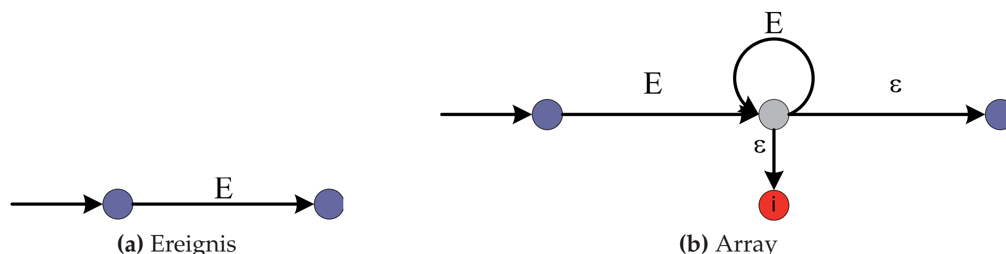
Nach dem Schlüsselwort `PATTERN` folgt die Spezifikation des zu erkennenden Musters im Ereignisstrom. Betrachtet man ausschließlich die Ereignistypen ohne eventuell zugewiesene Namen, erkennt man die Struktur des Musters. Nachfolgend wird dargestellt, wie aus der Musterstrukturbeschreibung ein DEA abgeleitet werden kann, der entsprechende Strukturen erkennt.

Die vorgestellte Musterspezifikationssprache ähnelt regulären Ausdrücken zur Verarbeitung von Zeichenketten. Das Eingabealphabet wird dabei durch die verwendeten Ereignistypen gebildet. Die Umwandlung von regulären Ausdrücken in deterministische Automaten ist ein bekanntes und gut verstandenes Gebiet des Compilerbau [22]. Nachfolgend werden die grundlegenden Schritte der Umwandlung kurz zusammengefasst. Besonderheiten ergeben sich bei der Behandlung von Feldern und Negationen.

Zur Umwandlung wird eine angepasste Variante des McNaughton-Yamada-Thompson Algorithmus verwendet. Prinzipiell wird der Automat dabei *top-down* mit Hilfe einer Menge von Konstruktionselementen aufgebaut.

Ein Konstruktionselement enthält eine Menge von Zuständen in der mindestens zwei Elemente enthalten sind: ein Start- und ein Endzustand. Ein mit einem `i` dargestellter Zustand ist der Invalid-Zustand. Dieser wird erreicht, wenn das spezifizierte Muster garantiert nicht erkannt werden kann. Weiterhin definiert ein Konstruktionselement basierend auf dem konkret beschriebenen Operator mögliche Übergänge zwischen diesen Zuständen.

Für Zustandsübergänge existieren zwei Varianten: Zum Einen kann ein Übergang mit einem konkreten Ereignistypen assoziiert sein, das heißt ein Ereignis dieses Typs muss im Ereignisstrom auftreten, damit der Zustandsübergang durchgeführt werden kann. Zum Anderen kann ein Übergang mit $\epsilon$ beschriftet sein. In diesem Fall muss für einen Übergang kein Ereignis gelesen werden. Beide Typen von Übergängen können zusätzlich mit Bedingungen verbunden werden - diese müssen erfüllt sein, damit der Übergang durchgeführt werden kann. Entsprechende Bedingungen werden bei der Diskussion der Konstruktionselemente erläutert.




**(a)** Ereignis **(b)** Array

**Abbildung 4.1:** Automatenkonstruktion: Grundelemente

Die Grundelemente zur Automatenkonstruktion - Ereignis und Array - sind in Abbildung 4.1 dargestellt.

| Feldspezifikation | Bedingung $c1$ | Bedingung $c2$ | Bedingung $c3$ |
|---|---|---|---|
| A[] | $E$ | $\neg E$ | _ |
| A[<x] | $E \wedge (l < x - 1)$ | $\neg E \wedge (l < x)$ | $E \wedge (l = x - 1)$ |
| A[=x] | $E \wedge (l < x)$ | $\neg E \wedge (l = x)$ | $E \wedge (l = x)$ |
| A[>x] | $E$ | $\neg E \wedge (l > x)$ | _ |
| A[x..y] | $E \wedge (l < y - 1)$ | $\neg E \wedge (l > x) \wedge (l < y)$ | $E \wedge (l = y - 1)$ |

**Tabelle 4.3:** Umsetzung von Feldlängenbeschränkungen im Automaten

Das Konstruktionselement für ein elementares Ereignis $E$ (Abbildung 4.1a) führt zu einem direkten Übergang vom Start- zum Endzustand. Die entsprechende Kante ist mit $E$ annotiert, als Zeichen dafür, dass dieser Übergang ausgeführt werden kann, wenn ein entsprechendes Ereignis auftritt und die Automatenverarbeitung sich im Startzustand befindet.

In Abbildung 4.1b ist das Konstruktionselement für Arrays dargestellt. Felder von Ereignissen enthalten mindestens ein Element, dementsprechend existiert ein Übergang vom Startzustand zum mittleren Zustand. Jetzt können beliebig viele weitere Ereignisse gelesen werden. Der mittlere Zustand kann über zwei $\epsilon$-Übergänge verlassen werden. Die Bedingung für diese Übergänge ist abhängig von eventuell vorhandenen Längenbegrenzungen für das zu erkennende Feld - so dass entweder eine gültige Menge von Ereignissen erkannt wurde (Übergang zum Endzustand) oder die Längenbedingung nicht erfüllt werden kann (Übergang zum Invalid-Zustand).

Alle drei vom mittleren Zustand ausgehenden Übergänge sind mit zusätzlichen Bedingungen verknüpft. Nachfolgend sei $c1$ die Bedingung für den Übergang vom mittleren Zustand zum mittleren Zustand, $c2$ die Bedingung für den Übergang vom mittleren Zustand zum Endzustand und $c3$ die Bedingung vom mittleren Zustand zum Invalid-Zustand.

Die konkreten Bedingungen $c1$, $c2$ und $c3$ für die möglichen Längenbeschränkungen von Feldern sind in Tabelle 4.3 dargestellt. Die Teilbedingung $E$ ist erfüllt, wenn das aktuell gelesene Ereignis zu dem Typ des Feldes passt. Weiterhin sei $l$ die aktuelle Anzahl von Elementen des zu lesenden Feldes. Nach dem initialen Übergang in den mittleren Zustand gilt $l = 1$; mit jedem weiteren gelesenen Arrayelement wird $l$ erhöht ($l = l + 1$). Die Variable $l$ ist Bestandteil des Laufzeitzustandes $Z$ des Automaten.

Ist die Feldlänge nicht beschränkt (`A[]`), können der Invalid-Zustand und der entsprechende Zustandsübergang aus dem generierten Teilautomaten entfernt werden, da sie nicht benötigt werden.

Die Annotation von Zustandsübergängen mit beliebigen Bedingungen kann zu nicht deterministischen Verhalten des generierten Automaten führen, wenn die Bedingungen für mehrere Übergänge gleichzeitig erfüllt sind. Bei der dargestellten Verarbeitung von Arrays ist dies nicht der Fall, wie nachfolgend belegt wird.

Bei Spezifikation einer Beschränkung der Feldlänge gilt das folgende Theorem:

**Theorem** *Für alle möglichen Kombinationen aus Werten für die Feldlänge $l$ und dem Ergebnis des Prädikats $E$ kann höchstens eine der Bedingungen $c1$, $c2$ oder $c3$ zu $true$ ausgewertet werden.*

**Beweis** *Für ein unbeschränktes Feld (`A[]` und `A[>x]`) ist die Aussage erfüllt, da $E \wedge \neg E = false$ beziehungsweise $E \wedge \neg E \wedge (l > x) = false$. In jedem der weiteren drei Fälle gibt es genau zwei*

*Bedingungen, die $E$ in der gleichen Form enthalten, also entweder negiert oder nicht negiert. Beide Bedingungen besitzen daher entweder die Form „$\neg E \wedge ...$“ oder „$E \wedge ...$“. Die dritte Bedingung enthält dann $E$ in der jeweils umgekehrten Form.*

*Da es sich bei allen Bedingungen um Konjuktionen handelt, können Bedingungen, die $E$ und $\neg E$ enthalten, nicht gleichzeitig zu* `true` *ausgewertet werden. Es bleibt also noch zu zeigen, dass die Bedingungen, die $E$ in der gleichen Form verwenden, nicht gleichzeitig zu* `true` *ausgewertet werden können.*

*Diese Eigenschaft ist erfüllt:*

`A[<x]`*:*
$c1 \wedge c3 = E \wedge (l < x - 1) \wedge E \wedge (l = x - 1) = E \wedge (l < x - 1) \wedge (l = x - 1) = false$

`A[=x]`*:*
$c1 \wedge c3 = E \wedge (l < x) \wedge E \wedge (l = x) = E \wedge (l < x) \wedge (l = x) = false$

`A[x..y]`*:*
$c1 \wedge c3 = E \wedge (l < y - 1) \wedge E \wedge (l = y - 1) = E \wedge (l < y - 1) \wedge (l = y - 1) = false$

Die gezeigte Eigenschaft der Bedingungen $c1$, $c2$ und $c3$ ist notwendig um einen deterministischen Teil-Automaten zu konstruieren: es ist keiner, oder genau einer der vorhanden Zustandsübergänge möglich.

In den gezeigten Fällen sind genau dann *keine* Zustandsübergänge definiert, wenn ein Ereignis mit einem anderen Typ als durch das Feld vorgegeben auftritt und der Zustand der Felderkennung nicht verlassen werden kann, da keine zulässige Anzahl von Elementen im Feld gespeichert ist. Die Mustersemantik definiert das Verhalten in solchen Situationen: je nach geforderter Semantik kann beispielsweise ein nicht zum Feld passendes Ereignis übersprungen werden. Durch das Einfügen von Default-Übergängen, die ausgeführt werden wenn keine anderen Übergänge gefunden werden, ist eine eindeutige Wahl der Kanten für jeden Zustand möglich. Eine genauere Beschreibung der Integration der Mustersemantik erfolgt weiter unten.

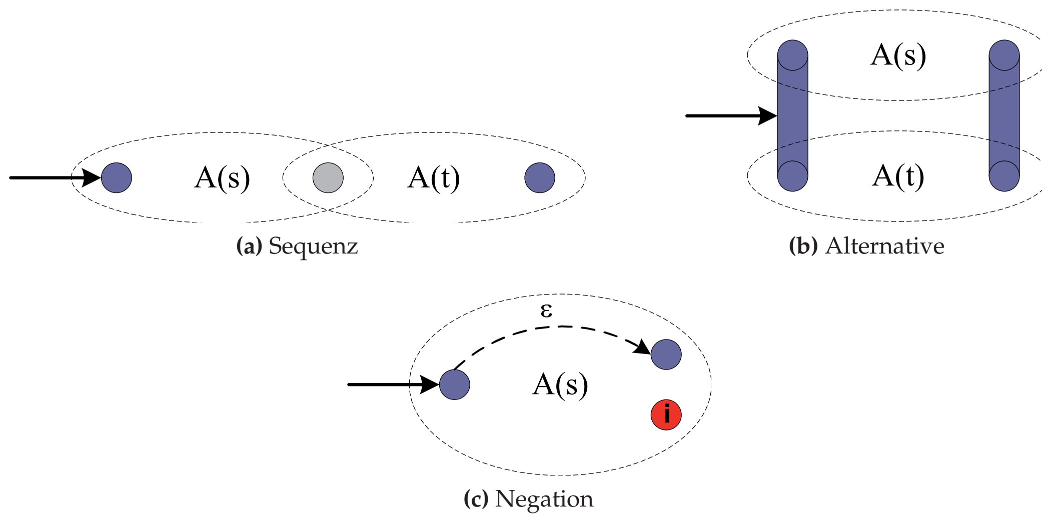
In regulären Ausdrücken werden Wiederholungen von beliebigen Sub-Mustern üblicherweise mit dem Kleene-Operator realisiert. Der Kleene-Operator ist in der prototypischen Implementierung nicht vollständig unterstützt und wird, wie dargestellt, auf Wiederholungen von einfachen Ereignissen reduziert. Dies ist für die durchgeführten Fallstudien ausreichend und ermöglicht vollständige Analysen der entstehenden Automaten, da diese entsprechend ihrer Konstruktion keine (beziehungsweise nur einfache) Zyklen enthalten können.

Die Einschränkungen des Kleene-Operator beziehen sich dabei nur auf die Sprachspezifikation und den Compiler - die Laufzeitumgebung verarbeitet generierte Automaten in ihrer übersetzten Form, die auch beliebige Automaten erlaubt. Zur vollständigen Unterstützung von sich wiederholenden Sub-Mustern ist demnach nur eine Erweiterung des Compilers notwendig.

Die beiden bisher gezeigten elementaren Konstruktionselemente können mit Hilfe von drei Operatoren zu komplexeren Automaten zusammengesetzt werden. Diese Operatoren werden nachfolgend beschrieben.

Die Konstruktionselemente der unterstützten Operatoren - Sequenz, Alternative und Negation - sind in Abbildung 4.2 dargestellt.

Die Operatoren Sequenz und Alternative sind in naheliegender Weise definiert. Bei einer Sequenz vom Sub-Muster $s$ gefolgt vom Sub-Muster $t$ werden beide Sub-Automaten durch ihre Start- und Endzustände verbunden - der dargestellte mittlere Automatenzustand ist also der Endzustand von $A(s)$ und gleichzeitig der Startzustand von $A(t)$. Bei einer Alternative stellt




**(a)** Sequenz

**(b)** Alternative

**(c)** Negation

**Abbildung 4.2:** Automatenkonstruktion: Operatoren

jedes Sub-Muster einen alternativen Weg vom Start- zum Endzustand dar. Dementsprechend kann vom Startzustand ausgehend jeder alternative Sub-Automat ausgewählt und so der Endzustand erreicht werden.

Die Negation eines Sub-Musters bedeutet, dass das negierte Muster *nicht* im Ereignisstrom auftreten darf, damit das Gesamtmuster erkannt wird. Insbesondere wird die Negation demnach nicht im Sinne eines Platzhalters definiert - das Muster `[a,~b,c]` wird durch ein Ereignis vom Typ `a` direkt gefolgt von einem Ereignis vom Typ `c` erfüllt - die Bedeutung ist nicht „ein Ereignis vom Typ `a`, gefolgt von einem beliebigen aber von Typ `b` verschiedenen Ereignis, gefolgt von einem Ereignis von Typ `c`“.

Die Negation in der hier beschriebenen Weise zu betrachten führt dazu, dass Muster direkt und nur mit bekannten Ereignistypen beschrieben werden müssen - Platzhalter, die potentiell auch unbekannte Ereignistypen repräsentieren können, stehen diesem Prinzip entgegen. Das Verhalten der Mustererkennung bei der Verarbeitung unbekannter Ereignistypen wird durch die Mustersemantik bestimmt. Unbekannte Ereignisse werden dann beispielsweise ignoriert, oder führen zum Abbruch der Mustererkennung.

Die fehlende Platzhalter-Semantik führt, unter der Voraussetzung, dass nur bekannte Ereignistypen in einer Musterbeschreibung zulässig sind, nicht zu einer Einschränkung der Beschreibungsmächtigkeit der Musterspezifikationssprache. Sei $M = \{a, b, c, d\}$ die Menge der Ereignisstypen, kann das Muster `[a,~b,c]` wie folgt umgewandelt werden: `[a,(a|c|d),c]`. Die umgewandelte Form beschreibt dann das Muster so, als ob der Negationsoperator die Platzhalter-Semantik aufweisen würde.

Zur Konstruktion eines Automaten für eine Negation wird das in Abbildung 4.2c dargestellte Schema verwendet - es ist folgendermaßen zu verstehen: der Sub-Automat $A(s)$ stellt eine Verbindung zwischen dem Startzustand und dem Invalid-Zustand her. Alle entstehenden Zwischenzustände erhalten einen $\epsilon$-Übergang zum dargestellten Endzustand. Weitere Verknüpfungen, beispielsweise im Rahmen einer Sequenz oder einer Alternative, werden über den Endzustand hergestellt.

Ein negiertes Sub-Muster darf in seiner vollständigen Form nicht im Ereignisstrom auftreten - das Einfügen von $\epsilon$-Übergängen von internen Zuständen zum Endzustand stellt sicher, dass

die Erkennung von negierten Sub-Mustern an jeder Stelle abgebrochen werden kann. Mit den $\epsilon$-Übergängen sind keine weiteren Bedingungen verbunden.

Mehrfache Negationen werden vom im Rahmen dieser Arbeit erstellten Compiler nicht unterstützt und entsprechende Musterspezifikationen als Semantik-Fehler zurückgewiesen. Dieses Verhalten stellte weder für die durchgeführten Fallstudien eine Einschränkung dar, noch konnten sinnvolle Anwendungsfälle für mehrfache Negationen identifiziert werden. Das Gleiche gilt auch für die Negation von Feldern.

Der entsprechend den beschriebenen Konstruktionsschritten entstandene Automat mit $\epsilon$-Übergängen wird im nächsten Schritt in einen Automaten ohne $\epsilon$-Übergängen konvertiert. Abbildung 4.3 zeigt den Automaten für das Muster `[(A|[B[],~[C,D]]),B[<5]]`. Ausgangspunkt bildet der in der Abbildung oben dargestellte Automat mit $\epsilon$-Übergängen.

Nacheinander werden jetzt die drei unterschiedlichen Typen von $\epsilon$-Übergängen bearbeitet:

**Übergänge zum Invalid-Zustand bei der Erkennung von Feldern:** Die in Tabelle 4.3 dargestellten Bedingungen $c3$ zeigen, dass entsprechende Übergänge in jedem Fall einen Bezug zum aktuell erkannten Ereignistyp $E$ besitzen. Demnach können diese $\epsilon$-Übergänge durch reguläre Übergänge ersetzt werden, die den Ereignistyp sowie weitere Bedingungen prüfen.

**Übergänge zum folgenden Zustand beim Abbruch von Negationen:** Der Zielzustand solcher $\epsilon$-Übergänge ist der Fortsetzungspunkt einer abgebrochenen Negation. In Abbildung 4.3 bildet Zustand 3 einen solchen Fortsetzungspunkt: nachdem das erste Element der negierten Sequenz, ein Ereignis vom Typ $C$, erkannt wird, kann die Erkennung der Negation abgebrochen werden, wenn danach ein Ereignis vom Typ $B$ auftritt - an Stelle von Zustand 4 befindet sich der Automat bei Abbruch der Negation virtuell in Zustand 3.

Die $\epsilon$-Übergänge werden daher nach folgendem Verfahren entfernt: alle ausgehenden Übergänge vom Fortsetzungspunkt der Negation werden (mit den gleichen Zielzuständen) auch an die Startzustände der $\epsilon$-Übergänge angefügt. Sollte der Fortsetzungspunkt anschließend keine weiteren eingehenden Zustandsübergänge haben, kann er entfernt werden.

**Übergänge zum folgenden Zustand bei der Erkennung von Feldern:** Auch in diesen Fällen werden die ausgehenden Übergänge des Zielzustands des $\epsilon$-Übergangs mit dem Startzustand des $\epsilon$-Übergangs verknüpft. Allerdings müssen eventuell vorhandene Bedingungen des $\epsilon$-Übergangs mit den Bedingungen der ausgehenden Übergänge verbunden werden.

Eine Musterspezifikation wird mit den beschriebenen Schritten direkt in einen deterministischen Automaten überführt.

Nachdem die beschriebenen Schritte für eine gegebene Regel durchgeführt worden sind, entsteht als Zwischenergebnis ein DEA zur Erkennung der Musterstruktur. Dieser DEA berücksichtigt allerdings noch keine unbekannten Ereignistypen, keine Mustersemantik und keine der definierten Bedingungen.

#### Integration der Mustersemantik

Im nächsten Schritt der DEA-Erzeugung wird die definierte Semantik der Musterbeschreibung in den bis hierher generierten Automaten integriert. Jeder Zustandsübergang im erzeugten

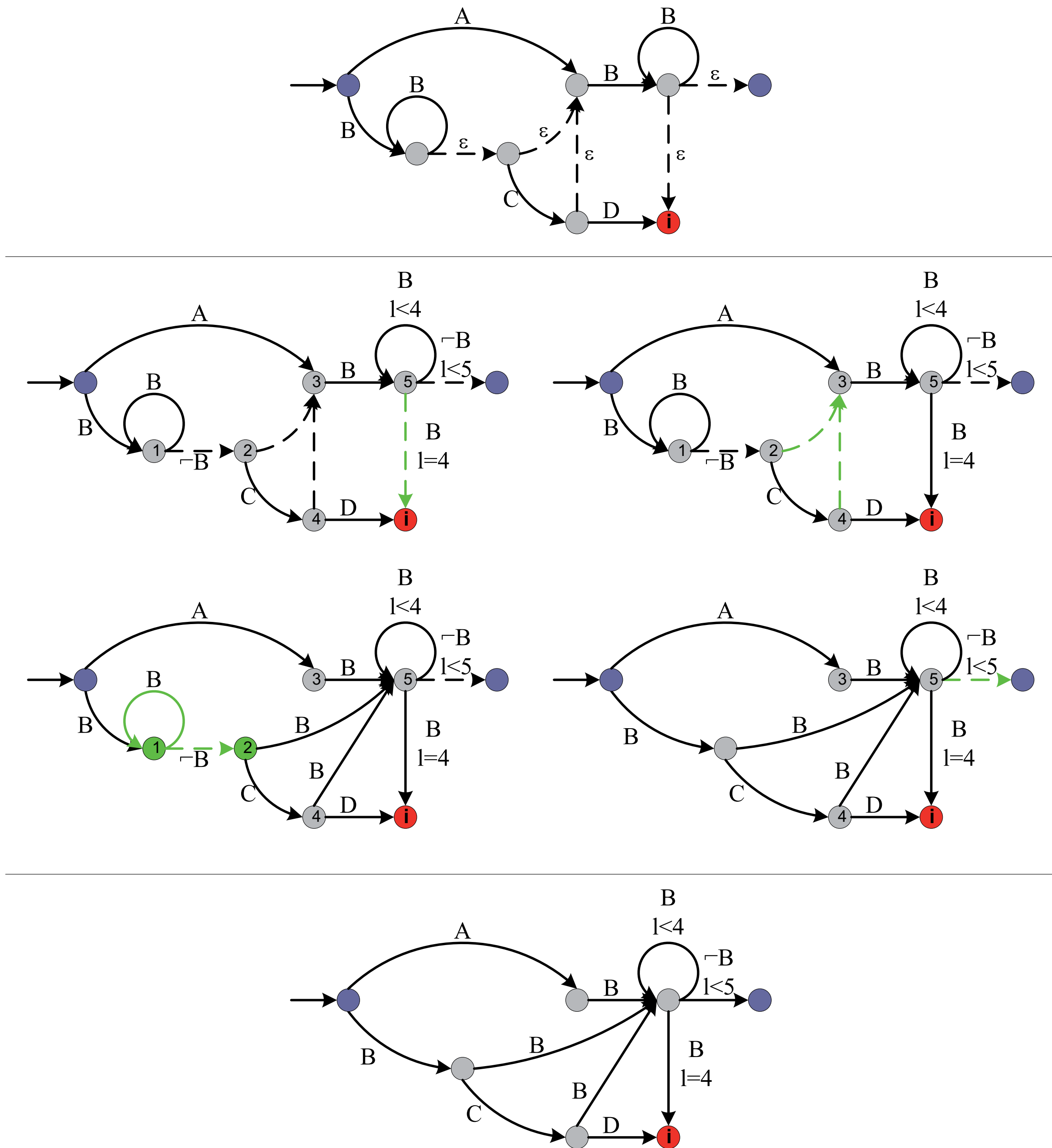



**Abbildung 4.3:** Beispiel: Automatenkonstruktion

DEA ist mit dem Auftreten eines konkreten Ereignistyps verbunden. Die Semantik definiert jetzt, welcher Übergang ausgeführt wird, wenn ein Ereignis auftritt, für das kein Zustandsübergang definiert ist.

`STRICTSEQUENCE`-Semantik: Die Ereigniskette, die auf das zu erkennende Muster zutrifft, soll lückenlos und direkt nacheinander im Ereignisstrom vorkommen. Daher gilt: werden Ereignisse im Ereignisstrom erkannt, die nicht von dem Musterstruktur-Automaten akzeptiert werden, liegt das beschriebene Muster nicht vor.

Bei der Integration dieser Semantik wird daher von jedem internen Zustand des Automaten eine Transition zum Invalid-Zustand eingefügt, die immer dann ausgeführt wird, wenn die Bedingungen für keine der bereits definierte Transitionen erfüllt sind.

`STRICTPARTITION`-Semantik: Bei dieser Semantik können Ereignisse ignoriert werden, die nicht zu gleichen Partition gehören. Die Ereignisse in einem zu erkennenden Muster sollen innerhalb der definierten Partition direkt aufeinander folgen.

Bei der Integration dieser Semantik in den bereits erstellten Struktur-Automaten werden jetzt für jeden internen Zustand $s_i$ zwei Transitionen hinzugefügt: (1) eine Transition von $s_i$ nach $s_i$, die unter der Bedingung ausgeführt wird, dass das gelesene Ereignis nicht zur definierten Partition gehört und (2) eine Transition von $s_i$ zum Invalid-Zustand, die ausgeführt wird, wenn keine andere gültige Transition gefunden wurde.

Durch die erste Transition verbleibt der Automat also in seinem aktuellen Zustand und ignoriert das gelesene Ereignis. Die zweite Transition führt zum Abbruch der Mustererkennung, wenn ein zur Partition gehörendes Ereignis gelesen wird, das jedoch nicht zur Struktur des Musters passt.

`SKIPTILLNEXT`-Semantik: In dieser Semantik ist keine Aussage bezüglich des Abstandes zwischen zwei zum Muster gehörenden Ereignissen möglich. Daher kann die Mustererkennung nicht abgebrochen werden, wenn ein nicht zum Muster passendes Ereignis gelesen wird.

Die Integration dieser Semantik besteht also darin, dass zu jedem internen Zustand eine Transition hinzugefügt wird, die den Automaten in diesem Zustand belässt. Diese Transition wird immer dann ausgeführt, wenn keine andere gültige Transition gefunden wird.

`SKIPTILLANY`-Semantik: Diese Semantik gleicht der `SKIPTILLNEXT`-Semantik - es sind ebenfalls keine Aussagen über den Abstand zweier Ereignisse möglich und die Mustererkennung kann nicht aufgrund eines nicht passenden Ereignis abgebrochen werden.

Um die Integration dieser Semantik zu beschreiben, muss auf einen Aspekt des Abarbeitungsmodells verwiesen werden: Jede Mustererkennung wird durch einen Laufzeitzustand repräsentiert. Dieser Zustand enthält beispielsweise Informationen über Felder bereits gelesener Ereignisse oder den aktuellen Zustand im Automaten.

Zur Integration der `SKIPTILLANY`-Semantik werden *Replikations-Transitionen* eingefügt. Wird der Zustandsübergang im Automaten durch eine Replikations-Transition beschrieben, so werden die folgenden Aktionen durchgeführt: (1) der aktuelle Laufzeitzustand wird kopiert, (2) die mit der Transition verbundenen Aktionen werden bezüglich des kopierten Zustandes durchgeführt, und (3) der ursprüngliche Laufzeitzustand verbleibt im ursprünglichen Automatenzustand.

Mit Hilfe dieses Konzepts kann die `SKIPTILLANY`-Semantik realisiert werden, indem alle Zustandsübergänge als Replikations-Transitionen durchgeführt werden.

Nach diesen Schritten ist die Mustersemantik in den DEA integriert. Jetzt berücksichtigt der DEA in jedem Zustand jedes mögliche auftretende Ereignis und verhält sich entsprechend der Semantik. Zum Abschluss müssen jetzt noch die in der Musterspezifikation definierten Bedingungen integriert werden.

### Auswertung von Bedingungen

Mit Hilfe von `WHERE`-Klauseln können Bedingungen definiert werden, die eine zum zu erkennenden Muster passende Ereignisfolge erfüllen muss. Bei der Integration dieser Bedingungen

in den Automaten sind drei Aspekte zu beachten: (1) die Benennung der Ereignisse, (2) die Daten der verwendeten Ereignisfelder, und (3) die tatsächliche Prüfung der Bedingungen.

In einer einfachen Implementierung könnte der dritte Schritt erst im $r$-Zustand, also nach vollständiger Erkennung der Musterstruktur erfolgen - sind nicht alle Bedingungen erfüllt, wird das Ergebnis des Automaten nicht zurückgegeben. Es ist sofort ersichtlich, dass diese Implementierungsstrategie ineffizient ist, da sie zu einer Vielzahl von Automaten im $r$-Zustand führen kann, die dennoch keine gültigen Muster im Ereignisstrom repräsentieren.

Mit dem nachfolgend beschriebenen Vorgehen wird daher der frühest mögliche Zustandsübergang im Automaten gesucht, bei dem eine Bedingung ausgewertet werden kann. Ist die entsprechende Bedingung nicht erfüllt, geht der Automat sofort in den $i$-Zustand über.

Eine Bedingung kann dann ausgewertet werden, wenn alle benötigten Informationen fest definiert sind. Dabei kann sich eine Bedingung auf zwei verschiedene Klassen von Informationen beziehen:

**Ereignis:** Informationen, die in einem Ereignis gespeichert sind - beispielsweise Werte eines bestimmten Feldes - sind fest definiert, wenn das entsprechende Ereignis gelesen wurde.

**Laufzeitzustand:** Informationen, die im Laufzeitzustand gespeichert werden - beispielsweise die Anzahl der Elemente in einem Feld - sind fest definiert, wenn ihre zugrunde liegende Musterkomponente vollständig eingelesen wurde.

Für beide Klassen von Informationen gilt, dass der Zeitpunkt ihrer festen Definition einem Zustandsübergang im Automaten entspricht. Ein Ereignis wird dadurch *gelesen*, also als zur gesuchten Konstellation passend akzeptiert, dass der Automat in einen neuen Zustand übergeht. Informationen im Laufzeitzustand zu einer Musterkomponente können sich nicht mehr ändern, wenn die entsprechende Komponente vollständig erkannt wurde.

Beim Einlesen von Bedingungen werden diese in ihre benötigten Informationskomponenten zerlegt. Die Integration in den Automatengraph erfolgt dann folgendermaßen:

Ausgehend vom Start-Zustand werden die Zustandsübergänge analysiert und zu jedem Zustand die Menge der fest definierten Informationskomponenten bestimmt. Bei einem Übergang von Zustand $A$ zu Zustand $B$ gilt, dass alle Komponenten, die bereits im Zustand $A$ fest definiert sind, auch in Zustand $B$ fest definiert sind. Dazu kommen dann noch jene Komponenten, die durch den Zustandsübergang definiert werden. Als Ausgangspunkt sind im Start-Zustand noch keine Informationskomponenten fest definiert - schrittweise können anschließend die Definitionsmengen der einzelnen Zustände bestimmt werden.

Eine spezifizierte Bedingung wird jetzt in den Zustandsübergängen zwischen den Zuständen $A$ und $B$ eingefügt, für die gilt: alle Informationskomponenten der Bedingungen sind in Zustand $B$ fest definiert - in Zustand $A$ hingegen nicht. Ein solcher Zustandsübergang bestimmt den frühest möglichen Zeitpunkt für die Prüfung der Bedingung.

In der Implementierung wird für die Bedingungsprüfung ein neuer Zustandstyp eingeführt: Ein *Entscheidungszustand* hat die Eigenschaft, dass sofort wenn der Automat einen solchen Zustand erreicht, der nächste Zustandsübergang durchgeführt wird. Das heißt insbesondere, dass nicht auf das nächste gelesene Ereignis gewartet wird. Mit diesem Konzept lässt sich die Integration von Bedingungen nach dem in Abbildung 4.4 dargestellten Schema durchführen.

Eine Bedingung $B$ soll in den mit $E$ gekennzeichneten Zustandsübergang integriert werden. Dazu wird ein Entscheidungszustand zwischen Start- und End-Zustand des Übergangs eingefügt. Alle Bedingungen und Aktionen, die mit dem Übergang $E$ verbunden sind, werden

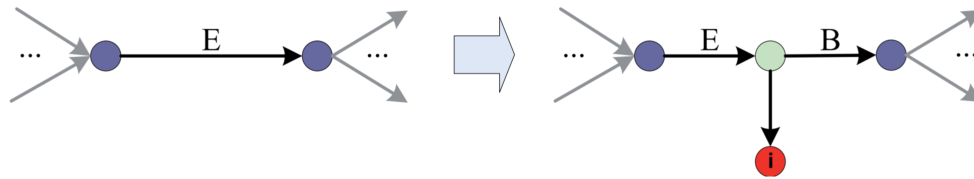



**Abbildung 4.4:** Integration einer Bedingung

für den Übergang in den neuen Entscheidungszustand übernommen. Die Prüfung der zu integrierenden Bedingung erfolgt jetzt bei einem neuen Zustandsübergang zu dem ursprünglichen Ziel-Zustand. Ein Zustandsübergang vom Entscheidungszustand zum Invalid-Zustand wird durchlaufen, wenn die Bedingung nicht erfüllt ist.

Sind weitere Bedingungen an dem gleichen Zustandsübergang zu integrieren, werden diese ebenfalls in den mit $B$ gekennzeichneten Zustandsübergang eingefügt. Dies ist möglich, da mehrere Bedingungen an einem Zustandsübergang Und-verknüpft sind, das heißt es müssen alle Bedingungen erfüllt sein.

Die Integration von Bedingungen erfolgt zwischen zwei Zuständen im erzeugten Automaten zur Musterstrukturerkennung. Daraus ergibt sich die Eigenschaft, dass niemals zwei Entscheidungszustände direkt miteinander verbunden sein können.

An dieser Stelle sei noch auf drei besondere Aspekte des vorgestellten Verfahrens hingewiesen: (1) Es ist möglich, dass die Prüfung einer Bedingung in mehrere Zustandsübergänge integriert werden muss. Eine solche Situation kann bei der Eliminierung von $\epsilon$-Übergängen entstehen. (2) Bedingungen, die sich auf den Minimum- oder Maximum-Wert eines Feldes beziehen, könnten unter bestimmten Bedingungen schon früher geprüft werden. Beispielsweise ist die Bedingung `a[].min.X > 5` nicht mehr zu erfüllen, wenn einmal ein Element des Feldes `a` für das Datenfeld `X` einen Wert $\leq 5$ aufweist. Nachteilig bei dieser Optimierung ist, dass die Bedingung bei jedem Element des Felds geprüft werden müsste. (3) Die Prüfung der Bedingungen erfolgt außerhalb der Mustersemantik, das heißt ein auftretendes Ereignis wird beispielsweise bei einer `SKIPTILLNEXT`-Semantik *nicht* übersprungen wenn es zu einer nicht erfüllten Bedingung führt. Dieses (gewünschte) Verhalten führt zu einer Trennung von Musterspezifikation und Bedingungen: die Semantik beschreibt das Verhalten bei der Erkennung von Konstellationen - die Bedingungen müssen zusätzlich erfüllt sein.

### 4.3.3 Compiler

Für die Entwicklung des Compilers wurde das Werkzeug *Coco/R*[2] von der Universität Linz in der C++-Variante verwendet. Coco/R erlaubt die direkte Umsetzung der Sprache zur Musterspezifikation (siehe Tabelle 4.1) und ermöglicht die Entwicklung eines 1-Pass Compilers, der Regelspezifikationen nach dem beschriebenen Verfahren in deterministische, endliche Automaten übersetzt.

Die Compiler-Implementierung orientiert sich an den in [22] dargestellten Prinzipien. In diesem Abschnitt werden daher nur ausgewählte Implementierungsdetails des Compilers beschrieben, insbesondere die Erzeugung des Binärcodes für eine Musterspezifikation.

#### Parsen von Musterspezifikationen

Für die Implementierung des Compilers wurden zwei Grammatiken spezifiziert: (1) Eine Grammatik zum Einlesen von C-Header Dateien. Mit Hilfe dieser Grammatik werden Ereig-

nistypen und deren Datenfelder sowie Ereignis-IDs aus einer C-Header Datei des WMK extrahiert, (2) eine Grammatik zum Einlesen von Mustern, Bedingungen und Aktionen. Diese Grammatik stellt den Kern des entwickelten Compilers dar.

Nach dem Lesen eines `EVENTS`-Ausdrucks wird die angegebene Datei eingelesen. Verfügbare Ereignisse werden in einer Tabelle zusammengefasst und zur Überprüfung der zulässigen Verwendung von Ereignistypen und -feldern genutzt. Die entsprechende Komponente des Compilers ist in der prototypischen Implementierung ausschließlich für den WMK umgesetzt worden - zur Verwendung mit anderen Instrumentierungsframeworks sind entsprechend andere Verfahren zum Einlesen der verfügbaren Ereignistypen zu entwickeln.

Für die Erstellung des Automaten werden die einzelnen Musterelemente schrittweise miteinander verknüpft. Dazu werden die in Abschnitt 4.3.2 beschriebenen Konstruktionselemente verwendet.

Im nachfolgenden Listing 4.2 ist ein auf das Wesentliche reduzierter Auszug der Compiler-Implementierung dargestellt. Der Code stellt einen Teil der Sprachgrammatik dar. Zusätzlich erlaubt Coco/R das Einfügen von Programmcode (in `(. .)`-Blöcken) in die Sprachbeschreibung. Dieser Code wird ausgeführt, wenn das voranstehende Element gelesen wurde.

**Listing 4.2:** Compiler Implementierung

```
1  eventstructure = ( eventspecification | eventsequence | eventalternative | ↩
       eventnegation ).
2
3  eventsequence
4    =
5    seqStartT
6    eventstructure
7    {
8    seqSeparatorT
9    eventstructure  (.
10                     RuleConstructor rc2 = ctorStack.pop();
11                     RuleConstructor rc1 = ctorStack.pop();
12                     ctorStack.push(SequenceOp(rc1,rc2));
13                   .)
14   }
15   seqEndT
16   .
17
18 eventalternative
19   =
20   altStartT
21   eventstructure
22   altSeparatorT
23   eventstructure  (.
24                     RuleConstructor rc2 = ctorStack.pop();
25                     RuleConstructor rc1 = ctorStack.pop();
26                     ctorStack.push(AlternativeOp(rc1,rc2));
27                   .)
28   {
29   altSeparatorT
30   eventstructure  (.
31                     RuleConstructor rc2 = ctorStack.pop();
32                     RuleConstructor rc1 = ctorStack.pop();
33                     ctorStack.push(AlternativeOp(rc1,rc2));
```

```
34                    .)
35  }
36  altEndT
37  .
38
39 eventnegation
40  =
41  negStartT
42  (
43  eventspecification
44  |
45  eventsequence
46  |
47  eventalternative
48  )                 (.
49                      RuleConstructor rc = ctorStack.pop();
50                      ctorStack.push(NegationOp(rc1,rc2));
51                    .)
52  .
53
54 eventspecification
55  =
56  identifierT
57  [
58  eventlength
59  ]
60  [
61  ':'
62  identifierT
63  ]                 (.
64                      RuleConstructor rc = BasicElement();
65                      ctorStack.push(rc);
66                    .)
67  .
```

Der Konstruktionsstack `ctorStack` enthält die Teilelemente des Automaten, die während der Konstruktion entstehen. Eine `eventspecification` führt zu einem elementaren Automatenelement - entweder einer Repräsentation eines einfachen Ereignis oder eines Feldes von Ereignissen. Dieses wird auf dem Stack abgelegt. Bei den anderen Konstrukten werden zwei Elemente (oder nur ein Element bei der Negation) vom Stack genommen, verknüpft und wieder abgelegt. In den verwendeten Funktionen `SequenceOp`, `AlternativeOp` und `NegationOp` sind die entsprechenden Verknüpfungsoperationen implementiert. Nachdem ein Muster vollständig eingelesen und so verarbeitet wurde, befindet sich genau ein Element auf dem Stack - die fertige Repräsentation des Automaten.

Nach diesen Schritten ist die Grundstruktur des Automaten aufgebaut. Sofort danach, also nach Abschluss des `PATTERN`-Ausdrucks im Skript, werden durch den Compiler die eventuell vorhandenen $\epsilon$-Übergänge entfernt und die Mustersemantik integriert. Die Prüfung von weiteren Bedingungen und sonstige Einstellungen am Automaten werden eingefügt, wenn entsprechende Elemente im Skript vorhanden sind und eingelesen wurden. Der so entstandene fertige Automat wird abschließend in Binärform in die Ausgabedatei geschrieben.

### Codegenerierung

Das Binärformat der kompilierten Regeln ist so aufgebaut, dass es direkt dem Abbild des Automaten im Hauptspeicher entspricht. Das heißt, dass der Inhalt einer Regeldatei beim Laden direkt in den Speicher kopiert werden und initialisiert werden kann. Das Binärformat der kompilierten Regeln ist in Abbildung 4.5 dargestellt.

Im Kopf der Datei sind allgemeine Informationen über die kompilierte Regel enthalten: (1) eine Signatur, um entsprechende Dateien zu erkennen (*Magic Bytes* = `EPC`), (2) Flags, die den Typ der Regel beschreiben, (3) Größe des Laufzeitzustandes und der Ergebnisdaten, (4) Offsets (relativ zum Beginn der Datei) zu Listen, die Transitionen, Bedingungen, Aktionen und Wertebeschreibungen enthalten, (5) die Anzahl der Zustände des Automaten, und (6) den Namen der Regel. Da der Regelname eine variable Länge besitzt, gilt dies auch für den gesamten Kopf der Datei.

Beim Laden einer Regel müssen Verweise auf Bestandteile der Regelbeschreibung initialisiert werden: in der Datei selbst sind anstelle der Verweise relative Offsets gespeichert, zu denen nach dem Laden die Startposition der Regel im Speicher addiert werden müssen.

Nach dem Dateikopf folgt eine Liste von Verweisen auf die Transitionstabellen für jeden Zustand des Automaten. Eine Transitionstabelle beschreibt die ausgehenden Transitionen von einem Zustand. Dazu werden (1) die Gesamtzahl der Transitionen, (2) Verweise auf die Default-Transition, und (3) Verweise zu allen weiteren Transitionen gespeichert.

Eine Transition wird durch die folgenden Bestandteile beschrieben: (1) die Nummer des Zustandes, die durch die Transition erreicht wird, (2) Flags, die den Transitionstyp beschreiben (zum Beispiel Replikations-Transition), (3) Verweise auf Bedingungen, und (4) Verweise auf Aktionen, die ausgeführt werden müssen wenn die Transition ausgewählt wird.

Abschließend folgen in der Datei die von Transitionen referenzierten Informationen zu Bedingungen, Aktionen und Werten. Bedingungen sind von einem bestimmten Typ, der durch eine Vergleichsoperation bestimmt ist. In einer Bedingung können vier Werte verknüpft und verglichen werden: $v_1 \circ_a v_2 \bullet v_3 \circ_b v4$; dabei sind $\circ_a$ und $\circ_b$ arithmetische Operatoren und $\bullet$ der Vergleichsoperator. Aktionen sind durch einen Typ und bis zu zwei Werten definiert. Der Typ einer Aktion ist beispielsweise „Kopieren“ oder „Erhöhen“ - die Werte beschreiben die entsprechenden Parameter der Aktion.

Werte beschreiben entweder einen Teilbereich des Laufzeitzustandes, einen Teilbereich des aktuell zu verarbeitenden Ereignis oder eine Konstante - dies ist der Typ des Wertes. Der Datentyp des Wertes beschreibt den konkreten C-Datentyp, der in dem Wert gespeichert ist. Außerdem ist in einem 8 Byte umfassenden Datenfeld der repräsentierte Zahlenwert gespeichert.

Das dargestellte Binärformat kompilierter Regeln besitzt zwei vorteilhafte Eigenschaften:

**Kompakte Speicherung:** Da die Daten vor allem über Verweise strukturiert sind, ist es möglich, dass beispielsweise unterschiedliche Transitionen auf die gleichen Bedingungen verweisen oder unterschiedliche Aktionen die gleichen Werte referenzieren.

**Einfache Umstrukturierung:** Die Laufzeitumgebung prüft Bedingungen in der Reihenfolge, in der sie in der Datei abgelegt sind. Änderungen dieser Reihenfolge erfordern daher nur einfache Operationen, die Verweise austauschen. Dadurch lässt sich die Ausführung von Regeln einfach optimieren.

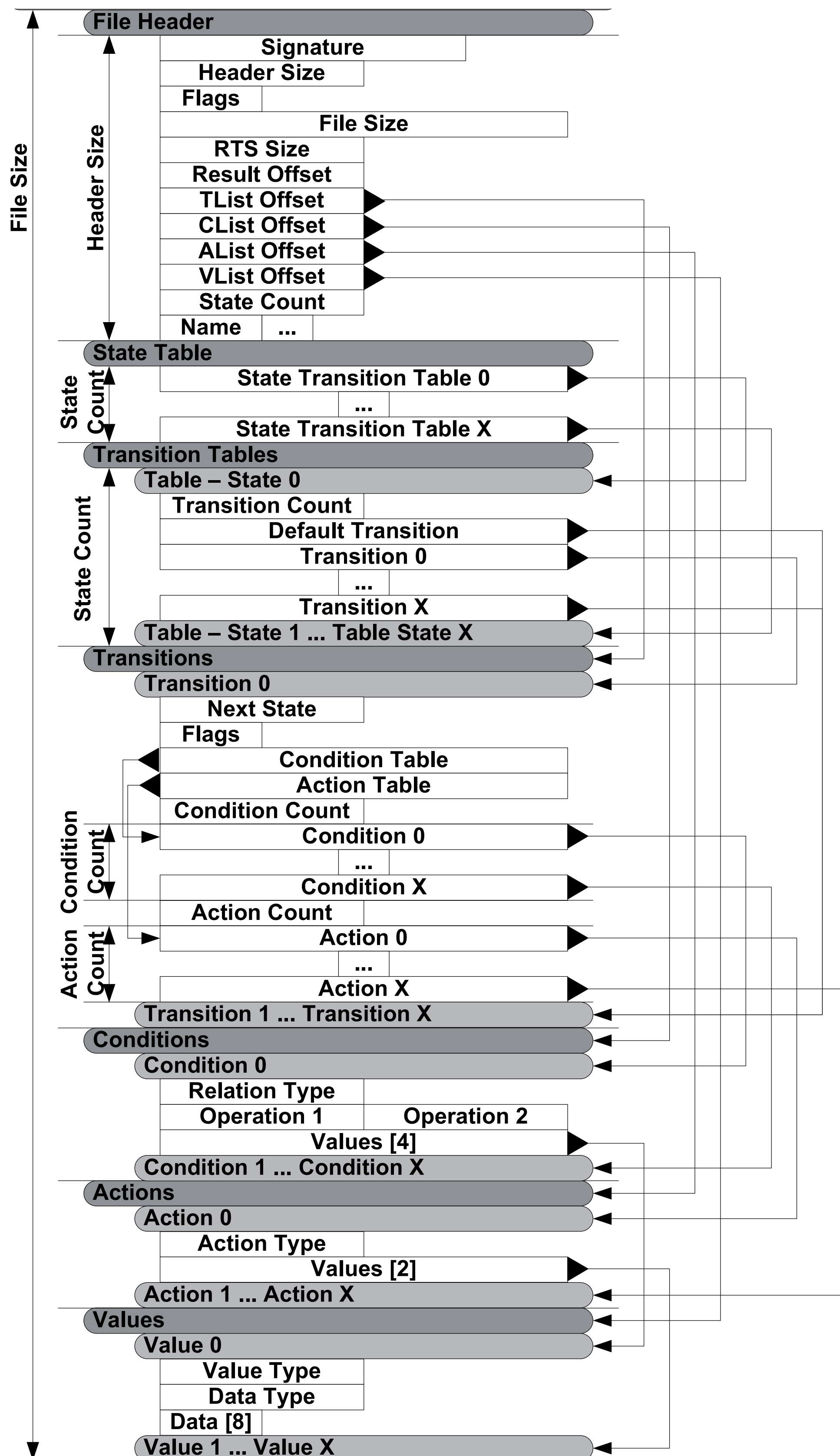



**Abbildung 4.5:** Binärstruktur kompilierter Regeln

Eine Datei, die eine Regel in Binärform enthält, kann nach den dargestellten Informationen zur Mustererkennung weitere Meta-Daten enthalten. Dies kann beispielsweise zur Speicherung von Statistiken zur durchgeführten Mustererkennung verwendet werden: zu jeder Transition und zu jeder Bedingung kann abgelegt werden, wie oft sie geprüft wurden und was das Ergebnis der Prüfung war.

#### Erweiterungsmöglichkeiten

Erweiterungen des Compilers können in zwei Klassen eingeteilt werden: (1) Erweiterungen, die das hier vorgestellte Binärformat weiterverwenden und (2) Erweiterungen, die Anpassungen am Binärformat erfordern.

Die erste Klasse von Erweiterungen erfordert keine Änderungen an der Laufzeitumgebung und sind demnach einfacher umzusetzen. Beispiele für solche Erweiterungsmöglichkeiten sind:

**Weitere Instrumentierungsinfrastrukturen:** Der Compiler liest Dateien mit Ereignistypbeschreibungen ein (`EVENTS`-Ausdruck im Skript) und erzeugt eine Ereignistyptabelle. Zur Unterstützung weiterer Instrumentierungsinfrastrukturen sind also weitere Komponenten notwendig, die andere Formate in eine solche Ereignistyptabelle umsetzen.

**Negationsoperator:** Mehrfache Verneinung beliebiger Ausdrücke sind für eine vollständige Unterstützung der beschriebenen Sprache notwendig. Die Implementierung des Negationsoperators muss entsprechend erweitert werden.

**Wiederholung beliebiger Sub-Muster:** In regulären Ausdrücken kann der *-Operator verwendet werden um eine beliebig oft auftretende Wiederholung eines Sub-Musters zu fordern. Dieses Verhalten lässt sich ebenfalls mit einer Erweiterung des Compilers implementieren.

Werden Anpassungen am Binärformat vorgenommen, sind zusätzliche Konzepte umsetzbar, die auch eine Erweiterung der Laufzeitumgebung erfordern. Ein Beispiel ist die Unterstützung von Zeichenketten als Werte und in Operationen. Weitere Beispiele für solche Erweiterungen sind in verwandten Arbeiten zu finden und werden in Kapitel 6 dargestellt. Für die in Kapitel 5 vorgestellten Fallstudien war der Compiler in der hier beschriebenen Form ausreichend.

## 4.4 Laufzeitumgebung zur Mustererkennung

In diesem Abschnitt wird die Implementierung der erstellten Laufzeitumgebung zur online Verarbeitung von Ereignissen beschrieben. Eine schematische Übersicht der Laufzeitumgebung ist in Abbildung 4.6 dargestellt.

Die Grundlage bildet der Windows Monitoring Kernel (siehe Kapitel 3). Die von Ereignisprovidern erzeugten Ereignisse werden gesammelt und zu einem systemweiten Ereignisstrom zusammengefasst. Dieser Ereignisstrom wird von der Laufzeitumgebung als Eingabe gelesen und verarbeitet.

Auf der entgegengesetzten Seite der Laufzeitumgebung befindet sich eine Anwendung im Usermode, die die Laufzeitumgebung nutzt. Die Anwendung besteht im Allgemeinen aus

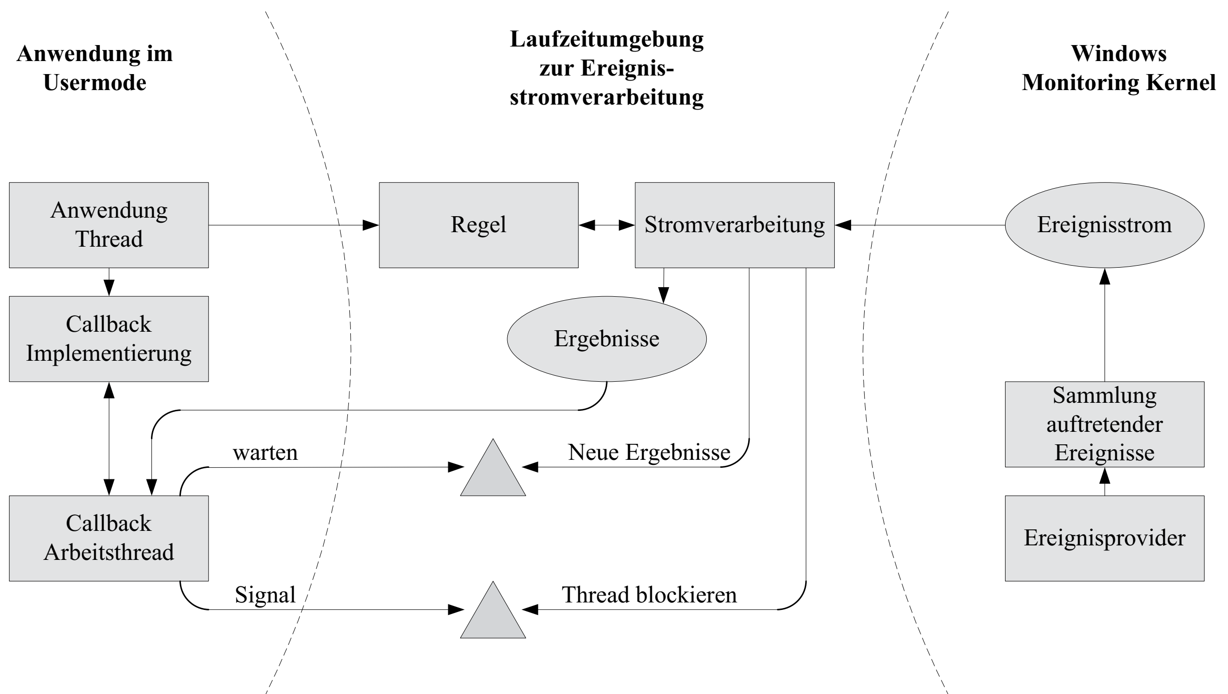



**Abbildung 4.6:** Überblick: Laufzeitumgebung zur Verarbeitung von Ereignisströmen

mehreren Threads. Ein Thread kann eine Regel in die Laufzeitumgebung laden und zu dieser Regel eine Callback-Implementierung bereitstellen. Die Kommunikation mit der Laufzeitumgebung erfolgt dabei über einen Arbeitsthread, der ebenfalls im Kontext der Anwendung ausgeführt wird und dessen Implementierung von einer Laufzeitbibliothek bereit gestellt wird.

Die Kommunikation zwischen Anwendung und Laufzeitumgebung im Betriebssystemkern erfolgt, ganz abstrakt beschrieben, auf die folgende Weise: werden bei der Ereignisstromverarbeitung zur Regel passende Muster erkannt, werden diese in einen Ergebnispuffer gespeichert. Der auf neue Ergebnisse wartende Callback-Arbeitsthread wird aktiviert und die Aktivität, die zur Erkennung des Muster führte, wird blockiert. Nachdem der Arbeitsthread die Ergebnisse gelesen und den anwendungspezifischen Callback ausgeführt hat, wird der blockierte Thread fortgesetzt oder abgebrochen. Anschließend wartet der Arbeitsthread auf die nächsten Ergebnisse.

In den nachfolgenden Abschnitten werden Implementierungsdetails der Laufzeitumgebung genauer beschrieben.

### 4.4.1 Abarbeitungsmodell

Die Mustererkennung erfolgt basierend auf den generierten deterministischen, endlichen Automaten (DEA). Konstellationsbeschreibungen in DEA-Form werden von der Laufzeitumgebung geladen und auftretende Ereignisse entsprechend verarbeitet.

Der genaue Abarbeitungszustand eines Automaten wird durch einen Laufzeitzustand beschrieben. Die Größe und die Struktur des Laufzeitzustandes wird vom Compiler aus der Regelbeschreibung abgeleitet. Ein Laufzeitzustand enthält mindestens die Information, welcher Zustand im Automaten gerade erreicht ist. Darüber hinaus sind alle Informationen enthalten, die für die Abarbeitung des Automaten relevant sind - beispielsweise Werte von jenen Ereignisfeldern, die in Bedingungen verwendet werden oder Informationen, die für die Rückgabe von

Ergebnissen benötigt werden. Bedingungen für Zustandsübergänge im Automaten sind beim Compilieren generiert worden, ebenso Aktionen, die Elemente im Laufzeitzustand verändern.

Das Abarbeitungsmodell beschreibt (1) wie Übergänge im Automaten gefunden werden, also wie ein Zustandsübergang für einen gegebenen Laufzeitzustand und ein gegebenes Ereignis gefunden wird und (2) wie Aktionen ausgeführt werden, also Felder im Laufzeitzustand geändert werden.

Von der Laufzeitumgebung geladene Regeln haben im Speicher die in Abschnitt 4.3.3 dargestellte Form. Zu jedem Automatenzustand existiert eine Transitionstabelle, die alle möglichen Zustandsübergänge enthält. Von diesen möglichen Übergängen ist für ein gegebenes Ereignis genau ein Übergang gültig, das heißt bei genau einem Übergang sind alle mit ihm verknüpften Bedingungen erfüllt.

Im ersten Schritt werden daher der Reihe nach die Bedingungen aller möglichen Übergänge ausgewertet. Sind bei einem Übergang alle Bedingungen erfüllt, kann die Suche sofort abgebrochen werden, da es sich um einen deterministischen Automaten handelt. In einem Zustand wird zwischen regulären Zustandsübergängen und dem Default-Übergang unterschieden. Ist keiner der regulären Übergänge gültig, wird der Default-Übergang verwendet.

Nach diesem Schritt ist der durchzuführende Übergang identifiziert. Dabei sind vier verschiedene Fälle beziehungsweise Typen von Zustandsübergängen zu unterscheiden:

**Regulärer Zustandsübergang:** In diesem einfachen Fall wird der bearbeitete Laufzeitzustand angepasst. Der Zielzustand des Übergangs wird als aktueller Zustand im Laufzeitzustand vermerkt. Die Aktionen werden durchgeführt und so gegebenenfalls Felder des aktuellen Ereignis gespeichert oder aggregierte Werte angepasst.

**Zustandsübergang mit Replikation:** Dieser Übergang unterscheidet sich von einem regulären Übergang dadurch, dass vor der Aktualisierung des Laufzeitzustands erst eine Kopie angelegt wird. Alle Anpassungen werden dann an der Kopie durchgeführt, so dass danach zwei Laufzeitzustände vorhanden sind: das unveränderte Original im ursprünglichen Zustand und die aktualisierte Kopie im neuen Zustand.

**Zustandsübergang zum Return-Zustand:** In diesem Fall werden erst die Aktionen durchgeführt und anschließend der Laufzeitzustand als Ergebnis gekennzeichnet. Nach der Verarbeitung aller zu einer bestimmten Regel gehörenden Laufzeitzustände werden die Ergebnisse weiter verarbeitet.

**Zustandsübergang zum Invalid-Zustand:** Die Prüfung der Bedingungen ergab, dass das spezifizierte Muster nicht mehr im Ereignisstrom erkannt werden kann. Der Übergang in den Invalid-Zustand führt dazu, dass der entsprechende Laufzeitzustand freigegeben wird.

Abschließend wird geprüft, ob der erreichte Zustand ein Entscheidungszustand ist, das heißt ein Zustand der sofort ausgewertet werden kann ohne auf das nächste Ereignis zu warten. Ist dies der Fall, wird mit Hilfe der Transitionstabelle des Entscheidungszustands sofort der nächste Automatenschritt durchgeführt.

### 4.4.2 Verarbeitung von Ereignissen

Es wird vorausgesetzt, dass eine Infrastruktur zur Aufzeichnung von Ereignissen vorhanden ist. Diese Infrastruktur ist im Prinzip unabhängig von der hier beschriebenen Laufzeitumgebung. Im Rahmen dieser Arbeit wurde der in Kapitel 3 beschriebene *Windows Monitoring Kernel*

(WMK) als Grundlage verwendet. Die Infrastruktur stellt Funktionen zur Verfügung, die in einem bestimmten Ausführungskontext ein Ereignis anzeigen können. Bei der Anzeige eines Ereignisses werden der Zeitpunkt und die Parameter des Ereignisses gesichert.

Im WMK werden Ereignisse im Hauptspeicher gepuffert und bei Bedarf in einer Logdatei auf der Festplatte gesichert. Die sofortige Verarbeitung von Ereignisströmen soll es ermöglichen, auf eine Logdatei verzichten zu können. Daher wird ein anderer Ansatz zur Pufferung von Ereignisdaten verwendet.

Das in Abschnitt 3.3 beschriebene Verfahren der Pufferverwaltung für Ereignisdaten wird auf eine Ausführungskontext-lokale Pufferung umgestellt. Das heißt, jeder Ausführungskontext hat einen eigenen Speicherbereich für ein auftretendes Ereignis. Eine Referenz zu diesem Speicherbereich wird in der `ETHREAD`-Datenstruktur des Windows Kerns ergänzt. Das beschriebene Makro zur Anforderung (*allocate*) eines Puffers für Ereignisdaten wird so modifiziert, dass immer der entsprechende Ausführungskontext-lokale Speicherbereich verwendet wird. Die Zusammenfassung der auftretenden Ereignisse zu einem Ereignisstrom erfolgt im Makro zum Abschließen (*commit*) einer Ereignisaufzeichnung.

Die Thread-Datenstruktur `ETHREAD` wird um ein Feld `EventPuffer` vom Typ `PVOID` erweitert. Bei der Initialisierung neuer Threads wird im nicht-auslagerbaren Speicher des Betriebssystems ein entsprechender Speicherbereich für den neuen Thread angelegt. Die Größe des Speicherbereichs wird dabei statisch zur Kompilierzeit des Kernels festgelegt. Die Analyse verschiedener getesteter Anwendungen ergab, dass eine Puffergröße von 1024 Byte groß genug ist.

Tritt ein Ereignis auf, welches die Speicherung einer größeren Datenmenge erfordert, wird auf das ursprüngliche Schema zur Ereignisspeicherung zurückgegriffen: entsprechende Ereignisse werden weiterhin in einen globalen Ereignispuffer geschrieben. Zugriffe müssen dann weiterhin synchronisiert erfolgen.

Dieses Verfahren wird ebenso angewendet, wenn ein Ereignis nicht im Ausführungskontext eines Threads auftritt, sondern in einem anderen Kontext, beispielsweise in einer Interrupt Service Routine. Auch in einem solchen Fall wird der globale Puffer verwendet.

Die gespeicherten Ereignisse werden an die Mustererkennung weitergegeben. Die Weitergabe erfolgt synchron, das heißt (1) die Mustererkennung nimmt zu jedem Zeitpunkt höchstens ein Ereignis entgegen und (2) die Ausführungskontexte sind blockiert, solange das Ereignis nicht entgegengenommen wurde.

Da die Mustererkennung höchstens ein Ereignis verarbeitet, sind die Ereignisse an dieser Stelle bereits serialisiert und in einen Ereignisstrom eingeordnet. Die geforderten Eigenschaften von Ereignisströmen (siehe Abschnitt 2.4.2) sind an dieser Stelle im System gegeben.

Die Verarbeitung der serialisierten Ereignisse erfolgt jetzt abhängig von den Regeltypen: sind synchrone Regeln definiert, muss der Ausführungskontext weiter blockiert bleiben, bei asynchronen Regeln kann das Ereignis aus dem Ausführungskontext-lokalen Puffer kopiert werden und später bearbeitet werden. Aktivitäten im Ausführungskontext können dann fortgesetzt werden.

Mit dem beschriebenen Verfahren zur Ereignisaufzeichnung kann auf einen systemweiten Puffer verzichtet werden, solange keine asynchronen Regeln bearbeitet werden. Bei der Verarbeitung synchroner Regeln sind keine synchronisierten Pufferzugriffe notwendig. Abhängig von der konkret zu erkennenden Ereigniskonstellation ist darüber hinaus der Speicherbedarf geringer, da nur die Daten gespeichert werden, die tatsächlich notwendig sind.

### 4.4.3 Verwaltung von Regeln

Die Erkennung von Ereigniskonstellationen, beziehungsweise die Abarbeitung der DEAs einzelner Regelbeschreibungen erfordert grundsätzlich zwei Schritte: (1) Finden der Menge von aktiven Laufzeitzuständen und (2) Kombination der Laufzeitzustände mit dem aktuellen Ereignis. Der zweite Schritt ist im Abarbeitungsmodell bereits beschrieben worden. In diesem Abschnitt geht es um die effiziente Verwaltung von Regeln und der dazugehörigen Laufzeitzustände.

Prinzipiell müssen alle aktiven Laufzeitzustände mit dem aktuellen Ereignis kombiniert werden. Je nach Art der zu erkennenden Konstellation sind dies unter Umständen sehr viele Zustände, so dass der Einfluss der Laufzeitumgebung auf das Gesamtsystem mit der Anzahl der Laufzeitzustände wächst. Analog kann jedes System zur Aufzeichnung von Ereignissen durch sehr hohe Ereignisraten überlastet werden. Vor diesem Hintergrund war das Ziel der nachfolgend dargestellten Konzepte zur Verwaltung von Regeln, wenn möglich die Zahl der einzeln zu prüfenden Laufzeitzustände zu reduzieren und so den Einfluss der Laufzeitumgebung auf das Gesamtsystem zu reduzieren.

Die Zahl der zu prüfenden Laufzeitzustände kann mit Hilfe verschiedener Heuristiken reduziert werden, indem mehrere Laufzeitzustände zusammengefasst und - auf verschiedene Art - gleichzeitig bearbeitet werden.

Zu einem gegebenen Ereignis muss die Menge von Regeln identifiziert werden, für die dieses Ereignis relevant ist. Dies gilt sowohl für die synchrone Verarbeitung (= die Regeln müssen gefunden werden, solange der Ausführungskontext noch blockiert ist) und für die asynchrone Verarbeitung (= das Ereignis wurde zwischengespeichert und wird später verarbeitet).

Betrachtet man eine gegebene Regelbeschreibung und den Typ eines neuen Ereignis, können zwei Fälle unterschieden werden: (1) der Ereignistyp wird in der Musterbeschreibung verwendet, oder (2) der Ereignistyp wird nicht in der Musterbeschreibung verwendet.

Im ersten Fall müssen die Laufzeitzustände überprüft werden und das Ereignis entsprechend verarbeitet werden. Im zweiten Fall können, je nach definierter Regel-Semantik, mehrere Laufzeitzustände gleichzeitig verarbeitet werden. Bei einer `STRICTSEQUENCE` Semantik werden alle Laufzeitzustände der entsprechenden Regel ungültig, unabhängig von ihrem aktuellen Zustand. Bei `STRICTPARTITION` müssen bei allen Laufzeitzuständen die Partitionierungsbedingungen geprüft werden. Bei `SKIPTILL*` Semantik können entsprechende Ereignisse ignoriert werden.

Die Verarbeitung von Laufzeitzuständen auf dieser höheren Ebene verringert die Anzahl der durchzuführenden Vergleiche bei der Abarbeitung des Automaten: wenn aus der Semantik der Regel bereits abgeleitet werden kann, welche Transition durchgeführt werden muss, brauchen keine anderen Transitionen geprüft werden. Beispielsweise ist bei `SKIPTILL*` Semantik klar, dass der Automat im aktuellen Zustand verbleibt, wenn der Typ eines Ereignisses nicht in der Musterbeschreibung einer Regel verwendet wird.

Die Prüfung des Ereignistyps mit Semantik und Musterbeschreibung der Regeln ist ein Beispiel für die zusammengefasste Bearbeitung mehrerer Laufzeitzustände.

Ein anderes Beispiel ist die Bearbeitung der Laufzeitzustände einer Regel. Angenommen, der Semantik/Ereignistyp-Vergleich ergab, dass die Laufzeitzustände einer Regel geprüft werden müssen. Jetzt kann für jeden Zustand des Automaten bestimmt werden, welche Ereignistypen erwartet werden. Laufzeitzustände die den Typ des aktuellen Ereignis nicht erwarten, können wieder basierend auf der Regelsemantik verarbeit werden. Durch Verwendung von Tabellen,

die einem bestimmten Ereignistyp eine Liste von Laufzeitzuständen zuordnen, die aktuell Ereignisse eines entsprechenden Typs verarbeiten, ist diese Verfahren effizient implementierbar.

Weitere Möglichkeiten ergeben sich aus der Gruppierung von Laufzeitzuständen nach den aktuell gespeicherten Datenfeldern. Treten dabei größere Gruppen von Laufzeitzuständen mit den gleichen Daten auf, müssen entsprechende Relationen nur einmal für die gesamte Gruppe geprüft werden. Welche Datenfelder sich für eine solche Gruppierung eignen und welche Relationsprüfungen dann optimiert werden können, ist durch den Compiler feststellbar. Im Rahmen des erstellten Prototyps ist diese Heuristik nur experimentell implementiert worden.

Generell muss für alle Laufzeitzustände die vollständige Suche nach einer Transition durchgeführt werden. Dazu werden die Relationen geprüft und eine Transition ausgewählt.

Dabei können zur Optimierung Heuristiken angewendet werden, die dazu führen, dass (1) häufig gewählte Transitionen und (2) häufig nicht zutreffende Relationen einer bestimmten Transition zuerst geprüft werden. Die Laufzeitumgebung kann Statistiken über die Verarbeitung einer konkreten Regel - also über die Ergebnisse von Relationsprüfungen und über gewählte Transitionen - erfassen. Beim Laden einer Regel wird die Reihenfolge der Transitionen und Relationen in den entsprechenden Tabellen (siehe Abschnitt 4.3.3) entsprechend angepasst.

### 4.4.4 Pufferverwaltung und Synchronisation

Die Codebasis des Windows Monitoring Kernels bildet die Grundlage für die Implementierung der Laufzeitumgebung für die online Verarbeitung von Ereignisströmen. Die Pufferverwaltung und die Synchronisation ist daher ähnlich - unterscheidet sich aber in ein paar wichtigen Punkten, die nachfolgend kurz dargestellt werden.

Bei der dynamischen Verarbeitung von Ereignissen sind die folgenden drei Klassen von Informationen zu speichern: (1) die Laufzeitzustände der einzelnen Regeln, (2) die Ereignisdaten und (3) Verweise auf die Ergebnisse der Mustererkennung, wenn die vollständige Ereignisfolge zurückgegeben werden soll.

Wie bereits beschrieben werden neue Ereignisse in einem Ausführungskontext-lokalen Speicherbereich zwischengespeichert. Jeder Thread besitzt also einen exklusiv genutzten Puffer für Ereignisdaten. Die Laufzeitumgebung verwaltet einen Puffer für Ereignisdaten die (1) relevant sind für Ergebnisse einer Mustererkennung bei der die gesamte Ereignissequenz zurückgegeben werden soll (`RETURN SEQUENCE`), oder die (2) im Rahmen von asynchronen Regeln verarbeitet werden. Soll nicht die gesamte Ereignissequenz erfasst werden, werden die für das Ergebnis relevante Informationen direkt im Laufzeitzustand gespeichert.

Neue Ereignisse werden vom Stromverarbeiter nach dem folgenden Vorgehen übernommen: Zuerst wird geprüft, ob das neue Ereignis für synchron zu bearbeitende Regelen relevant ist. In einem solchen Fall, wird das Ereignis durch diese Regeln verarbeitet. Sollte das neue Ereignis für das Ergebnis einer Mustererkennung notwendig sein, wird es in den Ereignisdaten-Puffer kopiert. Anschließend wird geprüft, ob das neue Ereignis für asynchron zu bearbeitende Regeln relevant ist. Ist das der Fall, werden die Ereignisdaten ebenfalls in den Puffer kopiert (wenn dies noch nicht aufgrund der Bearbeitung durch eine synchrone Regel geschehen ist). Außerdem wird eine Referenz auf das Ereignis in der Ereigniswarteschlange gespeichert.

Nach diesen Schritten, steht der Ausführungskontext-lokale Speicherbereich für das nächste Ereignis wieder zur Verfügung. Ereignisse aus der Warteschlange werden vom Thread zur asynchronen Bearbeitung verarbeitet.

Laufzeitzustände enthalten alle Informationen zu aktuell erkannten Musterfragmenten. Jede Regel hat einen eigenen Speicherpool für ihre Laufzeitzustände. Die Größe eines einzelnen Lauzeitzustandes wird vom Compiler bestimmt. Die Poolgröße kann auf zwei verschiedenen Wegen festgelegt werden - und kann entweder (1) eine bestimmte Anzahl von Einträgen enthalten, oder (2) eine bestimmte Größe besitzen. Die tatsächliche Poolgröße bestimmt dann, wie viele Muster maximal parallel erkannt werden können.

Die Verwaltung der Laufzeitzustände in einem Speicherpool wurde aus den folgenden Gründen gewählt: (1) der Speicherbedarf zur Bearbeitung einer bestimmten Regel ist vorhersehbar, (2) die Speicheranforderung und -freigabe von (in der Regel nur wenige Bytes großen) Laufzeitzuständen ist ineffizient, und (3) die Reservierung des gesamten benötigten Speichers stellt sicher, dass der Speicher tatsächlich verfügbar ist - wenn nicht, wird dies schon bei der Initialisierung festgestellt.

Der globale Ereignispuffer ist ebenfalls mit Hilfe eine Speicherpools implementiert und unterscheidet sich daher vom ursprünglichen WMK Ansatz. Dies ist notwendig, da es jetzt auch möglich sein muss, auf beliebige Teilsequenzen zuzugreifen. Der Puffer ist unterteilt in kleinere Puffereinheiten. Jede dieser Einheiten verwaltet einen Referenzzähler der beschreibt, wie viele der gespeicherten Ereignisdaten von einem Laufzeitzustand referenziert werden. Erreicht dieser Zähler den Wert 0, kann der entsprechende Puffer überschrieben und zur Speicherung neuer Ereignisdaten verwendet werden.

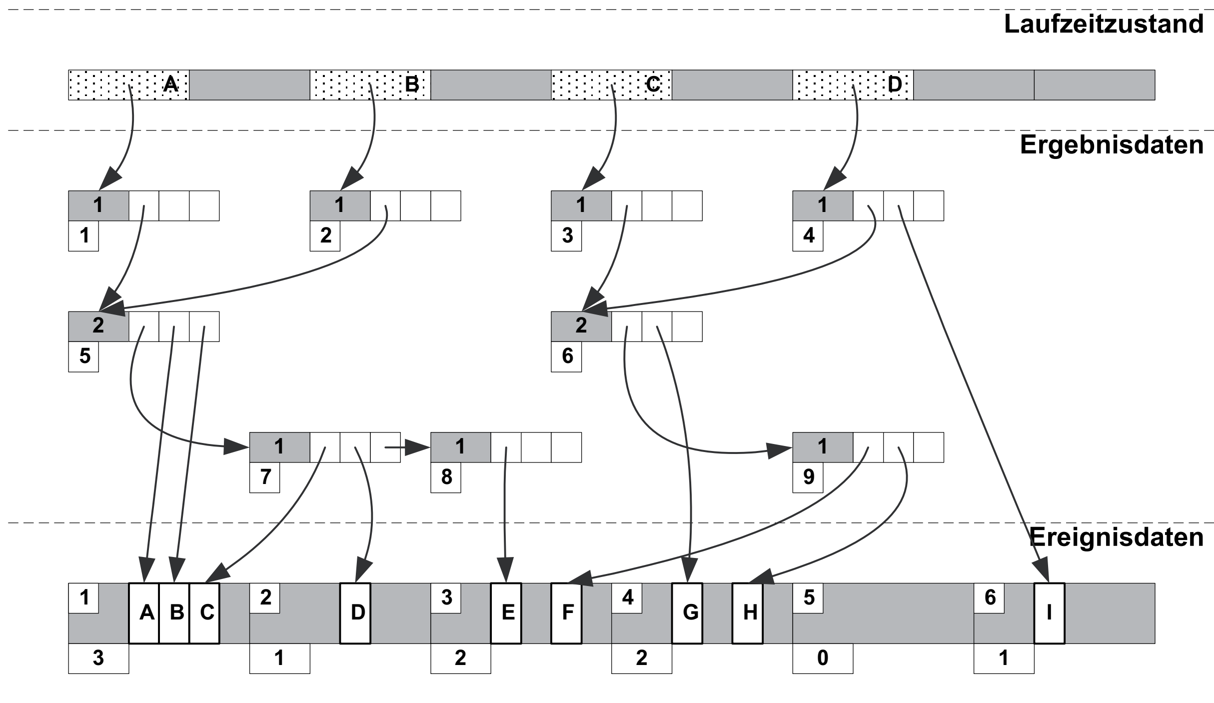



**Abbildung 4.7:** Pufferung der Ergebnisse von Regeln

Das Zusammenspiel der vorgestellten Elemente ist in Abbildung 4.7 dargestellt. Am oberen Bildrand ist der Speicherpool für Laufzeitzustände dargestellt. Der Pool bietet Platz für neun Laufzeitzustände, vier Plätze (A-D) sind momentan in Verwendung. Am unteren Bildrand ist der Ereignisdatenpool dargestellt, dieser enthält sechs Puffer (1-6, obere linke Ecke) mit Ereignisdaten (A-I). Unter jedem Puffer ist der Referenzzähler dargestellt.

In der Bildmitte ist der Speicherpool für Ergebnisdaten dargestellt. Jeder Laufzeitzustand enthält genau einen Verweis auf ein Ergebnisdatenelement. Jedes Ergebnisdatenelement enthält

| Laufzeitzustand | Ergebnissequenz |
|---|---|
| A | C D E A B |
| B | C D E A B |
| C | F H G |
| D | F H G I |

**Tabelle 4.4:** Ergebnissequenzen für das Beispiel

einen Referenzzähler (graue Box) und drei Verweise auf andere Ergebnisdatenelemente oder auf Einträge in einem Ereignisdatenpuffer.

Soll die Ereignissequenz eines Laufzeitzustands (= Ergebnismenge) durch ein neues Element erweitert werden, wird die entsprechende Referenz in dem Ergebnisdatenelement des entsprechenden Laufzeitzustands angefügt. Sollte nur noch ein Platz zur Referenzspeicherung vorhanden sein, wird dieser letzte Platz zur Erweiterung des Ergebnisdatenelements verwendet. Dazu wird auf ein neues Element verwiesen und neue Einträge dort gespeichert. Ein Beispiel ist in der Abbildung dargestellt: Ergebnisdatenelement sieben wird durch Element acht erweitert.

Bei der Replikation von Laufzeitzuständen müssen auch die bisherigen Ergebnisse repliziert werden. Mit dem beschriebenen Schema lässt sich die Replikation einfach implementieren: beide Laufzeitzustände (das Original und das Replikat) verweisen auf ein neues Ergebnisdatenelement. Die neuen Elemente erhalten einen Referenzzählerwert von eins und verweisen auf das ursprüngliche Ergebnisdatenelement des originalen Lauzeitzustandes. Der Referenzzähler des ursprünglichen Ergebnisdatenelements wird auf zwei gesetzt. Anschließend teilen sich beide Laufzeitzustände die Informationen über den bisherigen Verlauf der Mustererkennung.

In der Abbildung 4.7 ist dies bei den Laufzeitzuständen A und B der Fall: beide besitzen einen Verweis auf das Ergebnisdatenelement fünf.

Die belegten Speicherbereiche in den verschiedenen Puffern werden wieder freigegeben, wenn entweder das Muster vollständig erkannt wurde (also der Endzustand im entsprechenden DEA erreicht wird) oder der Laufzeitzustand ungültig wird, das Muster also nicht mehr erkannt werden kann.

Die Ergebnissequenz kann rekonstruiert werden, indem die Ergebnisdatenelemente als Baum betrachtet werden, deren Wurzel in dem Laufzeitzustand gespeichert ist - wird der Baum *left-first* traversiert und die Elemente aus dem Ereignispuffer ausgegeben, so erhält man die Ergebnissequenz. Tabelle 4.4 zeigt die Ergebnisse für das Beispiel aus Abbildung 4.7.

Nachdem (falls notwendig) die Ergebnissequenz zurückgegeben wurde, werden die Referenzzähler angepasst. Beginnend mit dem Ergebnisdatenelement, welches direkt vom Laufzeitzustand referenziert wird, werden die Referenzzähler um eins verringert. Ist der neue Wert des Referenzzählers gleich null, werden alle von diesem Element referenzierten Elemente betrachtet: bei anderen Ergebnisdatenelementen wird ebenfalls der Referenzzähler um eins verringert, bei Elementen aus dem Ereignisdatenpuffer wird der Referenzzähler des entsprechenden Pufferelementes verringert. Erreichen auch Referenzzähler anderer Ergebnisdatenelemente den Wert null, wird das entsprechende Vorgehen rekursiv wiederholt. Erreicht der Referenzzähler eines Pufferelements den Wert null, kann der entsprechende Puffer freigegeben und zur Speicherung neuer Ereignisse verwendet werden.

Das beschriebene Verfahren zur Pufferverwaltung kann effizient implementiert werden. Die Größe der Ereignisdaten wird durch die Instrumentierung bestimmt, die Größe der Laufzeitzustände durch den Compiler - es bleibt also die Betrachtung, wie viel Overhead durch die Ergebnisdatenelemente ensteht.

Die Anzahl der zu speichernden Einträge in einem Ergebnisdatenelement kann ebenfalls durch den Compiler abgeschätzt werden. Sind ausschließlich einfache Ereignisse und Längenbegrenzte Felder in der Musterspezifikation enthalten, kann die maximale Anzahl der zu speichernden Ereignisse direkt bestimmt werden. Nur bei unbegrenzten Feldern und bei Replikation von Laufzeitzuständen wird das beschriebene Verfahren zur Erweiterung von Ergebnisdatenelementen verwendet.

### 4.4.5 Systemaufruf Schnittstelle

Die Schnittstelle zu der Kernel-Implementierung der Laufzeitumgebung wird durch zwei Systemaufrufe realisiert. Ganz Allgemein dient der Aufruf `NtSetInformationEP` dazu, Daten (zum Beispiel Regeln oder Konfigurationsparameter) in den Kernel zu übertragen und dort anzuwenden. Der Aufruf `NtQueryInformationEP` dient dementsprechend dazu, Daten aus dem Kernel zu lesen (zum Beispiel Statusinformationen über die Regelausführung).

**Listing 4.3:** Ereignisverarbeitung: Systemaufruf Schnittstelle

```
1  NTSTATUS
2  NtSetInformationEP(
3    EPINFOCLASS EPInformationClass,
4    PVOID EPInformation,
5    ULONG EPInformationLength
6  );
7
8  NTSTATUS
9  NtQueryInformationEP(
10   EPINFOCLASS EPInformationClass,
11   PVOID EPInformation,
12   ULONG EPInformationLength,
13   PULONG ReturnLength
14 );
```

Die Signatur beider Systemaufrufe ist allgemein gehalten (siehe Listing 4.3), so dass die Funktionen für unterschiedliche Aufgaben verwendet werden können[2]: `EPINFOCLASS` enthält eine Aufzählung der möglichen Operationen - die Interpretation der weiteren Signaturparameter ist vom gewählten Wert in `EPInformationClass` abhängig.

Der durch `EPInformation` referenzierte Speicherbereich wird bei `NtSetInformationEP` für die Übergabe von Parametern zum Kernel und bei `NtQueryinformationEP` zur Rückgabe von Daten aus dem Kernel verwendet. In `EPInformationLength` wird die Puffergröße angegeben; `ReturnLength` gibt an wie viele Bytes an Daten aus dem Kernel zurückgegeben wurden.

[2] Das gewählte Implementierungsmuster (*NtSetInformation** und *NtQueryInformation**) wird auch an anderen Stellen im Windows Kern verwendet, beispielsweise zur Verwaltung von Threads oder Prozessen.

Die Definition von `EPINFOCLASS` ist in Listing 4.4 dargestellt.

**Listing 4.4:** Ereignisverarbeitung: Systemaufruf Funktionen

```
typedef enum _EPINFOCLASS {
  EPPing,
  EPLoadRule,
  EPUnloadRule,
  EPQueryRule,
  EPActivateRule,
  EPDeactivateRule,
  EPRuleResult,
  EPRuleResultContinue,
  EPRuleResultAbort,
  MaxEPInfoClass
} EPINFOCLASS;
```

Nicht alle der möglichen Werte von `EPINFOCLASS` sind für beide Systemaufruf-Funktionen zulässig. Wird ein ungültiger Wert verwendet, gibt der Systemaufruf `STATUS_UNSUCCESSFUL` beziehungsweise `STATUS_INVALID_INFO_CLASS` zurück.

Die Namen der Werte in `EPINFOCLASS` sind weitestgehend selbsterklärend. Mit `EPPing` kann abgefragt werden, ob ein Windows Kern mit integrierter Laufzeitumgebung vorhanden ist. Der Wert `EPRuleResult` wird zur Abfrage neuer Ergebnisse verwendet. Dieser Aufruf blockiert solange bis neuer Ergebnisse vorhanden sind. Mit `EPRuleResultContinue` und `EPRuleResultAbort` wird angezeigt, ob eine von der Laufzeitumgebung blockierte Aktivität fortgesetzt oder abgebrochen werden soll. Die entsprechenden Aufrufe werden durch den Callback-Workerthread im Usermode durchgeführt nachdem die Callback-Funktion aufgerufen wurde.

## 4.5 Reaktion auf erkannte Muster

Die Online-Verarbeitung von Ereignisströmen ist insbesondere dann vorteilhaft, wenn sofort auf erkannte Ereigniskonstellationen reagiert werden kann. Die Reaktion kann dabei in zwei unterschiedlichen Ausführungsdomänen erfolgen: (1) direkt im Kernelmode im Rahmen der Laufzeitumgebung, die auch die Mustererkennung durchführt und (2) im Usermode im Rahmen einer Anwendung, die Callback-Implementierungen mit spezifischen Reaktionen auf erkannte Muster bereitstellt. In diesem Abschnitt werden beide Ansätze vorgestellt.

### 4.5.1 Kernelmode Skripte

Mit einer Erweiterung der vorgestellten Musterspezifikationssprache ist es möglich Reaktionen auf erkannte Muster direkt zusammen mit der Spezifikation der Ereigniskonstellation zu definieren. Dabei können innerhalb eines solchen Skriptes die Informationen über das erkannte Muster und bestimmte, von der Laufzeitumgebung bereit gestellte, Funktionen verwendet werden.

Dieser Ansatz ist vergleichbar mit Skripten, die für *DTrace* definiert werden können: Für im System vorhandene Instrumentierungspunkte kann dort definiert werden, welche Aktionen bei Erreichen des Instrumentierungspunktes ausgeführt werden sollen. Die ausgeführten Aktionen dienen dort vor allem der Datenerfassung und der Verarbeitung von statistischen Informationen. Für den Windows Research Kernel ist ein ähnliches Konzept realisiert worden [98].

Die im Rahmen dieser Arbeit entwickelte Laufzeitumgebung ermöglicht die Ausführung von Aktionen als Reaktion auf erkannte Ereigniskonstellationen. Die dazu verfügbaren Sprachkonstrukte dienen also der Beeinflussung des beobachteten Systems.

Die Ausführung nutzerdefinierter Aktionen im Betriebssystemkernel muss vor allem unter Sicherheitsaspekten betrachtet werden. Das Laden von Regelbeschreibungen, die direkt ausführbare Aktionen enthalten, sollte einem Benutzer nur dann erlaubt sein, wenn er (prinzipiell) auch Treiber oder eine andere Art der Betriebssystemerweiterung installieren darf. In der Regel ist dies nur Systemadministratoren möglich.

In diesem Abschnitt wird eine Spracherweiterung vorgestellt. Die Diskussion wird sich auf die Sprachkonzepte und die Integration in die bereits beschrieben Laufzeitumgebung beschränken. Detaillierte Sicherheitsanalysen sind nicht Gegenstand dieser Arbeit. Weiterhin sind die hier vorgestellten Konzepte nur als Machbarkeitsstudie zu verstehen; für einen umfassenderen Einsatz muss die Zahl der Funktionen erweitert werden.

Die in Tabelle 4.1 dargestellte Musterbeschreibungssprache wird durch die in Tabelle 4.5 dargestellte Grammatik ergänzt.

Die ursprüngliche Grammatik wird um einen weiteren `rulemodifier` ergänzt: in einer Regelbeschreibung wird auch eine `scriptdefinition` akzeptiert. Eine Skriptdefinition wird durch das Schlüsselwort `DO` eingeleitet und enthält anschließend in geschweiften Klammern eine Folge von `scriptstatements`, welche durch Semikolons abgeschlossen werden.

Für ein `scriptstatement` gibt es die folgenden Möglichkeiten: erzeugen eines neuen Ereignis oder aufrufen einer von der Laufzeitumgebung bereitgestellten Funktion.

### Auslösen eines Ereignisses

In einem Skript kann ein weiteres Ereignis angezeigt werden. Dieses Ereignis kann als aggregiertes (oder komplexes) Ereignis betrachtet werden, da es als Resultat eines erkannten Musters auftritt. In einer anderen Musterbeschreibung kann jetzt auf dieses neue Ereignis reagiert werden.

Es können nur Ereignisse ausgelöst werden, die dem Compiler durch eine `EVENTS`-Anweisung bekannt sind. Weiterhin sind Schleifen in der Ereignisverarbeitung nicht zulässig, das heißt, dass in einem Skript kein Ereignis erzeugt werden kann, welches in der zum Skript gehörenden Musterbeschreibung verwendet wird. Zyklen die über mehrere Regeln hinweg entstehen, können erst zur Laufzeit erkannt werden. Die Laufzeitumgebung unterdrückt dann die erzeugten Ereignisse.

Eine alternative Implementierungsstrategie wäre, neu erzeugte Ereignisse in einer gesonderten Ereignisqueue zu sammeln. Bei jedem auftretenden Ereignis wird dann geprüft, ob diese Queue Ereignisse enthält und gegebenenfalls erst diese Ereignisse verarbeitet. Mit diesem Ansatz kann jedoch auf neu erzeugte Ereignisse nicht synchron reagiert werden.

Listing 4.5 zeigt, wie ein Ereignis in einem Skript angezeigt werden kann:

**Listing 4.5:** Skript zum Auslösen eines Ereignisses

```
1 EMIT complexevent (
2     a = ...,
3     b = ...
4 );
```

<table>
<tr><th>Konzept</th><th></th><th>Beschreibung</th></tr>
<tr><td>rulemodifier</td><td>=</td><td>( wheredefinition | whithindefinition |<br>resultdefinition | scriptdefinition )</td></tr>
<tr><td>scriptdefinition</td><td>=</td><td>"DO" "{" scriptstatement { ";" scriptstatement } "}"</td></tr>
<tr><td>scriptstatement</td><td>=</td><td>( eventindication | functioncall )</td></tr>
<tr><td>eventindication</td><td>=</td><td>EMIT <event type name> "("<br>{ eventparameter { "," eventparameter } } ")"</td></tr>
<tr><td>eventparameter</td><td>=</td><td><event field name> "=" value</td></tr>
<tr><td>functioncall</td><td>=</td><td>CALL <function name> "("<br>{ functionparameter { "," functionparameter } } ")"</td></tr>
<tr><td>functionparameter</td><td>=</td><td>( value | threadID | processID )</td></tr>
<tr><td>threadID</td><td>=</td><td>{ <event name> } "#" "TID"</td></tr>
<tr><td>processID</td><td>=</td><td>{ <event name> } "#" "PID"</td></tr>
</table>

**Tabelle 4.5:** Skript-Erweiterung der Musterbeschreibung

| Name | Beschreibung |
| --- | --- |
| SetThreadPriority | Threadpriorität ändern |
| SuspendThread | Thread suspendieren |
| ResumeThread | Thread fortsetzen |
| TerminateThread | Thread beenden |
| TerminateProcess | Prozess beenden |
| DbgMessage | Debug Ausgabe schreiben |

**Tabelle 4.6:** Skript Funktionen

Nach dem Schlüsselwort `EMIT` folgt der Name des auszulösenden Ereignis (`complexevent`). Anschließend können einzelne Felder des Ereignisses mit Werten belegt werden. Die Ereignisfelder sind standardmäßig mit Null-Werten initialisiert, es müssen also nicht alle Ereignisfelder explizit belegt werden.

Die Laufzeitumgebung initialisiert die Header-Informationen, also Zeitstempel und Ausführungskontext, des neuen Ereignis. Der Ausführungskontext des neuen Ereignis ist demnach der Thread, der auch das letzte Ereignis in der Musterbeschreibung angezeigt hat.

### Aufruf einer Funktion

Die Laufzeitumgebung stellt eine Reihe von Funktionen bereit, die von einem Skript verwendet werden können. Nach dem Schlüsselwort `CALL` wird der Funktionsname angegeben - anschließend folgen falls erforderlich eine Reihe von Funktionsparametern.

Eine Übersicht der im aktuellen Prototyp implementierten Funktionen ist in Tabelle 4.6 dargestellt.

Die möglichen Parameter der einzelnen Funktionen sind jeweils eine Thread- beziehungsweise eine Prozess-ID. Für `SetThreadPriority` wird außerdem noch ein Prioritätswert übergeben. Mit `DbgMessage` kann eine (statische) Zeichenkette auf der Debug Konsole ausgegeben werden.

Ganz allgemein sind nach der Grammatik als Funktionsparameter `value`- und Ausführungskontext-Konstrukte möglich. Mit `value` kann mit Hilfe von im Muster erkannten Ereignisdatenfeldern der Parameterwert errechnet werden.

Bei der Übergabe von Informationen zum Ausführungskontext müssen Thread- und Prozess-ID unterschieden werden. Für ein (benanntes) Ereignis `a` kann mit `a # TID` auf die Thread-ID und mit `a # PID` auf die Prozess-ID zugegriffen werden. Alternativ kann mit `# TID` und `# PID` auf die entsprechenden IDs des Ausführungskontexts des Ereignisses, welches zur erfolgreichen Erkennung der Ereigniskonstellation führte, zugegriffen werden.

Der Compiler prüft bei der Übersetzung des Skriptes die Typen der Parameter. Zusätzlich kann die Laufzeitumgebung (optional) zur Laufzeit die Parameter prüfen.

### 4.5.2 Usermode Callbacks

Um innerhalb einer Anwendung die Mustererkennung von Ereignisströmen verwenden zu können, wurde eine Usermode-Bibliothek entwickelt. Die bereitgestellten Funktionen und das generelle Programmiermodell werden nachfolgend vorgestellt.

Die Grundidee ist, dass Anwendungen Regelbeschreibungen in kompilierter Form laden können. Anschließend können einzelne Regeln aktiviert und deaktiviert werden. Zu jeder Regel kann ein Callback registriert werden. Wird ein entsprechendes Muster erkannt, wird der Callback aufgerufen und die Anwendung kann auf das erkannte Muster reagieren.

Bei synchronen Regeln kann der Callback durch seinen Rückgabewert anzeigen, ob die Aktivität, die das letzte Ereignis im Muster ausgelöst hat, fortgesetzt oder abgebrochen werden soll. Bei asynchron verarbeiteten Regeln ist der Rückgabewert des Callbacks nicht relevant.

Mit diesem Programmiermodell können Anwendungen implementiert werden, die adaptiv auf Vorgänge im System reagieren: tritt ein definiertes Muster auf, kann die Anwendungskonfiguration geändert oder eine andere beliebige Aktion ausgeführt werden.

Auch bei der Implementierung des Callback-Konzeptes war das Ziel die Machbarkeit eines solchen Ansatzes zu demonstrieren. Für den Einsatz in Produktivumgebungen sind noch eine Reihe weiterer Aspekte zu berücksichtigen:

**Ausführungsdauer von Callbacks:** Der als Reaktion ausgeführte Programmcode im Callback ist dem Laufzeitsystem unbekannt. Dadurch kann insbesondere die Ausführungszeit nicht abgeschätzt werden. Durch fehlerhaft implementierte (oder bösartige) Callbacks kann das Systemverhalten gestört werden, beispielsweise können Systemthreads blockiert werden.

**Beeinflussung von Threads in anderen Prozessen:** In einer Musterbeschreibung können beliebige Ereignisse aus verschiedenen Ausführungskontexten verwendet werden. Durch entsprechende Muster können demnach Vorgänge in beliebigen Threads im System beschrieben und überwacht werden.

Die dargestellten Sicherheitsaspekte können durch Anpassung/Einschränkung des Compilers oder der Laufzeitumgebung berücksichtigt werden.

Im Compiler kann die Art der verwendbaren Ereignisse eingeschränkt werden. Weiterhin ist ein Kennzeichnung bestimmter Ereignisfelder als sicherheitsrelevant möglich. Solche Felder können dann nicht als Rückgabewerte oder als Parameter in Reaktionsskripten verwendet werden.

Eine Möglichkeit zur Absicherung der Laufzeitumgebung ist die Einschränkung der von einer Anwendung ladbaren Regeln - indem beispielsweise nur Regeln erlaubt werden, die ausschließlich anwendungsspezifische Ereignisse verarbeiten.

Die Einrichtung von Sicherungsmechanismen schränkt in jedem Fall die Flexibilität des Konzepts bezüglich Ausdrucksmächtigkeit und Anwendungsmöglichketen ein. In der vorliegenden Implementierung ist eine Sicherung ausschließlich über die Ausführungsrechte der entsprechenden Werkzeuge möglich.

Nachfolgend werden die Funktionen der Usermode-Bibliothek zur Verwendung von Callbacks vorgestellt. Listing 4.6 zeigt eine Übersicht der verfügbaren Funktionen.

**Listing 4.6:** Reaktion auf erkannte Muster - Usermode API

```
1  BOOLEAN
2  EPInstalled(
3  );
4
5  EPSTATE
6  RegisterRule(
7      PCHAR FileName,
8      RuleResultCallback cb,
9      PEPHANDLE pResultHandle
10 );
11
12 EPSTATE
13 UnregisterRule(
14     EPHANDLE Handle
15 );
16
17 EPSTATE
18 ActivateRule(
19     EPHANDLE Handle
20 );
21
22 EPSTATE
23 DeactivateRule(
24     EPHANDLE Handle
25 );
```

Die Funktion `EPInstalled` prüft, ob auf dem System ein Kernel ausgeführt wird, der den Systemaufruf `NtQueryInformationEP` bereit stellt. Ist dies der Fall, wird `TRUE` zurückgegeben. Mit Hilfe von `EPInstalled` kann also überprüft werden, ob die Laufzeitumgebung zur Verarbeitung von Ereignisströmen im Kernel verfügbar ist.

Die weiteren Funktionen dienen der Verwaltung des Lebenszyklus einer Regel im System. Eine Regel kann dabei die Zustände „aktiviert" und „deaktiviert" einnehmen.

Zuerst muss eine Regel bei der Laufzeitumgebung registriert werden. Dazu wird die Funktion `RegisterRule` verwendet. Als Parameter wird der Pfad (`FileName`) zur Binärcode-Datei der zu ladenden Regel übergeben. Zusätzlich wird eine Callback-Funktion vom Typ `RuleResultCallback` übergeben. Dieser Callback wird aufgerufen, wenn das Muster der geladenen Regel erkannt wird. Der Parameter `pResultHandle` erhält als Rückgabewert der Funktion eine Referenz auf die registrierte Regel. Diese Referenz muss dann bei den nachfolgend beschriebenen Funktionen mit angegeben werden.

Nach dem Registrieren von Regel und Callback ist die Regel im Zustand „deaktiviert" und muss explizit mit `ActivateRule` aktiviert werden. Anschließend ist die Regel „aktiviert" und wird von der Laufzeitumgebung im Kernel bei der Verarbeitung von Ereignissen berücksichtigt. Mit `DeactivateRule` kann eine aktive Regel wieder deaktiviert werden und mit `UnregisterRule` wieder vollständig aus der Laufzeitumgebung entfernt werden.

Jede der Usermode-API Funktionen zur Verwaltung von Regeln liefert als Rückgabewert einen `EPSTATE`. Dieser Zustandswert zeigt eventuell aufgetretene Fehler an. Datentyp, Parameter und Rückgabewertebereich von Callback-Funktionen sind im Listing 4.7 dargestellt.

**Listing 4.7:** Callback-Definitionen

```
typedef enum _CBRESULT {
    ABORT,
    CONTINUE
} CBRESULT;

typedef struct _EXECCONTEXT {
    UCHAR CpuId;
    HANDLE ProcessId;
    HANDLE ThreadId;
} EXECCONTEXT, *PEXECCONTEXT;

typedef CBRESULT(*RuleResultCallback)(
    PCHAR ResultBuffer,
    ULONG ResultSize,
    PEXECUTIONCONTEXT ExecutionContext
);
```

Die Aufzählung `CBRESULT` beschreibt mögliche Rückgabewerte einer Callback-Funktion. Gibt ein Callback `ABORT` zurück, soll die musterauslösende Aktivität abgebrochen und mit `CONTINUE` fortgesetzt werden. Dies gilt jedoch nur für synchron verarbeitete Regeln. Bei asynchron verarbeiteten Regeln wird der Rückgabewert von der Laufzeitumgebung ignoriert. Die Struktur `EXECCONTEXT` beschreibt einen Ausführungskontext. Die Informationen werden aus dem Ereignisheader des Ereignisses kopiert, das zu einer Erkennung des Musters führte.

Mit `RuleResultCallback` wird eine Funktionssignatur definiert, die von Callback-Funktionen verwendet werden muss: Der Rückgabedatentype ist `CBRESULT`, die Parameter sind Ergebnispuffer, Ergebnisgröße und Ausführungskontext. Die Ergebnisdaten werden von der Laufzeitumgebung initialisiert und an die Callback-Implementierung übergeben.

Ein einfaches Beispiel für die Registrierung eines Callbacks ist im Listing 4.8 dargestellt.

**Listing 4.8:** Beispiel: Registrieren eines Callbacks für eine Regel

```
#include <eplib.h>

CBRESULT resultCb (
    PCHAR ResultBuffer,
    ULONG ResultSize,
    PEXECUTIONCONTEXT ExecutionContext
)
{
  // .. callback implementation
  return CONTINUE;
}

main()
{
  EPHANDLE h;
  RegisterRule("rule.epcc", resultCb, &h);
  // .. application code
  UnregisterRule(h);
}
```

Mit dem Aufruf von `RegisterRule` wird die (kompilierte) Regel aus der Datei `rule.epcc` geladen und für Ergebnisse dieser Regel der Callback `resultCb` registriert. Als eindeutige

Identifikation der geladenen Regel wird das Handle `h` verwendet. Das Handle muss immer angegeben werden, wenn Aktionen bezüglich der geladenen Regel ausgeführt werden sollen.

In der Callback-Implementierung `resultCb` kann jetzt auf die Ergebnisdaten und den Ausführungskontext zugegriffen werden. Der Compiler generiert zu einer übersetzten Regel eine C-Datenstruktur, die Ergebnisdaten beschreibt. Der `ResultBuffer` kann mit Hilfe dieser Datenstruktur interpretiert werden. Werden Informationen variabler Größe zurückgegeben, beschreibt `ResultSize` die Größe der Datenmenge.

Der dargestellte Callback-Mechanismus wird mit Hilfe von Usermode-Workerthreads implementiert. Für jede geladene Regel wird ein solcher Thread gestartet. Dieser übernimmt die Kommunikation mit der Laufzeitumgebung im Kern, lädt die Regel, verwaltet den Speicher für die Ergebnisse, wartet auf Ergebnisse und ruft die Callback-Implementierung. Auf diese Weise wird die Regelverarbeitung asynchron zu den restlichen Anwendungsaktivitäten durchgeführt.

Ein wichtiger Aspekt ist dabei, dass der erzeugte Workerthread *keine* Ereignisse im Betriebssystemkern verursacht - jeder Thread, der `NtSetInformationEP` mit `EPLoadRule` aufruft, wird entsprechend markiert. So wird verhindert, dass zyklische Abhängigkeiten zwischen der Erkennung einer Ereigniskonstellation und der Reaktion auf ein erkanntes Muster entstehen.

## 4.6 Werkzeuge

Zur online Verarbeitung von Ereignisströmen sind die wichtigsten Werkzeuge der vorgestellte Compiler und die Laufzeitumgebung.

Die Steuerung der aufgezeichneten Ereignisse erfolgt mit Hilfe der WMK-Werkzeuge - das heißt es kann weiterhin ein zu beobachtender Prozess explizit gestartet werden, oder (global) alle auftretenden Ereignisse verarbeitet werden.

Weiterhin ist ein Werkzeug implementiert worden, welches mit Hilfe der in Abschnitt 4.4.5 beschriebenen Systemaufrufschnittstelle mit der Laufzeitumgebung im Kernel kommuniziert und es ermöglicht einzelne Regeln zu testen. Eine Regel wird damit geladen und eine einfache Callback-Implementierung gibt die Ergebnisse auf der Konsole aus.

Das Testen der implementierten Laufzeitumgebung ist aufwändig, wenn für jede neue Version das Testsystem neu gestartet werden muss. Aus diesem Grund ist eine Usermode-Variante der Laufzeitumgebung erstellt worden. Durch die Verwendung von C-Preprozessor Makros ist es möglich die Implementierung der Laufzeitumgebung im Usermode zu testen und den gleichen Code dann für die Kernel-integrierte Variante zu verwenden.

Die Usermode-Variante der Laufzeitumgebung liest die auftretenden Ereignisse aus einer WMK Logdatei. Dieses Verfahren bietet beim Testen von Laufzeitumgebung und Regeln die folgenden Vorteile gegenüber der in den Betriebssystemkern integrierten Variante:

**Deterministisches Verhalten:** Die Ereignisse werden stets in der durch die Logdatei vorgegebenen Reihenfolge verarbeitet. Dies ermöglicht die Überprüfung von optimierenden Heuristiken basierend auf identischen Ereignisströmen.

**Maximale Ereignisrate:** Die Laufzeitumgebung kann sofort das nächste Ereignis verarbeiten, wenn das vorhergehende abgeschlossen wurde. Auf diese Weise kann die maximale Verarbeitungsgeschwindigkeit - für eine spezifische Regel und einen spezifischen Ereignisstrom - bestimmt werden.

**Analyse von Regeln und Laufzeitumgebung:** Regeln können auf einfache Weise getestet und mit einem konkreten Ereignisstrom überprüft werden. Für die Betriebssystemkern-integrierte Variante muss für jeden Test eine entsprechende Last erzeugt werden.

Die Implementierung der Systemaufrufschnittstelle und der Synchronisation von Ereignisaufzeichnung und -verarbeitung kann jedoch nur mit der im Betriebssystemkern integrierten Laufzeitumgebung getestet werden. Die Usermode-Variante ist ein hilfreiches Werkzeug zur Fehlersuche.

Als Fallstudie wird in Abschnitt 5.6 ein Verfahren zur Komplexitätsanalyse der Regelverarbeitung basierend auf absorbierenden Markov-Ketten vorgestellt. Zur Durchführung der beschriebenen Analyse ist ebenfalls ein Werkzeug entwickelt worden.

## 4.7 Evaluation

In diesem Abschnitt werden Evaluierungsergebnisse für die entwickelte Laufzeitumgebung beschrieben. Dabei stehen Mikro-Benchmarks und einfache Tests im Vordergrund. Die Anwendbarkeit der Konzepte im Rahmen von Fallstudien wird im Kapitel 5 dargestellt.

Der Compiler wurde getestet, indem verschiedene praktische und künstliche Beispiele übersetzt worden sind. Durch Prüfung des entstehenden Automaten wurde sichergestellt, dass die Umwandlung funktioniert. Die Geschwindigkeit des Übersetzungsprozess selbst war kein Optimierungskriterium. Die Skripte sind im Allgemeinen klein und werden in wenigen Sekunden übersetzt.

Als Testumgebung wurde eine Intel Core2 Duo Workstation (3 GHz, 4 GByte RAM, Windows 7) verwendet. Für die dargestellten Tests wurde die Laufzeitumgebung in ihrer Usermode-Variante verwendet. Dies ermöglicht wiederholte Testdurchläufe mit deterministischen Eingabedaten.

### 4.7.1 Messungen und Analysen

Für die Evaluierung wurden die in Listing 4.9 dargestellten Testskripte verwendet. Als Analyseziel sollte der Durchsatz der Laufzeitumgebung und die Zeiten für die Prüfung von Bedingungen beziehungsweise die Durchführung von Aktionen ermittelt werden.

**Listing 4.9:** Testskripte

```
1  RULE syscall
2    PATTERN { [syscall:a] }
3    RETURN { a.SyscallNr, a.TimeStamp }
4
5  RULE nosyscallexit
6    PATTERN { [syscall:a, ~(syscallexit|threadtermination|waitevent), ⤸
       syscall] }
7    WHERE { [ProcessId],
8            [ThreadId],
9            a.SyscallNr < 300 }
10   RETURN { a.SyscallNr, a.TimeStamp }
11
12 RULE longsyscalls
13   PATTERN { [syscall:a, syscallexit:b] }
```

```
14  WHERE { [ProcessId],
15          [ThreadId],
16          b.TimeStamp - a.TimeStamp > 2000000,
17          a.SyscallNr < 300 }
18  RETURN { a.SyscallNr, a.TimeStamp, b.TimeStamp }
```

Mit allen drei Skripten wurde die gleiche Logdatei analysiert. Die Logdatei enthielt dabei auch alle weiteren vom Windows Monitoring Kernel zur Verfügung gestellten Ereignistypen - also nicht nur die in den Skripten verwendeten Typen. Insgesamt enthielt die Logdatei 1,2 Mio. Ereignisse. Diese Ereignisse wurden direkt nacheinander in der Laufzeitumgebung verarbeitet.

Die Anzahl der jeweils erkannten Konstellationen und der Durchsatz sind in Tabelle 4.7 dargestellt.

Erwartungsgemäß variiert der Durchsatz mit der Komplexität der zu erkennenden Konstellationen. Je nach beschriebenem Muster sind pro Laufzeitzustand unterschiedlich viele Bedingungen zu prüfen und Aktionen durchzuführen. Auch die Anzahl der Replikationsübergänge hat Einfluss auf den Gesamtdurchsatz.

Beim Vergleich mit den Zeiten, die für die Aufzeichnung von Ereignissen mit dem WMK benötigt werden (siehe Abschnitt 3.5), ergibt sich, dass die Analyse zusätzlichen Aufwand verursacht - beispielsweise ist der Durchsatz bei der Erkennung von `nosyscallexit` geringer als die Rate mit der `syscall`-Ereignisse im System auftreten.

Zusätzlicher Aufwand ergibt sich aus der Prüfung von Bedingungen und der Durchführung von Aktionen bei Zustandsübergängen im Laufzeitzustand. In Abbildung 4.8 ist ein Histogramm der Dauer von Bedingungsprüfungen abgebildet.

Die Prüfung einer Bedingung erfordert in der Regel weniger Zeit als die Aufzeichnung eines Ereignisses mit dem WMK (siehe Abbildung 3.2) - allerdings sind pro Ereignis unter Umständen mehrere Bedingungen zu prüfen.

Die Implementierung der Prüfung von Bedingungen erfolgt in dem erstellten Prototypen durch eine Tabelle von Funktionspointern: für die unterstützten Datentypen ist für jeden Relationsoperator eine Funktion implementiert, die den entsprechenden Vergleich durchführt. Zur Laufzeit kann dann mit zwei Array-Indizes (für Datentyp und Operation) die entsprechende Funktion gerufen werden. Als Parameter werden jeweils `char`-Pointer übergeben, die dann entsprechend dereferenziert und gecastet werden.

Die Implementierungsstrategie kann noch verbessert werden indem beispielsweise *memory alignment*-Effekte und Caching berücksichtigt werden.

Zusammenfassend lässt sich feststellen, dass die Laufzeitumgebung zur Analyse verwendet werden kann. Die vorgestellten Heuristiken können zu einer weiteren Verbesserung des Durchsatz führen. Die Analyse von Konstellationen, die Ereignisse mit hoher Häufigkeit enthalten,

| **Testskript** | **Erkannte Muster** | **Durchsatz** [Ereignisse/s] |
|---|---:|---:|
| `syscall` | 420951 | 130269 |
| `nosyscallexit` | 1817 | 12630 |
| `longsyscalls` | 6621 | 58042 |

**Tabelle 4.7:** Skript Funktionen

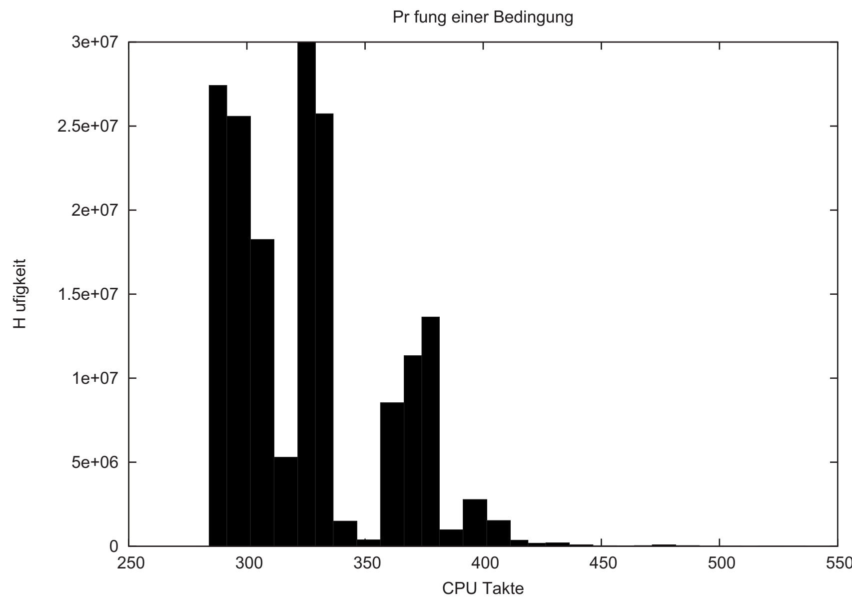



**Abbildung 4.8:** Analyse: Dauer von Bedingungsprüfungen

verursacht eine entsprechend hohe Systembeeinflussung durch die Laufzeitumgebung. In Abschnitt 5.6 wird ein Verfahren vorgestellt, mit dem (unter bestimmten Rahmenbedingungen) der Einfluss der Laufzeitumgebung auf das Gesamtsystem abgeschätzt werden kann.

### 4.7.2 Vergleich mit alternativen Systemen

Die Online-Verarbeitung von Ereignisströmen im Betriebssystem in der hier vorgestellten Form ist neu. Das abstrakte Problem der Mustererkennung in einem Strom von Ereignissen tritt auch in anderen Zusammenhängen auf und wird in Kapitel 6 diskutiert.

Ein bekanntes System zur dynamischen Betriebssysteminstrumentierung ist DTrace [35] für die Solaris Plattform. Auch DTrace bietet die Möglichkeit Skripte als Reaktion auf erkannte Ereignisse auszuführen. Nachfolgend soll die im Rahmen dieser Arbeit entwickelte Laufzeitumgebung zu DTrace abgegrenzt werden. Die wesentlichen Unterschiede sind:

- Das Ereignis, auf das reagiert werden soll, wird in DTrace durch Angabe eines Ereignisproviders, eines Moduls, eines Funktionsnames und eines Ereignisnamens beschrieben - jeweils getrennt durch einen Doppelpunkt, also `provider:modul:funktion:name`. Die Notation `syscall::kill:entry` beschreibt beispielsweise das `entry` Ereignis der Funktion `kill` des Ereignisproviders `syscall`. Durch Platzhalter-Symbole (`*`) beziehungsweise durch weglassen bestimmter Elemente können alle Module oder alle Ereignisse eines Providers ausgewählt werden. Zu einer solchen Ereignisbeschreibung kann spezifiziert werden, welche Operationen ausgeführt werden sollen, wenn das beschriebene Ereignis auftritt.

  Die im Rahmen dieser Arbeit entwickelte Laufzeitumgebung erlaubt die Spezifikation von *Ereignismustern*, mit Ereignissen aus *mehreren Ausführungskontexten* und stellt damit eine Erweiterung des DTrace-Ansatzes dar.

- Die wichtigsten Anwendungsfälle von DTrace sind die Anzeige von Ereignisketten und die Aggregation von Statistiken während der Ausführung. Die Skriptsprache ist auf diese

Anwendungsfälle ausgerichtet. Ein DTrace-Skript kann mit Hilfe des *destructive mode* von DTrace auch beliebige Shell-Skripte als Reaktion auf erkannte Ereignisse ausführen.

Das Ziel der hier vorgestellten Laufzeitumgebung ist, eine Umgebung bereitzustellen die sowohl die Analyse von Vorgängen im System ermöglicht (ähnlich zu DTrace), als auch eine Programmierschnittstelle für die sofortige Reaktion auf erkannte Ereignisse bereitzustellen. Die Laufzeitumgebung ermöglicht eine synchrone Reaktion im Ausführungskontext der erkannten Ereigniskonstellation und kann damit als Grundlage für selbstadaptive Anwendungen dienen.

DTrace ist als Werkzeug zur Analyse von produktiv eingesetzten Systemen entwickelt worden - Sicherheit in der Anwendung und geringe Systembeeinflussung waren daher wichtige Anforderungen. Die hier vorgestellte Laufzeitumgebung ist konzeptionell eine Ergänzung des DTrace-Ansatzes und soll die sofortige Reaktion auf erkannte Ereigniskonstellationen ermöglichen. Für die Verwendung in produktiv eingesetzten System sind Erweiterungen am entwickelten Prototypen notwendig.

## 4.8 Zusammenfassung

In diesem Kapitel wurde die entwickelte Laufzeitumgebung zur online-Verarbeitung von Ereignissen im Betriebssystemkern vorgestellt. Neben einer Sprache zur Spezifikation von Mustern in Ereignisströmen wurde ein auf deterministischen, endlichen Automaten basierendes Abarbeitungsmodell und eine entsprechende Laufzeitumgebung entwickelt. Die Laufzeitumgebung erlaubt die sofortige Reaktion auf erkannte Regeln - entweder durch Ausführung von Skripten direkt im Betriebssystemkern oder durch Aufrufen von anwendungsspezifischen Callbacks im Usermode.

# Kapitel 5

# Fallstudien und Leistungsbewertung

*In diesem Kapitel werden durchgeführte Tests und Fallstudien dokumentiert. Diese zeigen Einsatzmöglichkeiten und Grenzen der entwickelten Konzepte - des Windows Monitoring Kernels (WMK) und der Laufzeitumgebung zur online Verarbeitung von Ereignisströmen.*

## 5.1 Einleitung

Ziel und Kern der vorliegenden Arbeit ist die Bereitstellung einer Infrastruktur zur effizienten Verarbeitung von Ereignissen im Betriebssystemkern. Im Kapitel 3 wurde dazu ein Instrumentierungsansatz für den Windows Kern und im Kapitel 4 eine Laufzeitumgebung für die Verarbeitung von Ereignisströmen dargestellt.

Im Abschnitt 1.3 sind bereits verschiedene Anwendungsszenarien für die entwickelten Konzepte kurz beschrieben worden. Nachfolgend werden jetzt Fallstudien vorgestellt, die einzelne Aspekte der identifizierten Anwendungsszenarien realisieren und überprüfen.

Die Fallstudien zeigen (1) die Verwendung des WMK im Bereich Monitoring/Tracing von Softwaresystemen, (2) die Anwendung der Laufzeitumgebung zur sofortigen Verarbeitung von Ereignisketten und (3) verbindende Aspekte zwischen beiden Anwendungsgebieten.

## 5.2 Analyse der Bearbeitung von Anfragen an einen Webserver

Der WMK kann zur Analyse von Anwendungseigenschaften verwendet werden. Für den Betrieb eines Webservers ist interessant, wie Anfragen verarbeitet werden und welche Systemressourcen dabei benötigt werden. Ziel der in diesem Abschnitt vorgestellten Fallstudie war es, den Weg einer HTTP-Anfrage durch einen Apache Webserver [1] nachvollziehen zu können.

### 5.2.1 Instrumentierte Ausführung

Der WMK ermöglicht die Tracing-basierte Systemanalyse. Das Vorgehen ist dabei immer, dass ein in der Regel unmodifiziertes Programm mit einem WMK Kernel ausgeführt und nach instrumentierter Ausführung die Logdatei analysiert wird.

Der WMK wurde auf einer Workstation (Pentium 4 CPU, 2.66 GHz, 768 MByte RAM) ausgeführt. Der Apache Webserver wurde in der Version 2.2.4 für Windows neu übersetzt. Dabei wurde in der Datei `child.c` die Funktion `worker_main` modifiziert: der Apache wurde als Ereignisprovider beim WMK registriert und jeweils ein Ereignis zeigt den Beginn und das Ende einer Anfragebearbeitung an. Insgesamt sind drei Codezeilen hinzugefügt worden.

Aufwändiger war es, die Windows Socket API zu instrumentieren. Dazu wurde von den Systembibliotheken `wsock32.dll` und `ws2_32.dll` jeweils eine Proxy-Variante erstellt. Eine solche Proxy-DLL stellt die gleiche Schnittstelle zur Verfügung, lädt die ursprüngliche DLL und implementiert alle bereitgestellten Funktionen durch Aufrufen der original Funktionen. Weiterhin sind die wichtigsten Funktionen instrumentiert, so dass sie Ereignisse im WMK erzeugen.

Zur Lasterzeugung wurde `httperf`[7] auf einem leistungsfähigen Rechner (4x Xeon CPU, 3.0 GHz, 2 GByte RAM) ausgeführt. Auf diese Weise ist sichergestellt, dass die Lasterzeugung bis zur vollen Auslastung des Servers möglich ist. Als Test wurde eine statische HTML Seite (500 Byte) mit unterschiedlichen Anfrageraten abgerufen.

### 5.2.2 Ergebnisse

Mit `httperf` werden für einen definierten Zeitraum eine bestimmte Anzahl von Anfragen pro Sekunde erzeugt und anschließend die Anzahl der Antworten pro Sekunde gemessen. Ist der angefragte HTTP Server nicht überlastet, sollte die Anzahl von gesendeten und empfangenen Anfragen pro Sekunde identisch sein. Das Ergebnis für den beschriebenen Versuchsaufbau ist in Abbildung 5.1 dargestellt.

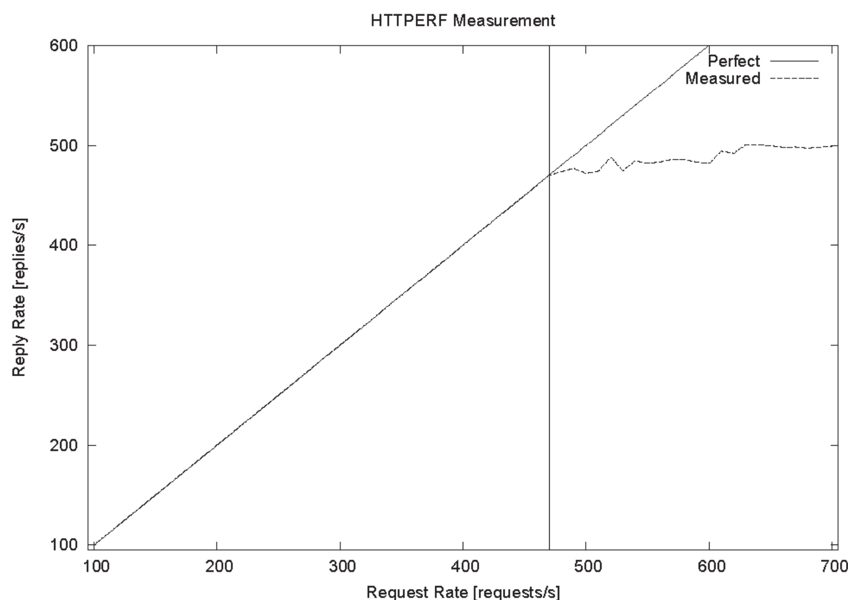



**Abbildung 5.1:** Apache/WMK: httperf Ergebnisse

Es ist zu erkennen, dass bis zu einer Anfragerate von 470 Anfragen pro Sekunde das getestete System nicht überlastet ist und mit der gleichen Rate antwortet. Bei einer höheren Anzahl von Anfragen kann nur mit einer Rate von weiterhin etwa 470 Antworten pro Sekunde geantwortet werden.

Mit Hilfe des WMKs kann jetzt versucht werden, die Ursache für das gesuchte Verhalten zu finden. In der Logdatei können alle Anfragen eindeutig identifiziert werden. Tritt im Kontext eines Threads ein Anfragestart-Ereignis auf, bearbeitet der Thread diese Anfrage bis ein Anfrageende-Ereignis auftritt.

Durch Analyse der Logdatei kann bestimmt werden, wie viele Anfragen gleichzeitig im System verarbeitet werden. Das Ergebnis ist in Abbildung 5.2 dargestellt.

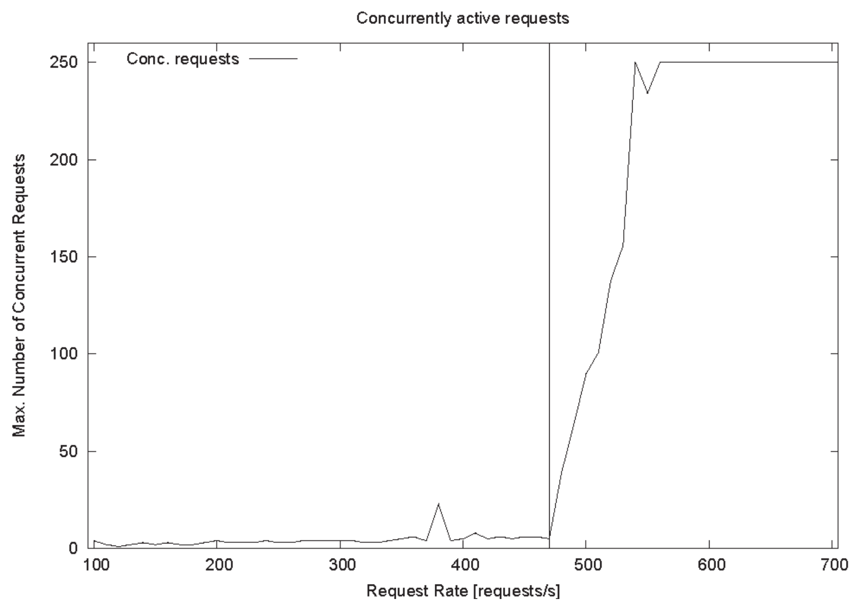



**Abbildung 5.2:** Apache/WMK: Anzahl der gleichzeitig bearbeiteten Anfragen

Bis zu der bestimmten Grenze von 470 Anfragen pro Sekunde werden nur wenige Anfragen gleichzeitig bearbeitet. Danach steigt die Zahl der parallel verarbeiteten Anfragen auf 250 und verbleibt dann bei diesem Wert.

Der Wert 250 ergibt sich aus der Konfiguration des Apache Webservers. Ein Eintrag in der Konfigurationsdatei bestimmt, dass maximal 250 Apache Prozesse zur Bearbeitung von Anfragen erzeugt werden sollen. Genau diese Grenze zeigt sich in der Darstellung.

Weiterhin ist zu sehen, dass die Überlastung nicht durch die Anzahl der gleichzeitig verarbeiteten Anfragen verursacht werden kann. Bis zu dem Punkt der Überlast ist diese Anzahl relativ konstant niedrig. Eine weitere mögliche Ursache ist die Überlastung der CPU. Der Verlauf der CPU Last in Abhängigkeit von der Anfragerate ist in Abbildung 5.3 dargestellt.

Es ist zu erkennen, dass auch dies nicht der Grund sein kann: die CPU Auslastung steigt linear mit der Anfragerate - bei der bestimmten Lastgrenze erreicht sie jedoch nur etwa 35%.

Eine Besonderheit der dargestellten Werte der CPU Auslastung ist, dass diese nicht direkt während der Ausführung der Tests gemessen wurde, sondern aus der Logdatei rekonstruiert wurde: Die Auslastung ergibt sich aus der Zeit in der ein (beliebiger) Thread des Webservers eine Anfrage bearbeitet - sich also zwischen einem Anfragestart- und einem Anfrageende-Ereignis befindet - und der Gesamtzeit des Tests.

Mit diesem Ansatz lassen sich auch für eine einzelne Anfrage die Anteile von Rechenzeit und Wartezeit an der Gesamtbearbeitungszeit bestimmen. Die Gesamtbearbeitungszeit ergibt sich aus den Zeitpunkten von Anfragestart- und Anfrageende-Ereignis. Die Gesamtrechenzeit kann

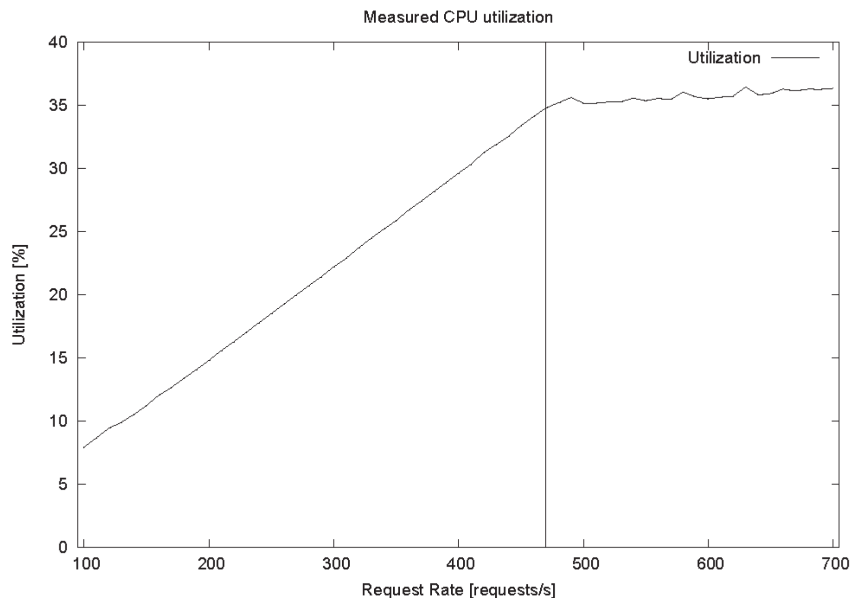



**Abbildung 5.3:** Apache/WMK: CPU Auslastung

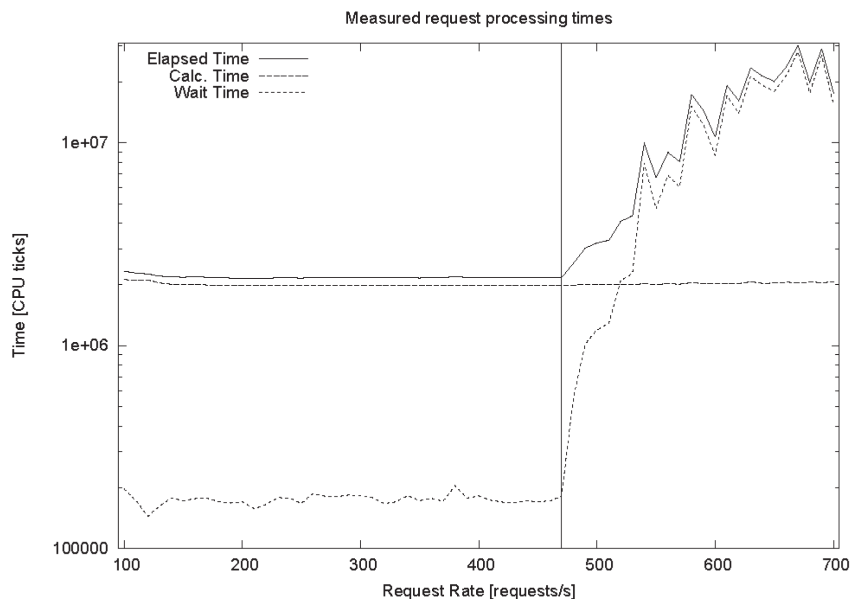



**Abbildung 5.4:** Apache/WMK: Zeiten für die Anfragebearbeitung

mit Hilfe der Kontextwechsel-Ereignisse dazwischen rekonstruiert werden. Die entsprechenden Anteile sind in Abbildung 5.4 in Abhängigkeit von der Anfragerate dargestellt.

Die Rechenzeit die zur Bearbeitung der Anfragen benötigt wird, ist konstant und unabhängig von der Anzahl der Anfragen pro Sekunde. Damit lässt sich der lineare Verlauf der CPU Auslastung (siehe Abbildung 5.3) erklären. Bei höheren Anfrageraten steigt die Anzahl der gleichzeitig aktiven Threads und damit die Wartezeit bei der Verarbeitung einer einzelnen Anfrage.

### 5.2.3 Diskussion

Die Ursache der Überlastung bei mehr als 470 Anfragen pro Sekunde liegt außerhalb des mit dem WMK beobachtbaren Systems: die Netzwerkpakete der versendeten Anfragen erreichen das System gar nicht und können daher auch nicht mit Hilfe des WMK analysiert werden.

Mit Hilfe des WMK können, wie gezeigt, Informationen über die durch Anfragen entstehende Last im System bestimmt werden - für einzelne Anfragen und unabhängig von der Gesamtlast des Systems. Werden auch Ein- und Ausgabeoperationen als Ereignisse berücksichtigt, ist auch die verursachte Last beim Festplatten- oder Netzwerkzugriff erfassbar.

Im beschriebenen Test werden alle Anfragen vollständig in einem Thread verarbeitet und damit die Analyse vereinfacht. Sind mehrere Threads beteiligt, ist die Verfolgung einzelner Anfragen komplexer: durch die Analyse der Verwendung von Synchronisationsobjekten sind Abhängigkeiten zwischen mehreren Threads identifizierbar. Kommunizieren mehrere Threads über einen gemeinsamen Speicherbereich ist auch diese Art der Analyse nicht durchführbar. Lösungsansätze sind in [38] und [65] dargestellt.

## 5.3 Analyse von Vorgängen im Betriebssystemkern

Ein weiteres Einsatzgebiet des Windows Monitoring Kernels ist die Analyse von Abläufen und Strukturen des Windows Betriebssystemkerns. Für ein gutes Verständnis von Vorgängen im Kern sind - neben der Quelltextanalyse - auch die Untersuchung dynamischer Aspekte hilfreich. Der WMK kann zur dynamischen Analyse verschiedenster Vorgänge verwendet werden. Nachfolgend werden zwei Beispiele kurz dargestellt.

### 5.3.1 Bootvorgang

Das Windows Betriebssystem besteht aus zahlreichen Komponenten und Diensten, die beim Starten des Systems initialisiert und konfiguriert werden müssen. Die Analyse des Bootvorgangs kann Informationen über die genauen Abläufe und beteiligten Komponenten liefern.

Die WMK Infrastruktur wird zusammen mit der Initialisierung des Ein-/Ausgabesystems eingerichtet und steht damit sehr früh im Bootprozess zur Verfügung. Betrachtet man nur die erzeugten Prozesse und Threads sowie deren Beziehung, kann ein Eindruck über die Abläufe beim Systemstart gewonnen werden. Ein Ausschnitt aus dem entstehenden Graph ist in Abbildung 5.5 dargestellt.

Prozesse sind als Ovale und Threads als Rechtecke dargestellt. Zu jedem Prozess wird die Prozess-ID und die dazugehörige ausführbare Datei dargestellt. Zu jedem Thread wird dementsprechend die Thread-ID, die vom Thread ausgeführte Funktion und die dazugehörige Binärdatei dargestellt. Eine Verbindung zwischen zwei Elementen geht vom erzeugenden Element zum erzeugten Element.

In der statischen Darstellung geht allerdings die eigentliche Dynamik des Startvorgangs verloren. Das heißt, die Reihenfolge, in der Threads und Prozesse erzeugt wurden, ist nicht mehr direkt ersichtlich, kann aber der Logdatei entnommen werden.

Nachdem Starten eines Windows Server 2003 Systems und der Anmeldung eines Benutzers können mit Hilfe des WMK unter anderem die folgenden Aktivitäten beobachtet werden:

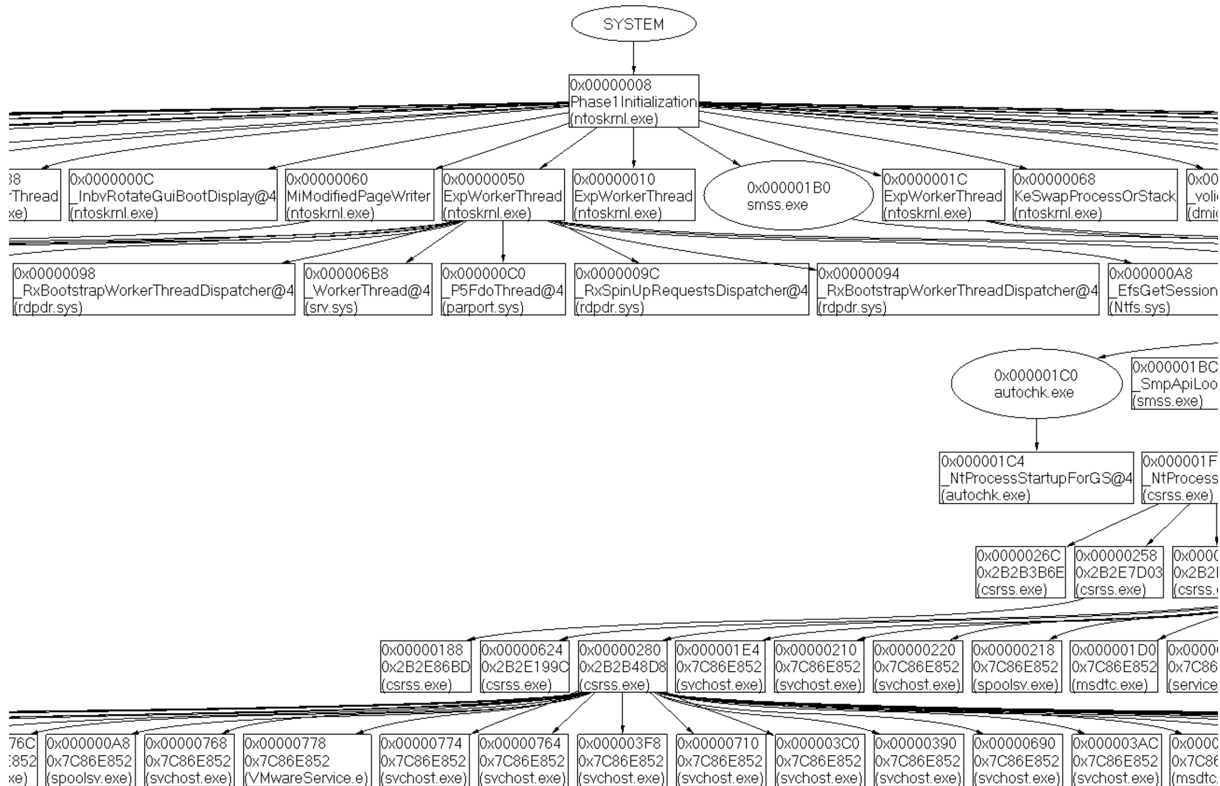



**Abbildung 5.5:** Bootvorgang: Prozesse und Threads

- Der erste erzeugte Thread führt die Funktion `Phase1Initialization` aus. Tatsächlich ist diese Funktion ein guter Startpunkt für die Analyse des Startvorgangs. Ausgehend von dieser Funktion werden die Initialisierungsfunktionen aller Kernelmodule aufgerufen, die dann zu den nächsten gestarteten Threads führen.

- Einige Verwaltungaufgaben des Windows Kerns werden in eigenen Threads ausgeführt, beispielsweise der `KeBalanceSetManager` oder der `MiModifiedPageWriter`. Weiterhin verwendet der Windows Kern eine Menge von `ExpWorkerThread`s, die zur asynchronen Abarbeitung von Aufgaben verwendet werden.

- Erkennbar ist auch die Ausführung von Diensten durch `services.exe`. Dazu werden mehrere Prozesse von `svchost.exe` erzeugt, die dann wiederum verschiedene Threads starten. Besondere Dienste werden beispielsweise durch `spoolsv.exe` oder `smlogsvc.exe` bereitgestellt.

- Der Anmeldevorgang eines Benutzers kann in seiner Prozess/Thread-Struktur ebenfalls nachvollzogen werden. Nach dem Start von `winlogon.exe` werden drei Prozesse erzeugt: (1) `services.exe`, wie bereits beschrieben, (2) `lsass.exe`, das *Local Security Authority Subsystem* zur Prüfung von Benutzerinformationen und (3) `userinit.exe`, der dann schließlich mit `explorer.exe` die Windows Shell startet.

Mit diesen über den Quelltext hinausgehenden Informationen ermöglicht der WMK ein besseres Verständnis der Abläufe beim Starten eines Windows Systems.

### 5.3.2 Quantumlängen und Scheduling

Einer der wichtigsten dynamischen Aspekte eines Betriebssystems ist das Scheduling - also die Zuteilung von Rechenzeit zu ausgeführten Aktivitäten. In den meisten Betriebssystemimplementierungen wird dafür ein Prioritäten- und Zeitscheiben-basierter Ansatz verwendet. Der Scheduler weist die CPU dem lauffähigen Thread mit der höchsten Priorität zu. Ein ausgeführter Thread wird periodisch von Hardwareinterrupts einer Zeitquelle unterbrochen - nach einer definierten Zeit (= einem *Quantum*) wird dem Thread die CPU wieder entzogen, sofern noch andere lauffähige Threads mit dem gleichen Prioritätswert im System vorhanden sind. Die CPU wird auch dann einem anderen Thread zugeteilt, wenn der ausgeführte Thread die CPU freiwillig wieder abgibt beziehungsweise Ein- und Ausgabe- oder andere blockierende Operationen ausführt.

Mit dem dargestellten Verfahren lässt sich eine faire Zuteilung von Rechenzeit an parallel ausgeführte Aktivitäten implementieren. Üblicherweise werden zusätzliche Heuristiken verwendet um das System bezüglich Metriken wie beispielsweise Antwortzeiten bei interaktiver Benutzung oder Durchsatz bei Serveranwendungen zu optimieren.

Ein konfigurierbarer Parameter ist die Quantumlänge, also die Zeit die ein Thread die CPU maximal nutzen kann, bevor der Scheduler ihn unter Umständen mit einem Thread gleicher Priorität unterbricht. Ein kurzes Quantum kann die Antwortzeit verringern, da schnell reagiert werden kann - ein längeres Quantum soll den Durchsatz verbessern, da Anfragen ohne Unterbrechung verarbeitet werden.

Windows Server 2003 verwendet eine Quantumlänge von 180 ms. Diese relativ lange Zeit soll es Serveranwendungen ermöglichen Anfragen effizient zu bearbeiten. Die Voraussetzungen, die zur Auswahl dieses Werts führten, haben sich in den letzten Jahren jedoch stark verändert: Durch leistungsfähigere CPUs ist der Anteil von Rechenzeit zu Ein-/Ausgabezeit stark gesunken. Abgesehen von einigen Anwendungen im *High Performance Computing* beanspruchen Ein- und Ausgabeoperationen in der Regel die meiste Zeit.

Um diese These zu belegen sind Experimente mit dem WMK durchgeführt worden. Ziel war es, die *effektive Quantumlänge* - also die Zeit, die tatsächlich unterbrechungsfrei von Threads zur Berechnung verwendet wird - zu bestimmen.

Auf einem WMK Testsystem sind `CONTEXT_SWITCH`- und `QUANTUM_END`-Ereignisse aufgezeichnet worden. Diese Ereignisse ermöglichen die Rekonstruktion von Aktivitäten des Schedulers - und damit die Bestimmung der Rechenzeiten von Threads. Zwei Tests sind durchgeführt worden: Einmal wurde das instrumentierte System eine Stunde interaktiv mit verschiedenen Anwendungen verwendet. Dabei wurden beispielsweise Videos abgespielt, ein Text editiert oder im Web gesurft. In einem zweiten Test wurde der CPU-intensive Linpack-Benchmark [99] ausgeführt. Die Ergebnisse sind in Abbildung 5.6 und 5.7 dargestellt.

Die Ausführung verschiedener Anwendungen führte zu der in Abbildung 5.6 dargestellten Verteilung von effektiven Quantumlängen. Es ist zu beachten, dass beide Diagrammachsen logarithmisch unterteilt sind. In den meisten Fällen nutzten die ausgeführten Threads weniger als 1ms ihres verfügbaren Quantums von 180ms. Bei Desktopvarianten von Windows beträgt die maximale Quantumlänge immer noch 120ms - so dass auch dort nur ein Bruchteil der zur Verfügung stehenden Rechenzeit ohne Unterbrechung genutzt wird.

Aus der ermittelten Verteilung ergibt sich weiterhin ein Fairnessproblem. Der Windows Kern verwaltet für Threads die Information, wie lange sie im Usermode und im Kernelmode gerechnet haben. Dazu wird der periodische Uhr-Interrupt verwendet, der auch dazu dient das bereits verbrauchte Quantum eines Threads zu aktualisieren. Das Problem ergibt sich jetzt aus

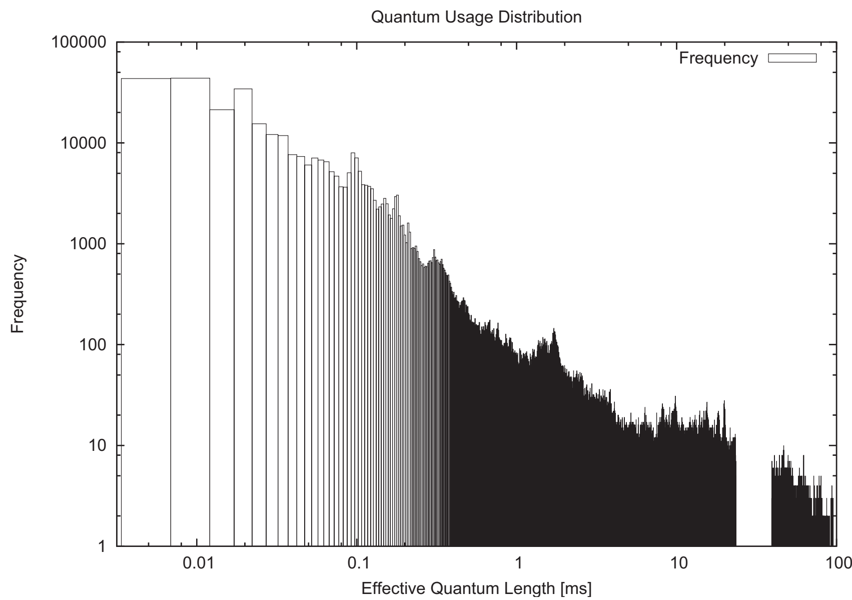



**Abbildung 5.6:** Effektive Quantumlänge: Anwendungen

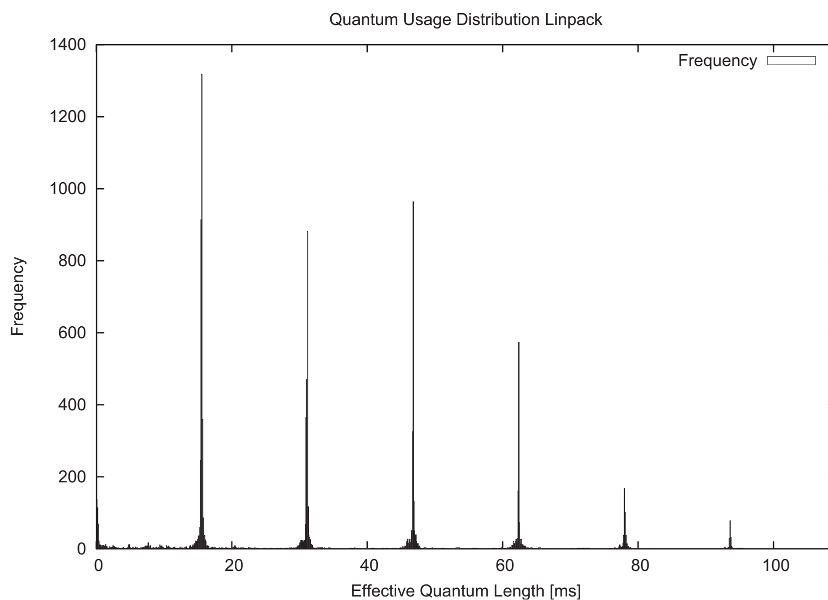



**Abbildung 5.7:** Effektive Quantumlänge: Linpack Benchmark

der Tatsache, dass der Windows Kern den Abstand zwischen zwei Uhr-Interrupts auf etwa 15ms konfiguriert: Zwischen zwei Uhr-Interrupts können mehrere Aktivitäten ausgeführt werden. Die verbrauchte Zeit wird jedoch dem Thread zugeordnet, der durch den Uhr-Interrupt tatsächlich unterbrochen wurde. Es ist daher möglich, dass Anwendungen die CPU stark belasten aber nach den Informationen des Kernel keine Rechenzeit verbrauchen. Dies führt zu verfälschten Heuristiken und damit zu ungünstigen „Optimierungen“.

Die gleichen Beobachtungen gelten auch für den Linuxkern [48, 101] und führten zur Implementierung alternativer Schedulingverfahren beziehungsweise eines *tickless* Betriebssystemkerns [14], der ohne den periodischen Uhr-Interrupt auskommt und die verbrauchte Rechenzeit misst.

| Anzahl | Aktivität |
|---|---|
| 252 | (csrss.exe) |
| 142 | ExpWorkerThread |
| 98 | ExpWorkerThreadBalanceManager |
| | *446 weitere Unterbrechungen* |
| 1 | MiModifiedPageWriter |

**Tabelle 5.1:** Linpack-unterbrechende Aktivitäten

Die Ausführung einer CPU-intensiven Last führt zu einer Verteilung der effektiven Quantumlängen wie sie in Abbildung 5.7 dargestellt ist. Die Ausführung des Linpack-Benchmarks erfordert keine Ein- und Ausgabeoperationen - prinzipiell ist ein Thread, der Linpack ausführt, immer lauffähig und könnte die CPU permanent verwenden. Dies ist jedoch nicht der Fall. Es existieren Häufungen von effektiven Quantumlängen in einem Abstand von etwa 15ms. Der Abstand ergibt sich erneut aus der Konfiguration des Uhr-Interrupts durch den Windows Kern: etwa alle 15ms wird der laufende Thread unterbrochen und eine Schedulingentscheidung findet statt.

Ist ein Thread mit höherer Priorität lauffähig, wird diesem die CPU zugewiesen. Der WMK ermöglicht die Identifikation dieser Threads, die den Benchmark tatsächlich unterbrechen. Eine Übersicht ist in Tabelle 5.1 dargestellt. Wenn möglich, wurde mit Hilfe der Debug-Informationen der Name der von den unterbrechenden Threads ausgeführten Funktionen identifiziert.

Der Linpack-ausführende Thread wurde durch Aktivitäten des Betriebssystems unterbrochen. Aufgrund unvollständiger Debug-Informationen konnte die Aktivität von `csrss.exe` nicht identifiziert werden. Es ist zu vermuten, dass diese Unterbrechungen mit der Aktualisierung der graphischen Benutzerschnittstelle zusammenhängen. Die Funktion `ExpWorkerThread` führt verschiedene Verwaltungsaufgaben des Kernels aus, beispielsweise das Laden von Treibern. Der *Balance Manager* wird einmal pro Sekunde aktiviert, um Threads mit niedriger Priorität, die lange nicht ausgeführt wurden, zu aktivieren. Einmal wurde der Benchmarkthread auch vom *Modified Page Writer* unterbrochen, der modifizierte Speicherseiten in die Auslagerungsdatei schreibt.

Alle diese Systemaktivitäten verhindern, dass der Benchmarkthread sein vollständiges Quantum nutzen kann. Weitere Untersuchungen haben gezeigt, dass sich die Anzahl der Unterbrechungen verringern lassen, wenn der Benchmarkthread auf einer höheren Prioritätsstufe (*Realtime* Priorität) ausführt wird.

Zusammenfassend lässt sich feststellen, dass die vom Windows Kern vorgegebene Quantumlänge fast keinen Einfluss auf die Ausführung von Anwendungen hat. Selbst Anwendungen, die die CPU vollständig auslasten könnten, nutzen ihr gesamtes Quantum nicht vollständig, da sie häufig von Systemaktivitäten unterbrochen werden.

### 5.3.3 Diskussion

Die beschriebenen Experimente zeigen, dass der WMK zur Analyse von Vorgängen im Windows Kern geeignet ist. Die Aufzeichnung der bereitgestellten Ereignisse können zu einem besseren Verständnis von Abläufen im Kern beitragen. Weiterhin konnte mit Hilfe des WMK ein Problem bei der Verwaltung von Timern im Windows Kern untersucht werden [88].

## 5.4 Softwareprüfstand

Die Integration der vorgestellten Techniken zur Aufzeichnung und Analyse von Ereignissen in den Betriebssystemkern ermöglicht es beliebige Anwendungen zu analysieren. Da in produktiv eingesetzten Systemen der Betriebssystemkern nicht (oder nur mit großem Aufwand durch Herunterfahren der Systeme) ausgetauscht werden kann, ist eine Verwendung der entwickelten Konzepte vor allem in Testsystemen möglich.

In einem solchen Testsystem kann jetzt der Betriebssystemkern ersetzt werden und eine Version gebootet werden, die die beschriebenen Konzepte zur Ereignisverarbeitung realisiert. Die Anwendung kann ohne Modifikation installiert und eingerichtet werden.

Ein entsprechendes Testsystem kann als Softwareprüfstand verwendet werden: Eine Anwendung wird installiert und mit einer (anwendungsspezifischen) Last getestet. Betriebssystemkernereignisse werden aufgezeichnet und analysiert. Zentrales Element des Prüfstands sind vordefinierte Konstellationen, die im Ereignisstrom entdeckt und gemeldet werden sollen. Die Definition der Muster wird dabei entweder vom Prüfstandentwickler oder vom Anwendungsentwickler vorgenommen. Die Anwendung kann eigene Ereignisse definieren und zur Laufzeit erzeugen. Sollen solche Ereignisse bei der Analyse berücksichtigt werden, muss die Musterdefinition durch den Anwendungsentwickler erfolgen.

Grundsätzlich können drei unterschiedliche Verfahren identifiziert werden, mit denen Musterdefinitionen zur Verwendung in einem solchen Softwareprüstand erzeugt werden können:

1. Zu entdeckende Muster können aus den Anforderungen der Anwendung abgeleitet werden. Dabei wird der Prüfstand dazu verwendet zu analysieren, ob die Anwendung gewünschte Abläufe tatsächlich durchführt. Bei Abweichungen wird der Prüfer benachrichtigt und kann anschließend entweder die Anwendung oder die Musterdefinition anpassen.

2. Darüber hinaus können allgemeine Muster definiert werden, die unabhängig von der Anwendung Konstellationen im Ereignisstrom entdecken, die Rückschlüsse auf spezifische Eigenschaften der Software erlauben. Dies können Probleme im Programmablauf (beispielsweise hohe Pagefault-Raten oder Zugriff auf nicht vorhandene Dateien), aber auch Leistungskenndaten (beispielsweise Durchsatz oder durchschnittliche Wartezeiten) sein.

3. Weiterhin kann der Prüfstand zur Analyse von fehlerhaften Anwendungsänderungen verwendet werden: Eine (fehlerfreie) ursprüngliche Variante der Software wird im Prüfstand ausgeführt und die Ereignisströme aufgezeichnet. Die Ereignisketten können durch die vorgestellten Werkzeuge analysiert werden und das gewünschte Verhalten aufgezeichnet werden. Wird nachfolgend der Ereignisstrom einer fehlerhaften Anwendungsvariante erzeugt, kann abweichendes Verhalten erkannt und analysiert werden.

Nachfolgend werden verschiedene generische Muster vorgestellt, die zur Analyse von Anwendungen verwendet werden können. Weiterhin werden Beispiele für anwendungsspezifische Muster vorgestellt.

Die dargestellten Regelbeschreibungen beziehen sich, wenn nicht anders angegeben, auf die im WMK verfügbaren Ereignistypen und deren Parameter. Eine Übersicht über die Datenelemente der Ereignisse ist im Anhang A zu finden.

In den Listings ist ausschließlich die zu verarbeitende Regel (`RULE`-Ausdruck) aufgeführt - das Einlesen der Ereignisdefinition mit `EVENTS "wmkevents.h"` ist nicht dargestellt. Ebenso werden keine Rückgabewerte dargestellt. Je nach zu erkennender Konstellation sind verschiedene Parameter für eine angemessene Reaktion erforderlich.

### 5.4.1 Erkennung von Wartezeiten

Dauert ein Vorgang in einer Anwendung „zu lange", kann dies zwei Ursachen haben: (1) eine Ressource (beispielsweise die CPU) ist durch die Komplexität der ausgeführten Funktion voll ausgelastet, oder (2) die Anwendung ist blockiert und wartet auf die Beantwortung einer Anfrage an eine externe Komponente. Der Anwendungsentwickler hat im Allgemeinen keinen Einfluss auf externe Komponenten. Daher ist das Zeitverhalten dieser Komponenten ein interessanter Analyseaspekt.

Betrachtet man aus Anwendungssicht den Betriebssystemkern als externe Komponente, so sind Systemaufrufe die entsprechende Schnittstelle. Mit der in Listing 5.1 dargestellten Regel lassen sich Systemaufrufe erkennen, die länger als eine Sekunde und damit ungewöhnlich lange dauern.

**Listing 5.1:** Beispiel: Erkennung langlaufender Systemaufrufe

```
1 ASYNCHRONOUS RULE longsyscalls
2   SKIPTILLNEXT PATTERN { [syscall:a, syscallexit:b] }
3   WHERE { [ProcessId],
4           [ThreadId],
5           b.TimeStamp - a.TimeStamp > 1s }
```

In der dargestellten Form wird mit der Regel nicht berücksichtigt, dass einige Systemaufrufe deutlich länger als eine Sekunde zur Ausführung benötigen: alle potentiell blockierenden Aufrufe, zum Beispiel `NtWaitForSingleObject`. Diese Systemaufrufe könnten explizit mit `a.SyscallNr != X` von der Erkennung ausgeschlossen werden. Ebenfalls ausgeschlossen werden müssen Systemaufrufe, die niemals zurückkehren - beispielsweise `NtTerminateThread`.

Existieren Ereignisse, die Wartezeiten - vor allem unnötige Wartezeiten, wie beispielsweise *timeouts* - explizit anzeigen, können die entsprechenden Ereignisdaten ausgewertet werden. In Listing 5.2 ist ein Beispiel für Synchronisationsfunktionen im Windows Kern dargestellt.

**Listing 5.2:** Beispiel: Erkennung von Timeouts bei der Synchronisation

```
1 ASYNCHRONOUS RULE synctimeout
2   PATTERN { [waitevent:w] }
3   WHERE { w.Flag == FALSE,
4           w.ReturnValue == 258 } // WAIT_TIMEOUT 0x00000102L
```

Das Ereignis `waitevent` wird (unter anderem) angezeigt, wenn ein blockierender `NtWaitFor*Objects`-Aufruf abgeschlossen wird. Das `Flag`-Datenfeld ist dann auf `FALSE` gesetzt und `ReturnValue` enthält das Ergebnis des Wartevorgangs. Timeouts treten auf, wenn innerhalb einer spezifizierten Zeitspanne keines der Objekte, auf deren Signalisierung gewartet wird, tatsächlich signalisiert wird. Mit der dargestellten Bedingung können Timeouts erkannt werden.

### 5.4.2 Erkennung von Fehlern

Fehler bei der Ausführung einer Anwendung können erkannt werden, wenn entweder ein Ereignistyp (oder ein entsprechendes Datenfeld) diese anzeigt oder durch Abweichung von erwartetem Verhalten.

Die in Listing 5.2 dargestellte Regel kann zum Beispiel auch zur Erkennung von weiteren Rückgabewerten bei Synchronisationsoperationen verwendet werden. Ebenfalls denkbar ist, dass in `syscallexit`-Ereignissen die `NTSTATUS`-Rückgabewerte erfasst werden. Diese können dann ebenfalls ausgewertet werden und beispielsweise Aufrufe mit dem Ergebnis `STATUS_UNSUCCESSFUL` erkennen.

Neben der Auswertung von Rückgabewerten können bei WMK-Ereignissen Fehler auch durch ungültige Objektreferenzen erkannt werden. Beispielsweise kann, wie in Listing 5.3 gezeigt, geprüft werden ob ein Objekt erfolgreich erzeugt werden konnte.

**Listing 5.3:** Beispiel: Erkennung von fehlerhafter Objekterzeugung

```
1 ASYNCHRONOUS RULE detectObjectCreationError
2   PATTERN { [createobject:o] }
3   WHERE { o.Object == 0 }
```

Zur Erkennung von Abweichungen von erwartetem Verhalten ist eine Spezifikation der Erwartung notwendig. In Listing 5.3 ist die (implizite) Erwartung, dass ein erfolgreich erzeugtes Objekt eine von 0 verschiedene Objektadresse besitzt.

Bei der Verwendung anwendungspezifischer Ereignisse können weitere Fehlerzustände erkannt werden, die direkt vom Anwendungsentwickler beschrieben werden. Die Möglichkeiten werden dabei durch Art und Struktur der angezeigten Ereignisse begrenzt.

### 5.4.3 Analyse von Synchronisationsvorgängen

Die Analyse von Synchronisationsvorgängen ist komplex. Die damit verbundenen Ereignisse können in sehr großer Zahl auftreten und es besteht das Problem, dass die Aufzeichnung dieser Ereignisse das Zeitverhalten des Systems beeinflusst.

Trotz dieser generellen Einschränkungen lassen sich mit den WMK-Ereignissen eine Reihe von Synchronisationsproblemen analysieren - Probleme, bei denen die Betrachtung der Synchronisationsoperationen ausreicht. Beispielsweise ist die Analyse von *Race Conditions* nicht möglich, wenn die zu schützenden Operationen nicht als Ereignis angezeigt werden.

Möglich ist die Erkennung einfacher Deadlock-Situationen. Eine entsprechende Regel ist in Listing 5.4 dargestellt.

**Listing 5.4:** Beispiel: Erkennung von Deadlocks

```
1 ASYNCHRONOUS RULE detectDeadlock
2   SKIPTILLNEXT PATTERN {
3     [ waitevent:a, ~ waitevent:ar0,
4       waitevent:b, ~(waitevent:ar1|waitevent:br0),
5       waitevent:c, ~(waitevent:ar2|waitevent:br1|waitevent:cr0),
6       waitevent:d ]
7   }
8   WHERE { a.ProcessId == c.ProcessId,
9           a.ThreadId == c.ThreadId,
```

```
10          b.ProcessId == d.ProcessId,
11          b.ThreadId == d.ThreadId,
12
13          a.Object == d.Object,
14          b.Object == c.Object,
15
16          a.Object == ar0.Object,
17          a.Flag != ar0.Flag,
18          a.Object == ar1.Object,
19          a.Flag != ar1.Flag,
20          a.Object == ar2.Object,
21          a.Flag != ar2.Flag,
22          b.Object == br0.Object,
23          b.Flag != br0.Flag,
24          b.Object == br1.Object,
25          b.Flag != br1.Flag,
26          c.Object == cr0.Object,
27          c.Flag != cr0.Flag,
28
29          [MultipleObjectsDataSize],
30          a.MultipleObjectsDataSize == 0 }
```

Die gezeigten Bedingungen stellen sicher, dass (1) beide Threads jeweils ein Synchronisationsobjekt belegt halten und (2) beide Threads auf das Synchronisationsobjekt des jeweils anderen warten. Dieser Zustand kann durch die Negationen im Muster geprüft werden.

Die Partition über `MultipleObjectsDataSize` ist notwendig um Ereignisse auszuschließen, die das gleichzeitige Warten auf mehrere Synchronisationsobjekte anzeigen. Da die Spezifikationssprache in der prototypisch implementierten Form keine Arrays als Ereignisparameter unterstützt, können solche Ereignisse nicht verarbeitet werden.

Damit ist keine sichere Erkennung von Deadlocks möglich. Für eine allgemeine Erkennung von Deadlocks - auch mit mehr als zwei beteiligten Threads - sind die entwickelten Sprachmittel nur umständlich anwendbar oder nicht ausreichend. Alternativ kann versucht werden über Scheduling-Ereignisse blockierte Aktivitäten zu erkennen und daraus Rückschlüsse auf Deadlocks zu ziehen.

Eine weitere Analyse von Synchronisationsbeziehungen kann bezüglich überlasteter Synchronisationsobjekte erfolgen. Befinden sich viele Threads in Konkurrenz um ein spezifisches Synchronisationsobjekt, sind sie im ungünstigsten Fall die meiste Zeit blockiert. Dieser Zustand wird als *lock contention* bezeichnet. Der Anwendungsentwickler kann versuchen die Synchronisationsstrategie zu ändern. Lock contention kann mit der in Listing 5.5 dargestellten Regel erkannt werden.

**Listing 5.5:** Beispiel: Erkennung von Lockcontention

```
1 ASYNCHRONOUS RULE detectLockContention
2   STRICTPARTITION PATTERN { [lock:a, lock[>=1]:b, lock:c] }
3   WHERE { [ObjAddress],
4           a.Flags & 16 == 1,
5           c.Flags & 16 == 0 }
6   RETURN { a.ObjAddress, b.len }
```

Mit `[ObjAddress]` wird eine Partition des Ereignisstroms definiert, die alle `lock`-Ereignisse eines bestimmten Objekts umfasst. Mit den Ereignissen `a` und `c` wird ein Zyklus aus Belegung

und Freigabe dieses Objekts erkannt - in den Bedingungen werden die entsprechenden Werte von `Flags` geprüft. Im Array `b` werden alle dazwischen liegenden Versuche das Synchronisationsobjekt zu belegen gezählt. Zurückgegeben wird das Synchronisationsobjekt, welches zu Blockierungen führte und die Anzahl erfolgloser Belegungsversuche.

### 5.4.4 Analyse von Seitenzugriffsfehlern

Ist der physikalische Speicher für ausgeführte Anwendungen nicht ausreichend, muss das Betriebssystem Speicherseiten auf Sekundärspeicher auslagern. Geschieht dies in verstärktem Maße, kann die Leistung von Anwendungen beeinträchtigt werden. Mit der Regel in 5.6 können hohe Seitenzugriffsfehlerraten erkannt werden.

**Listing 5.6:** Beispiel: Erkennung einer hohen Seitenzugriffsfehlerrate

```
1 ASYNCHRONOUS RULE detectHighPagefaultRate
2   SKIPTILLNEXT PATTERN { [pagefault[>1000]:a] }
3   WHERE { [ProcessId] }
4   WITHIN 1s
```

Der Windows Betriebssystemkern verwaltet Speicher auf Prozessebene, daher können mit `[ProcessId]` alle zu einem Prozess gehörenden Seitenzugriffsfehler gefiltert werden. Das Muster zählt jetzt die auftretenden Zugriffsfehler.

Problematisch an der Musterspezifikation ist, dass für jeden Seitenzugriffsfehler ein neuer Automatendurchlauf gestartet wird. Die Zahl der gleichzeitig aktiven Automaten kann daher sehr groß werden. Die Anzahl kann jedoch begrenzt werden, wenn die Regel sofort nach erfolgreicher Erkennung deaktiviert wird. Durch die `WITHIN`-Beschränkung ist ein Automat maximal eine Sekunde aktiv und wird danach verworfen. Wird die Konstellation erkannt, sind daher maximal 1000 Automaten aktiv - wird die beschriebene Konstellation nicht erkannt, ist die Anzahl kleiner.

### 5.4.5 Überwachung von Annahmen und Grenzwerten

In einigen Fällen lassen sich aus der Anwendungsspezifikation Annahmen über das Laufzeitverhalten bestimmter Threads ableiten. In Listing 5.7 ist eine Schablone für Muster zur Prüfung von Annahmen dargestellt.

**Listing 5.7:** Beispiel: Prüfen von Annahmen

```
1 ASYNCHRONOUS RULE checkAssumption
2   SKIPTILLNEXT PATTERN { [threadcreation:a, ~[ X ] ,threadtermination] }
3   WHERE { [ProcessId],
4           [ThreadId] }
5   RETURN { a.ThreadId }
```

Die Musterstruktur stellt sicher, dass für jeden Thread genau ein Automat erzeugt wird - `threadcreation`-Ereignisse treten nur einmal je Thread auf. An der Stelle `X` kann jetzt eine Sub-Musterbeschreibung eingefügt werden, die das Verhalten spezifiziert, das ein überwachter Thread *nicht* zeigen soll. Die Regel gibt die Threads zurück, die sich nicht entsprechend der Annahme verhalten.

Mit dem gleichen Schema lassen sich auch Ereignisse zählen: zwischen einem eindeutigen Startereignis und einem Endereignis (so dass nur ein entsprechender Automat aktiv wird) werden Sub-Muster erkannt und gezählt. Nach dem Endereignis kann dann beispielsweise die maximale Anzahl aktiver Prozesse zurückgegeben werden.

### 5.4.6 Diskussion

Die vorgestellten Regeln zeigen, dass mit Hilfe der entwickelten Spezifikationssprache Ereigniskonstellationen aus unterschiedlichen Anwendungsbereichen beschrieben werden können. Die Laufzeitumgebung kann diese erkennen und dem Analyst oder der Anwendung anzeigen.

Da die Anzahl und die Frequenz der auftretenden Ereignisse von der analysierten Anwendung abhängen und nicht vorhergesagt werden können, sind einige der vorgestellten Muster unter Umständen nicht analysierbar. Durch Einschränkung der analysierten Systembestandteile kann dieses Problem teilweise gelöst werden.

## 5.5 Adaption des Betriebssystems

Die Laufzeitumgebung bietet die Möglichkeit zur Ausführung von Skripten, wenn eine bestimmte Ereigniskonstellation erkannt wurde. Mit Hilfe dieser Funktion kann der bestehende Betriebssystemkern erweitert werden und mit neuen Algorithmen ergänzt werden. Dieser Ansatz soll nachfolgend am Beispiel der Prioritäten-Vererbung verdeutlicht werden.

Prioritäten-Vererbung soll das Problem der Prioritäten-Invertierung lösen. Prioritäten-Invertierung tritt auf, wenn ein Thread mit hoher Priorität auf ein Synchronisationsobjekt wartet, welches von einem Thread niedriger Priorität gehalten wird und dieser von einem Thread mittlerer Priorität unterbrochen wird. Der Thread mit mittlerer Priorität verhindert dann indirekt die Ausführung des Threads mit höherer Priorität.

Durch Prioritäten-Vererbung erhält der Thread mit niedriger Priorität die hohe Priorität des blockierten Threads und kann dann nicht mehr vom Thread mittlerer Priorität unterbrochen werden. Eine entsprechende Regel ist in Listing 5.8 dargestellt.

**Listing 5.8:** Beispiel: Implementierung von Prioritäten Vererbung

```
1  SYNCHRONOUS RULE resolvePriorityInversion
2    SKIPTILLNEXT PATTERN { [contextswitch:l, lock:a, ~lock:b, contextswitch:
       h, lock:c] }
3    WHERE { a.ObjAddress == b.ObjAddress,
4            a.ObjAddress == c.ObjAddress,
5
6            a.Flags & 16 == 1,
7            b.Flags & 16 == 0,
8            c.Flags & 16 == 1,
9
10           l.Priority < h.Priority }
11   DO {
12     CALL SetThreadPriority(a#TID, h.Priority)
13   }
```

Die Konstellationsspezifikation erkennt den Zustand der Prioritäten-Invertierung bezogen auf ein Synchronisationsobjekt. Das Skript führt die Prioritäten-Vererbung durch. Die Ereignisse

`l`, `a` und `b` gehören zum Thread mit niedriger Priorität. Wird dieser von einem Thread mit höherer Priorität unterbrochen (Ereignis `h`), der das gleiche Synchronisationsobjekt belegen möchte (Ereignis `c`), so wird die höhere Priorität vererbt.

Nicht dargestellt ist, dass der Thread mit niedriger Priorität diese wieder erhalten muss, sobald das Synchronisationsobjekt freigegeben wurde. Im Prototyp wurde dieses Verhalten näherungsweise nachgebildet, indem statt `SetThreadPriority` mit `BoostThreadPriority` die Priorität nur temporär angehoben wurde. Alternativ kann die Prioritäten-Vererbung durch ein Ereignis angezeigt werden und in einer zweiten Regel die Absenkung der Priorität implementiert werden.

Der im Rahmen dieser Arbeit entwickelte Prototyp stellt nur wenige Funktionen zur Reaktion auf erkannte Konstellationen bereit. Diese sind jedoch ausreichend um - wie gezeigt - Fälle von Prioritäten-Invertierung aufzulösen. Weiterhin wäre es denkbar alternative Schedulingansätze zu realisieren, indem die Thread-Prioritäten basierend auf Ereigniskonstellationen manipuliert werden.

## 5.6 Online Verarbeitung von Ereignissen - Komplexitätsanalyse

In diesem Abschnitt werden die vorgestellten Konzepte zur Ereignisverarbeitung analysiert und Eigenschaften der Verarbeitung abgeleitet. Ziel der Analyse ist theoretische Grenzen der entwickelten Konzepte zu identifizieren. Im Mittelpunkt stehen dabei notwendige Puffergrößen und Obergrenzen zur Verarbeitungszeit von Ereignisströmen.

Die Randbedingungen, unter denen eine Komplexitätsbetrachtung erfolgen kann, sind durch die folgenden Parameter gegeben:

- Ereignisrate $r$ - Anzahl der auftretenden Ereignisse pro Sekunde. Die Ereignisrate wird beispielsweise durch die Art der ausgeführten Programme beeinflusst.
- Muster-Zeitfenster $t$ - Zeitraum, in dem das erste und das letzte Element einer Musterbeschreibung erkannt werden muss. Das Zeitfenster beschreibt also eine Gültigkeitsdauer und wird durch eine `WITHIN`-Spezifikation definiert.
- Anzahl der zu speichernden Ereignisse $a$ - bezeichnet die Anzahl der Ereignisse die benötigt werden um das Muster zu erfüllen. Diese Anzahl ist beispielsweise unbestimmt, wenn unbeschränkte Felder verwendet werden.

Die Ereignisrate ist als statistische Verteilung zu verstehen, das heißt (beispielsweise für eine Normalverteilung) im *Durchschnitt* werden $r$ Ereignisse pro Sekunde im System auftreten. Die Angabe von $r$ ist unabhängig von konkreten Ereignistypen und fasst alle auftretenden Ereignisse im System zusammen.

Sind die Größen $t$ und $a$ nicht definiert, so können keine oberen Schranken für die Ereignisverarbeitung oder Puffergrößen angegeben werden: (1) eventuell müssen bereits erkannte Ereignisse beliebig lange gepuffert werden, so dass Puffer jeder Größe nicht ausreichen ($t$ undefiniert) und (2) eventuell müssen beliebig viele Ereignisse gepuffert werden ($a$ undefiniert).

Neben diesen Einschränkungen gilt, dass für jeden übersetzten Automaten ein *worst-case* Ereignisstrom konstruiert werden kann, der zu unbegrenzt vielen gleichzeitigen Laufzeitzuständen des Automaten führt: dies ist beispielsweise der Fall, wenn ausschließlich Ereignisse auftreten,

die einen Übergang vom Startzustand in einen Folgezustand auslösen. In einem solchen Fall können ebenfalls keine Obergrenzen für Puffer und Verarbeitungszeit angegeben werden.

In den nachfolgenden Abschnitten werden unter Berücksichtigung dieser Randbedingungen verschiedene Aspekte der Verarbeitung von Ereignisströmen untersucht.

### 5.6.1 Charakterisierung von Ereignisströmen

Auf abstrakter Ebene kann, wie bereits dargestellt, ein Ereignisstrom durch eine Ereignisrate $r$ beschrieben werden. Wird angenommen, dass alle Ereignistypen mit der gleichen Wahrscheinlichkeit auftreten, so kann die in den nachfolgenden Abschnitten durchgeführte Analyse ausschließlich von $r$ abhängend durchgeführt werden.

Ein solches Analysemodell kann jedoch nur eine grobe Näherung darstellen, da in der Praxis die Auftrittswahrscheinlichkeiten der Ereignistypen stark variieren kann. Weiterhin gilt, dass Abhängigkeiten zwischen Ereignistypen bestehen können. So wird beispielsweise ein Ereignis, welches einen Systemaufruf anzeigt, mit hoher Wahrscheinlichkeit innerhalb einer kurzen Zeitspanne von einem Ereignis, welches eine Rückkehr von einem Systemaufruf anzeigt, gefolgt.

In diesem Abschnitt wird gezeigt, wie ein realistischeres, stochastisches Modell für die Ereignisse eines Ereignisstroms aufgestellt werden kann. Ein solches Modell für ein konkretes System bildet dann die Grundlage für weiterführende Komplexitätsanalysen.

#### Modellierung

Mit Hilfe des Windows Monitoring Kernels (WMK) (siehe Kapitel 3) sind verschiedene Fallstudien und Tests durchgeführt worden. Unabhängig von der konkreten Fallstudie wurden auf diese Weise eine Menge von Logdateien aufgenommen, die für verschieden Lastbedingungen die in einem System auftretenden Ereignisse enthalten. Mit der Hilfe dieser Datenmenge soll ein probabilistisches Modell der auftretenden Ereignisse erstellt werden.

Ganz allgemein ist das Ziel, die Wahrscheinlichkeiten für das Auftreten bestimmter Ereignissequenzen $(e_1, .., e_n)$ im Ereignisstrom durch Auswertung der relativen Häufigkeiten entsprechender Sequenzen näherungsweise zu bestimmen. Genauer: für eine Menge von Ereignistypen $T$, die die Ereignistypen $t_1$ bis $t_x$ enthält, soll die Wahrscheinlichkeitsverteilung über $T$ für $e_n$ bestimmt werden unter der Voraussetzung, dass vorher die Ereignissequenz $(e_1, .., e_{n-1})$ aufgetreten ist (mit $e_i \in T$).

Für die Modellierung mit Bezug zu einer konkreten Regel sind die zu betrachtenden Sequenzen aus dem Automaten ableitbar: für jeden Zustand kann eine Folge von Ereignissen aufgestellt werden, die zum Erreichen notwendigerweise aufgetreten sein muss. Es ist möglich, das für einen Automatenzustand mehrere solche Sequenzen bestimmt werden. Die so erhaltenen Sequenzen sind die Vorbedingung für die nachfolgenden Transitionen. Betrachtet man die Ereignistypen, die eine entsprechende Transition auslösen, so sind die Übergangswahrscheinlichkeiten abschätzbar.

Einen Sonderfall stellen Felder von Ereignissen dar. Sind unbeschränkte Felder in der Musterbeschreibung vorhanden, lässt sich das nachfolgend beschriebene Verfahren nicht direkt verwenden, da unbeschränkte Sequenzen nicht gezählt werden können.

Für die Logdatei-Analyse ist ein Programm erstellt worden, welches eine Logdatei einliest und anschließend alle auftretenden Sequenzen von Ereignissen bis zu einer bestimmten Länge zählt

| vorheriges Ereignis | Ereignistyp | rel. Häufigkeit [%] |
|---|---|---|
| - | terminatethread | 0.05 |
| - | syscall | 50.07 |
| - | syscallexit | 49.88 |
| (terminatethread) | terminatethread | 28.79 |
| (terminatethread) | syscall | 18.08 |
| (terminatethread) | syscallexit | 53.13 |
| (syscall) | terminatethread | 0.05 |
| (syscall) | syscall | 5.76 |
| (syscall) | syscallexit | 94.18 |
| (syscallexit) | terminatethread | 0.03 |
| (syscallexit) | syscall | 94.57 |
| (syscallexit) | syscallexit | 5.40 |

**Tabelle 5.2:** Beispiel: Abhängigkeiten zwischen Ereignistypen

und so die relative Häufigkeit bestimmt. Die entsprechenden Daten werden zum Einen global, das heißt auf die gesamte Datei bezogen, und zum Anderen Ausführungskontext-spezifisch erfasst.

Die Ausführungskontext-spezifische Analyse wird in zwei Varianten durchgeführt: Einmal werden alle Teilmengen einzeln analysiert, die alle Ereignisse eines bestimmten Prozesses enthalten. Außerdem werden alle Teilmengen einzeln analysiert, die alle Ereignisse eines bestimmten Threads enthalten.

Neben der relativen Häufigkeit zur Abschätzung von Übergangswahrscheinlichkeiten ist die Rate, mit der bestimmte Ereignistypen auftreten, für die Analyse erforderlich. Nach dem beschriebenen Abarbeitungsmodell wird ein Automat zur Mustererkennung erzeugt, wenn der Startzustand verlassen wird. Dies ist der Fall, wenn das erste zum Muster passende Ereignis auftritt. Die Rate und Häufigkeit, mit der dieses *erste* Ereignis auftritt, bestimmt also die Zahl der erzeugten Automaten. Zusammen mit Abschätzungen der Zeit, die für einen Automatendurchlauf benötigt wird, kann dann errechnet werden, wie viele Automaten gleichzeitig aktiv sein werden. Daraus ergeben sich dann Werte zur Abschätzung benötigter Puffergrößen.

**Beispiele und Ergebnisse**

In einer Windows Monitoring Kernel Logdatei wurden die Ereignisse `syscall`, `syscallexit` und `threadtermination` analysiert. Insgesamt sind ca. 800000 Ereignisse in der Logdatei enthalten. Die relative Häufigkeit ist in Tabelle 5.2 zusammengefasst.

Die Tabelle zeigt, dass Abhängigkeiten zwischen den drei Ereignistypen bestehen: die relative Häufigkeit der Ereignisse ohne Betrachtung des vorhergehenden Ereignis (erster Tabellenabschnitt) unterscheidet sich deutlich von den Werten, die mit Betrachtung des vorhergehenden Ereignis bestimmt werden.

Beispielsweise treten (wie zu erwarten) `syscallexit`-Ereignisse mit fast doppelt so hoher Häufigkeit auf, wenn vorher ein `syscall`-Ereignis auftrat. Für den Zustand eines Automaten, der mit einem `syscall`-Ereignis instantiiert wurde, lassen sich mit den Werten aus der Tabelle wesentlich bessere Abschätzungen der Übergangswahrscheinlichkeiten angeben.

Bei im weiteren Text beschriebenen Modellierungen von Ereignisströmen wurde eine Analyse nach dem hier vorgestellten Muster durchgeführt: WMK-Logdateien, die mit für den konkreten Anwendungsfall representativer Last aufgenommen wurden, werden hinsichtlich der relevanten Ereignisfolgen untersucht und die relativen Häufigkeiten als Abschätzung von Aufritts-wahrscheinlichkeiten verwendet.

#### Zusammenfassung

Die dargestellte Vorgehensweise führt zu realistischeren Modellen von Ereignisströmen. Es sei jedoch nochmals darauf hingewiesen, dass extreme *worst-case* Ereignisströme (in denen beispielsweise ausschließlich ein einzelner Ereignistyp mit hoher Frequenz auftritt) durch dieses Modell nicht erfasst werden können. Die Grundannahme der nachfolgenden Analysen ist, dass auch in Fehlersituationen das System eine mit Hilfe des beschriebenen Modells abbildbare Verteilung von Ereignistypen erzeugt.

### 5.6.2 Modellierung der Ereignisstromverarbeitung

Für Abschätzungen der benötigten Puffergrößen und Bearbeitungszeiten ist ein Modell der Ereignisstromverarbeitung erforderlich, das quantitative Abschätzungen der Verarbeitung ermöglicht. Nachfolgend wird eine solche Modellierung basierend auf absorbierenden Markov Ketten (siehe zum Beispiel [55]) entwickelt.

#### Modellierung

Für einen gegebenen Automaten einer Mustererkennung kann, kombiniert mit der im letzten Abschnitt beschriebenen Ereignisstromcharakterisierung, die Transitionsmatrix einer absorbierenden Markov Kette bestimmt werden.

Jeder Zustand des Automaten entspricht einem potentiellen Zustand in der Markov Kette: der Invalid- und der Resultat-Zustand sind absorbierend, alle weiteren Zustände sind transient. Die Wahrscheinlichkeit des Anfangszustands ist mit $p = 1.0$ der Startzustand des Automaten.

Durch die Konstruktion des Automaten gilt, dass von jedem Zustand aus ein absorbierender Zustand erreicht werden kann. Das heißt, für jeden Zustand des Automaten existiert eine Folge von Transitionen zu einem absorbierenden Zustand. Damit ist auch diese Eigenschaft absorbierender Markov Ketten erfüllt.

Ausgehend vom Startzustand kann mit Hilfe einer gegebenen Ereignisstromcharakterisierung für jede Transition die Übergangswahrscheinlichkeit abgeschätzt werden, sofern die Transition durch ein neues Ereignis ausgelöst wird.

Relationen, also Bedingungen, die erfüllt sein müssen, damit eine Transition durchgeführt wird, basieren in der Regel auf bestimmten Ereignisfeldern und sind daher nicht allgemein abschätzbar. Betrachtet man die entsprechenden Übergangswahrscheinlichkeiten als Variablen, lassen sich dennoch Aussagen über die Bearbeitung eines entsprechenden Musters ableiten.

### Beispiel

Nachfolgend wird dargestellt, wie für eine konkrete Regel ein Modell der Bearbeitung aufgestellt werden kann. Als Beispiel dient die im Listing 5.9 dargestellte Regel `nosyscallexit`.

**Listing 5.9:** Beispiel: nosyscallexit-Regel

```
EVENTS "wmkevents.h"

ASYNCHRONOUS RULE nosyscallexit
  SKIPTILLNEXT PATTERN {
    [syscall:a, ~(syscallexit|threadtermination), syscall]
  }
  WHERE  { [ProcessId],
           [ThreadId],
           a.SyscallNr < 300 }
  RETURN { a.SyscallNr,
           a.TimeStamp }
```

Mit Hilfe der Regel sollen `syscall`-Ereignisse erkannt werden, zu denen kein dazugehöriges `syscallexit`-Ereignis im Ereignisstrom registriert wurde. Alle zu betrachtenden Musterereignisse müssen die gleiche `ProcessId` und `ThreadId` besitzen. Weiterhin werden nur Systemaufrufe betrachtet, deren Nummer kleiner als 300 ist.

Der Compiler generiert aus dieser Regel den in Abbildung 5.8 dargestellten Automaten.

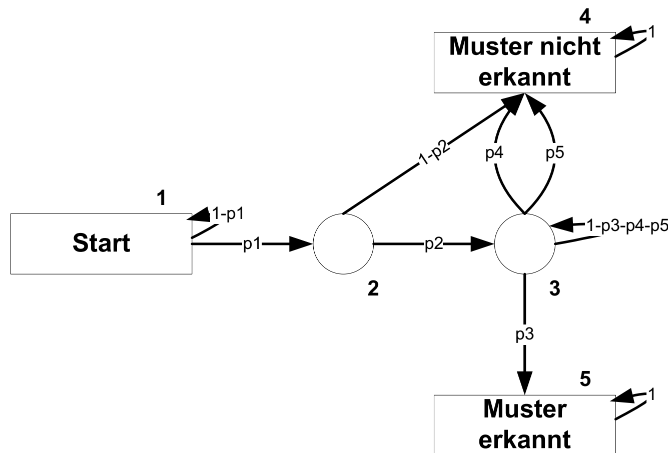



**Abbildung 5.8:** Automat zur nosyscallexit-Regel

Für die Übergangswahrscheinlichkeiten gelten die folgenden Annahmen: $p_1$ beschreibt die Wahrscheinlichkeit für das Verlassen des Startzustandes. Der Startzustand wird verlassen, wenn ein `syscall`-Ereignis auftritt; demnach lässt sich $p_1$ mit Hilfe des Ereignisstrommodells schätzen. $p_2$ beschreibt die Wahrscheinlichkeit, dass die Bedingung `a.SyscallNr < 300` erfüllt ist. Dies lässt sich nicht aus dem Modell ableiten. Die Werte für $p_3$ bis $p_5$ lassen sich wiederum aus dem Modell ableiten und beschreiben die Abschätzungen für die Wahrscheinlichkeiten von `syscallexit`-Ereignissen ($p_4$), `threadtermination`-Ereignissen ($p_5$) und weiteren `syscall`-Ereignissen ($p_3$) unter Betrachtung des entsprechenden Kontexts, also der vorher aufgetretenen Ereignisse. Die absorbierenden Zustände „Muster erkannt“ und „Muster nicht erkannt“ können nicht mehr verlassen werden.

Mit diesen Informationen lässt sich die Übergangsmatrix in Normalform für absorbierende Markov-Ketten aufstellen. An der Position in Zeile $x$ und Spalte $y$ wird die Wahrscheinlichkeit

für einen Übergang von Zustand $x$ nach Zustand $y$ eingetragen. Weiterhin gilt in der Darstellung der Übergangsmatrix in Normalform, dass die letzten Spalten (und Zeilen) der Matrix die absorbierenden Zustände beschreiben - die $n \times n$-Matrix (mit $n$ gleich der Anzahl der absorbierenden Zustände) in der „rechten unteren Ecke“ bildet die Einheitsmatrix.

$$P_{nosyscallexit} = \begin{pmatrix} 1-p_1 & p_1 & 0 & 0 & 0 \\ 0 & 0 & p_2 & 1-p_2 & 0 \\ 0 & 0 & 1-p_3-p_4-p_5 & p_4+p_5 & p_3 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Jede Zeile der Übergangsmatrix beschreibt die Wahrscheinlichkeit für den Übergang in einen anderen Zustand. Die erste Zeile zeigt die entsprechenden Werte für den Startzustand: mit einer Wahrscheinlichkeit von $p_1$ erfolgt ein Übergang in den Zustand 2. Dementsprechend ist die Zeilensumme jeder Zeile gleich $1.0$.

Aus der Normalform der Übergangsmatrix lassen sich die Teilkomponenten $Q$ und $R$ ableiten (siehe [55]), die für die weiteren Berechnungen benötigt werden.

Die Matrix $Q$ beschreibt die Wahrscheinlichkeiten, dass die Markov-Kette nach einem Übergang weiterhin in einem transienten (also nicht absorbierten) Zustand verbleibt. Durch die Zustandsanordnung der Normalform ist $Q$ also die $x \times x$-Matrix in der „linken oberen Ecke“ der Übergangsmatrix, wobei $x$ die Zahl der transienten Zustände angibt. Für das Beispiel ergibt sich daher:

$$Q_{nosyscallexit} = \begin{pmatrix} 1-p_1 & p_1 & 0 \\ 0 & 0 & p_2 \\ 0 & 0 & 1-p_3-p_4-p_5 \end{pmatrix}$$

Die Matrix $R$ beschreibt die Wahrscheinlichkeiten für Übergänge aus einem transienten in einen absorbierenden Zustand: die $y \times x$-Matrix in der „rechten oberen Ecke“ der Übergangsmatrix; $x$ gibt dabei wieder die Zahl der transienten und $y$ die Anzahl der absorbierenden Zustände an.

$$R_{nosyscallexit} = \begin{pmatrix} 0 & 0 \\ 1-p_2 & 0 \\ p_4+p_5 & p_3 \end{pmatrix}$$

Weiter der Terminologie aus [55] folgend lässt sich jetzt die Fundamental-Matrix $N$ der absorbierenden Markov-Kette durch Bestimmung der invertierten Matrix aus der Differenz der $x \times x$-Einheitsmatrix und $Q$ errechnen.

$$N = (I-Q)^{-1} = \begin{pmatrix} \frac{1}{p_1} & 1 & \frac{p_2}{p_3+p_4+p_5} \\ 0 & 1 & \frac{p_2}{p_3+p_4+p_5} \\ 0 & 0 & \frac{1}{p_3+p_4+p_5} \end{pmatrix}$$

Mit Hilfe der Fundamental-Matrix lassen sich jetzt die Wahrscheinlichkeiten der Absorbtion in den absorbierenden Zuständen bestimmen: Es gilt $B = NR$, die erste Zeile der Matrix $B$

| Parameter | | | | | Ergebnis | | |
|---|---|---|---|---|---|---|---|
| $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ | $p_{invalid}$ | $p_{return}$ | $s$ |
| 0.33 | 0.50 | 0.33 | 0.33 | 0.33 | 0.833 | 0.167 | 1.50 |
| 0.40 | 0.00 | 0.048 | 0.743 | 0.0001 | 1.000 | 0.000 | 1.00 |
| 0.40 | 0.25 | 0.048 | 0.743 | 0.0001 | 0.985 | 0.015 | 1.32 |
| 0.40 | 0.50 | 0.048 | 0.743 | 0.0001 | 0.970 | 0.030 | 1.63 |
| 0.40 | 0.75 | 0.048 | 0.743 | 0.0001 | 0.954 | 0.046 | 1.95 |
| 0.40 | 1.00 | 0.048 | 0.743 | 0.0001 | 0.939 | 0.061 | 2.26 |

**Tabelle 5.3:** Analyse: nosyscallexit

beschreibt dann die Wahrscheinlichkeit, dass die Markov-Kette im Invalid-Zustand absorbiert wird (erste Spalte) beziehungsweise den Return-Zustand (zweite Spalte) erreicht. Für das Beispiel ergeben sich folgende Werte:

$$p_{invalid} = 1 + p_2 + \frac{p_2(p_4 + p_5)}{p_3 + p_4 + p_5}$$
$$p_{return} = \frac{p_2 p_3}{p_3 + p_4 + p_5}$$

Es gilt $p_{invalid} + p_{return} = 1$, das heißt einer der beiden Zustände wird in jedem Fall erreicht. Die durchschnittliche Anzahl von Schritten (ausgehend vom Startzustand) lässt sich in der ersten Zeile des Vektors $t = Nc$ ablesen, dabei ist $c$ der Einheitsvektor. Für das Beispiel ergibt sich:

$$s = \frac{p_2}{p_3 + p_4 + p_5}$$

**Ergebnisse**

Die Berechnungsergebnisse für das Beispiel sind in Tabelle 5.3 dargestellt. Die Ergebnisse in der ersten Zeile beruhen auf einer Gleichverteilung der relevanten Ereignisse, das heißt alle drei Ereignistypen treten mit der gleichen Häufigkeit auf und die definierte Bedingung für die Systemaufrufnummern ist in der Hälfte aller Fälle wahr. Die restlichen Ergebnisse beruhen auf dem vorgestellten Ereignisstrommodell mit entsprechenden Übergangswahrscheinlichkeiten. Die Wahrscheinlichkeit, dass die Bedingung erfüllt ist, wird in 0.25 Schritten von 0.0 bis 1.0 variiert.

Treten `syscall`-Ereignisse mit einer Wahrscheinlichkeit von 0.4 im Ereignisstrom auf ($p_1$) und sind die Wahrscheinlichkeiten für die weiteren Muster-relevanten Ereignisse bekannt, hängt die Anzahl der notwendigen Schritte für eine Automatenbearbeitung nur von der Bedingung `SyscallNr < 300` ab. Diese ist durch die Wahrscheinlichkeit $p_2$ gegeben.

Es ist zu erkennen, dass das Muster nur mit einer geringen Wahrscheinlichkeit (< 6%) in einem Ereignisstrom vorkommen wird. Die Berechnung ergibt weiterhin, dass bei Annahme des bestimmten Ereignisstrommodells der Automat im Schnitt mit 2.26 Schritten (also Zustandsübergängen) durchlaufen wird.

Aus der Formel für die Anzahl der Schritte ergibt sich, dass der Wert maximal (beziehungsweise unendlich groß) wird, wenn $p_3 + p_4 + p_5 \to 0$ gilt. Das heißt, dass der Automat (siehe Abbildung 5.8) mit sehr geringer Wahrscheinlichkeit den Zustand 3 verlassen wird und mit jedem Ereignis im gleichen Zustand verbleibt.

### 5.6.3 Diskussion

Gegebene Regeln zur Mustererkennung können nach dem gezeigten Verfahren analysiert werden:

1. Übersetzen der Regel in einen deterministischen, endlichen Automaten nach dem im Abschnitt 4.3 vorgestellten Verfahren
2. Basierend auf dem erzeugten DEA: Aufstellung einer Übergangsmatrix für absorbierende Markov-Ketten
3. Parametrisierung der Übergangsmatrix mit Hilfe eines Ereignisstrommodells, das die zu erwartende Systemlast abbildet
4. Berechnung der durchschnittlichen Anzahl von Automatenschritten und der Absorbtionswahrscheinlichkeiten

Die Abschätzung der Anzahl der Schritte ermöglicht dann eine Abschätzung der benötigten Puffergrößen.

Wird ein sinnvolles Ereignisstrommodell vorausgesetzt, sind Relationen in der Regelbeschreibung entscheidend für die Anzahl der notwendigen Schritte. Ist eine Abschätzung der Wahrscheinlichkeiten möglich, kann die Übergangsmatrix entsprechend konfiguriert und die Anzahl der Schritte errechnet werden. Ist keine Abschätzung möglich, kann eine Extremwert-Analyse durchgeführt werden: aus der Formel für die Anzahl der Schritte wird abgeleitet, für welche Relationswahrscheinlichkeiten die Anzahl maximal wird und wie groß diese maximale Anzahl tatsächlich ist.

Bei entsprechender Interpretation der Extremwert-Analyse kann auch bei unbekannten oder schwer abschätzbaren Relationen die Anzahl der Schritte geschätzt werden. Dies gilt weiterhin nur unter der Einschränkung, dass sich das Gesamtsystem durch ein sinnvolles Modell abbilden lässt und Extremfälle nicht auftreten.

## 5.7 Weitere Untersuchungen

Im Rahmen dieser Arbeit sind weitere Fragestellungen untersucht worden, die verwandten Bereichen zuzuordnen sind. Dabei wurden Aspekte der entwickelten Konzepte aus einer anderen Perspektive untersucht. Entsprechende Implementierungen und Ergebnisse sind in diesem Abschnitt dargestellt.

### 5.7.1 Fein-granulare Zuteilung von CPU Zeit

In vielen Betriebssystemen wird eine Prioritäten-basierte Zuteilung von CPU Zeit verwendet: Mit auszuführenden Aktivitäten (Prozesse oder Threads) ist eine Priorität, beispielsweise ein

Zahlenwert, verknüpft. Der Scheduler des Betriebssystems wählt die Aktivität mit der höchsten Priorität aus und weist dieser die CPU zu. Nach dem Ablauf einer bestimmten maximalen Rechenzeit (= Quantum) oder wenn die Aktivität freiwillig die CPU aufgibt (beispielsweise durch eine Ein-/Ausgabeoperation), wählt der Scheduler die nächste Aktivität aus.

Diese indirekte Zuweisung von Rechenzeit basierend auf einem Prioritätswert erlaubt keine genaue Steuerung der Rechenzeit, die einzelne Aktivitäten erhalten: Die tatsächlich zugewiesene Zeit hängt beispielsweise von Anzahl und Verhalten von Aktivitäten mit höherer Priorität ab. Es gibt verschiedene Verfahren um beispielsweise Fairness [107] oder um eine bestimmte Rechenzeit [45, 110] zu garantieren. Im allgemeinen ist jedoch keine Möglichkeit zur direkten Zuteilung bestimmter Anteile der CPU zu nebenläufigen Aktivitäten vorhanden.

In diesem Abschnitt wird der *Scheduling Server* vorgestellt. Dieser erlaubt eine explizite und fein-granulare Zuteilung von CPU Rechenzeit zu parallelen Aktivitäten. Die Implementierung im Windows Research Kernel wurde in [92] vorgestellt und hier zusammengefasst dargestellt.

Die Grundidee des Scheduling Server wurde im Rahmen des SONiC-Projektes [81] entwickelt: Arbeitsplatz-PCs sollten zu einem Cluster kombiniert werden und für verteilte Berechnung verwendet werden können, ohne dass die Benutzer, die interaktiv an den PCs arbeiten, gestört werden. Der Scheduling Server soll daher sicherstellen, dass die Berechnungen im Cluster maximal einen bestimmten Anteil der CPU verwenden.

Viele Betriebssysteme verwenden, wie bereits beschrieben, einen Prioritäten-basierten Round-Robin Scheduling-Algorithmus. Diese Algorithmen können genutzt werden, um die Ausführung einer Menge von Aktivitäten unabhängig vom eigentlichen Kernel-Scheduler zu kontrollieren: Ein vom Scheduling Server gestarteter Dispatcher-Thread läuft auf der höchsten Priorität im System und hebt die Prioriäten kontrollierter Aktivitäten auf die zweit-höchste Prioritätsstufe. Anschließend gibt der Dispatcher-Thread die CPU frei, indem er eine bestimmte Zeit „schläft“. Jetzt wird die Aktivität auf der zweit-höchsten Prioritätsstufe ausgeführt. Durch die Steuerung, welche Aktivität gerade auf der zweit-höchsten Prioritätsstufe ist und durch die Steuerung der Wartezeit des Dispatcher-Threads kann die CPU-Nutzung der kontrollierten Aktivitäten genau gesteuert werden.

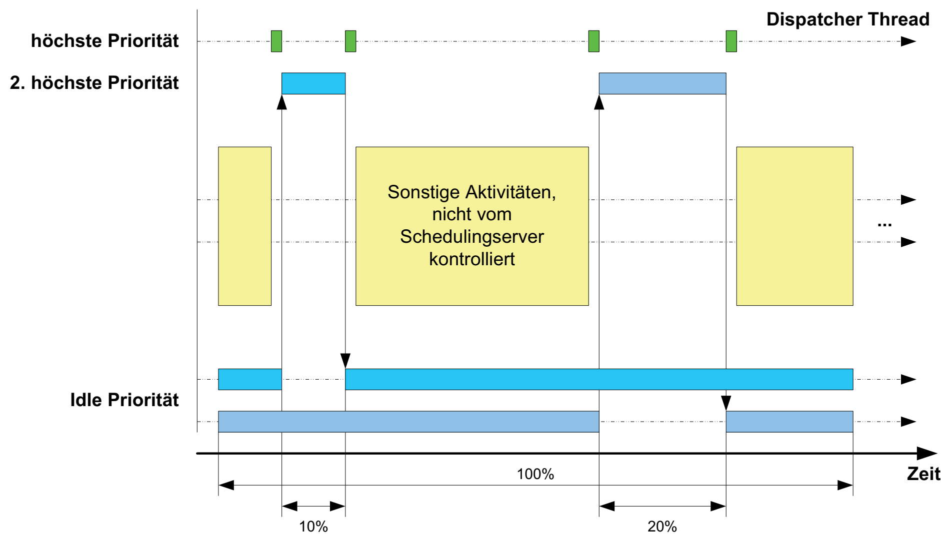



**Abbildung 5.9:** Schedulingserver Konzept

Die Funktionsweise des Scheduling Server ist in Abbildung 5.9 dargestellt: zwei Aktivitäten werden vom Scheduling Server verwaltet; eine soll nicht mehr als 10%, die andere nicht mehr

als 20% der CPU Zeit nutzen. Die verbleibende Zeit ist für sonstige Aktivitäten im System reserviert. Die 100%-Bezugszeit für die CPU Zuteilung wird als Scheduling Server Periode bezeichnet und kann durch Konfiguration vorgegeben oder dynamisch bestimmt werden.

Der Dispatcher-Thread hebt die Prioritäten der kontrollierten Tasks für eine bestimmte Zeit auf die zweit-höchste Prioritätsstufe - dadurch werden alle anderen Aktivitäten im System blockiert. Die zugewiesene CPU Zeit wird dadurch kontrolliert, dass der Dispatcher für eine bestimmte Zeit suspendiert wird. Nach Ablauf der Wartezeit wird die ausgeführte Aktivität unterbrochen, da der Dispatcher-Thread eine noch höhere Priorität besitzt. Jetzt kann der Dispatcher die nächste Aktivität auswählen und die Prioritäten entsprechend anpassen: die Priorität der zuletzt ausgeführten Aktivität wird wieder abgesenkt, die der neuen Aktivität erhöht. Wählt der Dispatcher keine neue Aktivität aus und suspendiert sich für eine bestimmte Zeit, können sonstige Aktivitäten im System die CPU verwenden. Durch die balancierte Ausführung von kontrollierten und sonstigen Aktivitäten (durch Suspendierung des Dispatcher-Threads) kann die CPU in den gewünschten Anteilen aufgeteilt werden.

Zu beachten ist, dass das beschriebene Konzept vollständig im User-Mode implementiert werden kann. Die einzige Anforderung ist, dass die Prioritäten entsprechend verändert werden können.

Eine Implementierung des Scheduling Server im Kernel-Mode kann ebenfalls nach dem beschriebenen Verfahren realisiert werden: ein System-Thread auf höchster Priorität übernimmt die Rolle des Dispatchers. Darüber hinaus kann die Aufgabe des Dispatcher-Threads in den regulären Betriebssystem-Scheduler integriert werden.

Im Rahmen der vorliegenden Arbeit wurde das Scheduling Server Konzept in den Windows Kern integriert. Dazu wurden zwei verschiedene Varianten realisiert und verglichen: (1) Eine Scheduling Server Implementierung für die Windows Plattform wird in [100] vorgestellt. Die dort beschriebenen Algorithmen wurden in den Kernel übertragen. (2) Die Aufgaben eines expliziten Dispatcher-Threads werden in den Clock-Interrupt Handler integriert. Zusätzlich wurden (wenige) Teile des Windows Schedulers entsprechend angepasst.

Die Güte der CPU Partitionierung kann durch die Differenz zwischen zugewiesener CPU Zeit und tatsächlich erhaltener CPU Zeit für rechenintensive Anwendungen angegeben werden. Eine detailliertere Darstellung zur Evaluation ist in [92] gegeben.

Für Usermode-Implementierungen [39, 68, 83, 100] wurde eine Güte von $> 5\%$ bestimmt. Die Usermode-Implementierungen verwenden jedoch unterschiedliche Strategien zur Verwaltung der auszuführenden Tasks, teilweise auch mit unterschiedlichen Zielstellungen der CPU Zuteilung.

Die Integration des Scheduling Server in den Betriebssystemscheduler (also in den Clockinterrupt-Handler) führt zu einem besseren Wert (etwa 1%) und zu einer stabileren Zuteilung der CPU an nebenläufig ausgeführte Aktivitäten.

Die Programmierschnittstelle des Schedulingservers erlaubt eine direkte und fein-granulare Zuteilung der CPU an nebenläufige Aktivitäten. In Kernel-Skripten, die von der Laufzeitumgebung zur Verarbeitung von Ereignisströmen ausgeführt werden, kann diese Funktionalität zur Anpassung von CPU Zuteilungen als Reaktion auf erkannte Ereigniskonstellationen verwendet werden.

### 5.7.2 Instrumentierung von Spinlocks

Der Windows Monitoring Kernel ermöglicht die Aufzeichnung von Ereignissen im Betriebssystemkern. Bestimmte Typen von Ereignissen treten jedoch in einer so hohen Zahl auf, dass eine Speicherung dieser Ereignisse eine starke Systembeeinflussung zur Folge hat.

Zur effizienten Synchronisation von Aktivitäten, die auf unterschiedlichen CPU Kernen im gleichen System ausgeführt werden, können Spinlocks verwendet werden. Spinlocks stellen eine Variante von „aktivem Warten“ dar: eine CPU liest fortwährend den Wert eines Speicherbereichs, bis dieser einen spezifischen Wert angenommen hat. Während dieses Vorgangs ist die CPU blockiert und führt ausschließlich den Wartevorgang aus. Wird angenommen, dass die Wartevorgänge in der Regel sehr kurz sind - also nur wenige Schleifendurchläufe bis zur gewünschten Wertänderung erforderlich sind - ist die Verwendung von Spinlocks verglichen mit aufwändigeren Synchronisationskonzepten besser.

Spinlocks werden vor allem zur Synchronisation von Betriebssystemkern-internen Vorgängen auf unterschiedlichen CPU Kernen verwendet.

Um die Effizienz der Synchronisation über Spinlocks zu untersuchen, ist die zentrale Fragestellung wie lange ein CPU Kern durch einen Spinlock blockiert ist. Wird jeder Spinlock-Durchlauf als Ereignis betrachtet, kann diese Fragestellung mit Hilfe von Instrumentierung analysiert werden. Solche Spinlock-Ereignisse würden jedoch in sehr hoher Zahl auftreten, so dass dieser Ansatz nicht durchführbar ist.

Tatsächlich war bei dem Versuch mit Hilfe des WMK die Zahl der Spinlock-Durchläufe aufzuzeichnen das Testsystem nicht mehr bootfähig. Schon beim Bootvorgang werden verschiedene Datenstrukturzugriffe durch Spinlocks synchronisiert. Die Art der Instrumentierung im WMK ist zu aufwändig für die Aufzeichnung dieser Ereignisse.

Zur Messung und Analyse von Spinlocks ist daher ein anderer Ansatz implementiert worden. Die Grundidee ist die Zeiten einzelner Spinlock-Durchläufe aggregiert zu erfassen und so auf explizit angezeigte, synchron zu verarbeitende Ereignisse zu verzichten.

In [52] wurde dieses Verfahren für den Linux-Kernel implementiert um *Lock Holder Preemption* (LHP) in mit Xen virtualisierten Umgebungen zu untersuchen. LHP tritt dann auf, wenn eine virtuelle CPU vom Hypervisor unterbrochen wird und diese gerade ein Spinlock erfolgreich aquiriert hat. Eine anschließend aktivierte andere virtuelle CPU, die auf den gleichen Spinlock wartet, verbraucht ihr CPU Quantum ohne Fortschritt.

Nachfolgend wird ein Ansatz zur Spinlock-Instrumentierung innerhalb des WRK vorgestellt. Anschließend werden Ergebnisse diskutiert. Das Auftreten von LHP im Windows Kern bei Verwendung von *VMWare Workstation* ist ebenfalls untersucht worden und konnte nicht nachgewiesen werden. Details sind in [86] zu finden.

#### Instrumentierung

Im WRK werden für 32 Bit x86-Systeme Spinlocks mit Hilfe zweier Assembler-Makros in `\base\ntos\inc\mac386.inc` implementiert. Die Makros `ACQUIRE_SPINLOCK` und `SPIN_ON_SPINLOCK` sind in Listing 5.10 dargestellt.

**Listing 5.10:** Makros: Spinlock Implementierung im WRK

```
ACQUIRE_SPINLOCK macro LockAddress, SpinLabel

    lock bts dword ptr [LockAddress], 0 ; test and set the spinlock
    jc       SpinLabel               ; spinlock owned, go SpinLabel

endm

SPIN_ON_SPINLOCK macro LockAddress, AcquireLabel

local a

a:  test   dword ptr [LockAddress], 1   ; Was spinlock cleared?
    jz     AcquireLabel                 ; Yes, go get it
    YIELD
    jmp    short a

endm
```

Beiden Makros wird mit `LockAddress` die Adresse des Spinlocks übergeben. Als zweiter Parameter wird `ACQUIRE_SPINLOCK` eine Sprungmarke übergeben, die aufgerufen wird, wenn das Spinlock nicht acquiriert werden konnte. In Zeile 3 wird versucht das Spinlock auf den Wert 1 zu setzen - gelingt das nicht, wird bei der Codeausführung zu `SpinLabel` gesprungen. Das Makro `SPIN_ON_SPINLOCK` implementiert den Wartevorgang. Als zweiter Parameter wird hier eine Sprungmarke angegeben, die aufgerufen wird, wenn das Spinlock wieder freigegeben wurde. Der Code zeigt, wie die Schleife `a: ... jmp short a` solange ausgeführt wird, bis mit Hilfe der `test`-Instruktion festgestellt wurde, dass das Spinlock wieder freigegeben wurde.

Beide Makros ermöglichen Spinlock-Implementierungen in verschiedenen Varianten (siehe `\base\ntos\ke\i386\spinlock.asm`). Ein Beispiel - die Implementierung der Funktion `KeAcquireSpinLockAtDpcLevel` - ist in Listing 5.11 dargestellt.

**Listing 5.11:** Beispiel: KeAcquireSpinLockAtDpcLevel

```
cPublicProc _KeAcquireSpinLockAtDpcLevel, 1
cPublicFpo 1,0

        mov     ecx,[esp+4]         ; SpinLock

aslc10: ACQUIRE_SPINLOCK    ecx,<short aslc20>

        stdRET    _KeAcquireSpinLockAtDpcLevel

aslc20: SPIN_ON_SPINLOCK    ecx,<short aslc10>

        stdRET    _KeAcquireSpinLockAtDpcLevel

stdENDP _KeAcquireSpinLockAtDpcLevel
```

Die Adresse des Spinlocks wird auf dem Stack übergeben und im `ecx`-Register gespeichert. Jetzt wird mit Hilfe der vorgestellten Makros versucht das Spinlock zu belegen: `ACQUIRE_SPINLOCK` wird mit dem Label `aslc20` aufgerufen, `SPIN_ON_SPINLOCK` mit dem

Label `aslc10`. Der Code zwischen den beiden Makros (Zeile 8) wird ausgeführt, wenn das Spinlock erfolgreich belegt werden konnte.

Um die Zeit zu messen, die für die Belegung des Spinlocks benötigt wird, muss ein Zeitstempel vor der ersten Ausführung von `ACQUIRE_SPINLOCK` und vor der ersten Instruktion, die nach erfolgreicher Belegung ausgeführt wird, gesichert werden.

Zur Speicherung der Messwerte wird pro CPU ein Array mit 32 `ULONG`-Werten verwendet. Die Datenstruktur *Processor Control Block* (PRCB) wurde um eine Referenz auf ein entsprechendes Array erweitert. Der erste Eintrag im Array dient dabei der temporären Speicherung des Startzeitpunktes einer Spinlock Belegung. Dies ist ausreichend, da ein spezifischer CPU Kern zu einem Zeitpunkt höchstens eine Spinlock Belegung durchführen kann. Der zweite Eintrag im Array wird als Überlaufzähler verwendet.

Die restlichen Werte repräsentieren Zähler für Wartevorgänge unterschiedlicher Länge: Nach der Messung der Start- und Endzeitpunkte wird die Differenz bestimmt. Die *Position* des höchsten Bits des Differenzwerts bestimmt den Eintrag im Array. An der achten Stelle wird also gezählt, wie viele Wartevorgänge zwischen 128 (10000000*b*) und 255 (11111111*b*) Taktzyklen zur Belegung eines Spinlocks benötigt haben. Dementsprechend wird an der neunten Stelle im Array gezählt, wie viele Wartevorgänge zwischen 256 und 511 Zyklen benötigten. Bei höheren Wartezeiten wird der von einem Zähler abgedeckte Bereich größer.

Die Wartezeit wird in Taktzyklen mit Hilfe der `rdtsc`-Instruktion bestimmt. Die ersten beiden Array-Einträge können wie beschrieben für andere Werte verwendet werden, da schon die eigentliche Messung der Zeitpunkte mehr als 4 Taktzyklen benötigt.

Durch die Verwendung der Makros zur Implementierung von Spinlocks ist das Einfügen von Instrumentierungscode für alle im Windows Kern verwendeten Spinlocks aufwändig. Für die nachfolgend beschriebenen Messungen wurde genau eine häufig verwendete Spinlock-Funktion (`KiAcquireSpinLock`) instrumentiert. Ein Auszug der instrumentierten Funktion ist in Listing 5.12 dargestellt.

**Listing 5.12:** Instrumentierung KiAcquireSpinLock

```
1  [...]
2
3      pushad                    ; save registers
4      mov     edi, fs:PcPrcb    ; get PRCB
5      mov     edi, [edi].PbSpinLockPerfData
6                                ; get spinlock performance counter
7                                ; array
8      cpuid                     ; serialize execution
9
10     ;; ### start measurement
11
12     rdtsc                     ; read time stamp counter
13     mov     [edi], eax        ; save low part of tsc in first field
14     popad                     ; restore registers
15
16 asl10:  ACQUIRE_SPINLOCK    ecx,<asl20>
17
18     pushad                    ; save registers
19     cpuid                     ; serialize execution
20     rdtsc                     ; read time stamp counter
21
22     ;; ### stop measurement
```

```
23
24      mov     edi, fs:PcPrcb      ; get PRCB
25      mov     edi, [edi].PbSpinLockPerfData
26                                  ; get spinlock performance counter
27                                  ; array
28      sub     eax, [edi]          ; subtract old tsc value from
29                                  ; new value
30      jo      overflow              ; counter overflow? -> skip
31      bsr     ebx, eax            ; determine most significant bit
32      shl     ebx, 2              ; calculate array entry -> multiply 4
33      add     edi, ebx            ; set edi to array entry
34      add     [edi], 1            ; increase array entry
35      jmp     done                ; done
36      overflow:
37      add     [edi+4], 1          ; increment overflow counter
38                                  ; (second field in array)
39      done:
40      popad                       ; restore registers
41
42  [...]
```

Gezeigt wird die Aufnahme des Startzeitpunkts vor dem ersten Aufrufen von ACQUIRE_SPINLOCK sowie die Berechnung der tatsächlichen Gesamtwartezeit nach erfolgreicher Belegung des Spinlocks.

Weiterhin wurde ein Systemaufruf implementiert, der die Spinlock Daten aus dem Kernel in den Usermode überträgt. Dort werden die Informationen der einzelnen CPU Kerne kombiniert und ausgegeben.

### Diskussion

Die Messungen wurden auf einer Intel Core2 Duo Workstation (3 GHz, 4 GByte RAM) durchgeführt. Als Testlast wurde der WRK kompiliert. Die Instrumentierung wurde erst direkt vor dem Test aktiviert. Die Messergebnisse für einen Durchlauf sind in Tabelle 5.4 dargestellt.

Die Belegung eines Spinlocks benötigt in den meisten Fällen zwischen 128 und 255 CPU Takte. Darin enthalten sind die Takte, die zur Zeitmessung benötigt werden. Vergleicht man diese

| Dauer [Zyklen] | Anzahl | % |
|---|---:|---:|
| < 64 | 0 | |
| 64 - 127 | 135 | 0.2 |
| 128 - 255 | 67749 | 98.1 |
| 256 - 511 | 1200 | 1.7 |
| 512 - 1023 | 3 | |
| 1024 - 2047 | 1 | |
| > 2048 | 0 | |

**Tabelle 5.4:** Dauer einer Spinlock Belegung

Anzahl von Takten mit dem Aufwand eines Funktionsaufrufes und blockierender Synchronisation, so wird ersichtlich, dass die Verwendung von Spinlocks in solchen Fällen tatsächlich sinnvoll ist.

Das dargestellte Verfahren stellt einen möglichen Weg zum Umgang mit sehr häufig auftretenden Ereignissen dar: Wenn nicht das einzelne Ereignis interessant ist, sondern ein abgeleiteter Wert, der mehrere Ereignisse zusammenfasst, kann die sofortige Zusammenfassung die Last reduzieren, die durch die Instrumentierung entsteht. Dieses Prinzip ist stellenweise in der Laufzeitumgebung zur sofortigen Verarbeitung von Ereignisströmen bereits berücksichtigt.

Die Analyse von Spinlocks zeigt jedoch auch, dass es Typen von Ereignissen geben kann, deren Analyse nicht möglich ist, wenn jedes Ereignis explizit angezeigt wird. Die synchrone Verarbeitung von Ereignissen ist offensichtlich nur dann möglich und sinnvoll, wenn der zeitliche Abstand zwischen zwei aufeinander folgenden Ereignissen (im gleichen Ausführungskontext) in der Regel größer ist als die Zeit, die zur Verarbeitung der Ereignisse erforderlich ist.

## 5.8 Zusammenfassung

In diesem Kapitel wurden verschiedene Fallstudien vorgestellt, die die Anwendbarkeit der im Rahmen dieser Arbeit entwickelten Konzepte demonstrieren. Im ersten Teil wurden verschiedene Analysen mit Hilfe des Windows Monitoring Kernels durchgeführt. Der WMK kann zu einem besseren Verständnis von Vorgängen in Anwendungen und im Betriebssystemkern beitragen. Im zweiten Teil wurde gezeigt, wie mit Hilfe der Laufzeitumgebung zur sofortigen Verarbeitung von Ereignisströmen ein Softwareprüfstand implementiert werden kann. Darüberhinaus ist ein Verfahren zur Abschätzung der Komplexität der Konstellationserkennung basierend auf absorbierenden Markov-Ketten vorgestellt worden. Abschließend sind weitere Konzepte zur Steuerung des Betriebssystemverhaltens und zur Aufzeichnung von häufig auftretenden Ereignissen dargestellt worden.

KAPITEL 6

# VERWANDTE ARBEITEN

*In diesem Kapitel werden die entwickelten Konzepte in den Kontext verwandter Arbeiten eingeordnet. Dabei werden die Konzepte mit alternativen Ansätzen verglichen und die Ergebnisse bewertet.*

## 6.1 Instrumentierungstechniken

Grundlegende Definitionen und Begriffe zur Instrumentierung von Systemen sind im Abschnitt 2.2 bereits dargestellt worden. In diesem Abschnitt werden konkrete Implementierungen überblicksartig dargestellt. Es werden ausschließlich Tracing-basierte Ansätze betrachtet. Sampling-basierte Ansätze, beispielsweise Profiler, sind nicht Gegenstand dieser Arbeit.

### 6.1.1 Klassifikation

Ansätze zur Instrumentierung können nach den folgenden Aspekten unterschieden werden:

**Systemebene:** Die verschiedenen Ansatzpunkte zur Instrumentierung eines Softwaresystems sind in Abbildung 2.1 dargestellt. Eine Instrumentierungsinfrastruktur kann an einer oder an mehreren der dargestellten Systemebenen ansetzen und Ereignisse aufzeichnen.

**Zeitpunkt:** Das wichtigste Unterscheidungsmerkmal ist, ob die Instrumentierung statisch - also vor der Programmausführung - oder dynamisch zur Laufzeit durchgeführt wird. Dieses Merkmal bestimmt, welche Informationen über das System für die Instrumentierung bekannt sein müssen.

**Definition:** Instrumentierungspunkte können basierend auf dem Quellcode oder basierend auf dem Binärcode eines Systems definiert werden. Dabei kann die Instrumentierung manuell eingefügt oder automatisiert generiert werden.

Im nächsten Abschnitt erfolgt eine Aufstellung von Systemen zur Instrumentierung. Diese folgt der Einteilung nach der Systemebene der Instrumentierung - von der Hardware zu den Anwendungssystemen.

### 6.1.2 Implementierungen

Instrumentierungstechniken sind in der Regel für alle Plattformen und Betriebssysteme verfügbar. Die dargestellten Systeme stellen eine Auswahl verschiedener Ansätze für unterschiedliche Systemebenen dar. Der Schwerpunkt der Diskussion liegt jedoch auf Tracing-basierten Instrumentierungstechniken im Betriebssystemkern.

### Hardwarenahe Instrumentierung

Die Verwendung von Hardwaretechniken, das heißt die Messung und Analyse von physikalischen Vorgängen im zu analysierenden System, wird in dieser Arbeit nicht betrachtet. Durch die Verwendung von Hardwareemulatoren können solche Analysen teilweise auch mit Software durchgeführt werden.

Ein Beispiel ist in [67] dargestellt: Der x86 Emulator QEMU wurde instrumentiert, so dass jede CPU-Instruktion aufgezeichnet werden kann. Dieses Vorgehen führt einerseits zu einem sehr hohen Detaillierungsgrad der aufgezeichneten Vorgänge und andererseits zu sehr großen Logdateien. Durch die Art der Instrumentierung und dem Volumen der aufgezeichneten Ereignisse ist nur *post-mortem* Analyse unterstützt.

Die Zuordnung der aufgezeichneten Ereignisse zu Vorgängen in höheren Schichten ist bei Instrumentierung auf Emulator-Ebene sehr komplex. Nur mit umfangreichem Wissen über die ausgeführten Systeme oberhalb des Emulators (Betriebssystem und Anwendungen) können entsprechende Vorgänge reproduziert werden, da die Bedeutung jedes Registerzugriffs für die Anwendung rekonstruiert werden muss.

Der hohe Detaillierungsgrad ermöglicht allerdings Analysen, die mit anderen Hilfsmitteln nicht möglich oder nur mit hohem Aufwand realisierbar sind, beispielsweise die Verfolgung von Datenfluss über gemeinsam genutzte Speicherbereiche. Die automatisierte Beobachtung und Anpassung des Systemverhaltens ist mit einem solchen Ansatz jedoch nicht realisierbar, da die Beeinflussung des zu beobachtenden Softwaresystems zu groß ist.

Auch ohne Emulator können bestimmte Vorgänge in der Hardware beobachtet werden. Intel CPUs stellen über das Auslesen (CPU-)Modell Spezifische Register [61] eine Möglichkeit bereit, Metriken zur Ausführung von Programmcode in der CPU abzufragen - beispielsweise L2 Cache Misses. Die Metriken können in der Regel in zwei Modi abgefragt werden: (1) über Sampling und (2) Ereignis-basiert. Die Erzeugung von Ereignissen erfolgt zusammengefasst, da sehr viele elementare Ereignisse auftreten können: Ein Register wird mit einem Wert initialisiert. Tritt das zu analysierende Ereignis auf - beispielsweise ein Cache Miss - wird der Registerwert dekrementiert. Erreicht das Register den Wert 0, wird ein Interrupt erzeugt und das entsprechende aggregierte Ereignis wird angezeigt.

Für die Aufzeichnung von Ereignisfolgen wird dieser Ansatz jedoch selten verwendet. Tools wie beispielsweise Oprofile für Linux [11] nutzen das beschriebene Vorgehen für das Profiling von Betriebssystemkern und Anwendungen.

Valgrind [77] bietet ebenfalls einen hardwarenahen Instrumentierungsansatz für Intel CPUs. Grundsätzlich wird auch hier eine CPU emuliert: ein spezieller Loader lädt den Binärcode eines Programms, analysiert diesen und fügt Instrumentierungscode ein. Auf diesen Weg ist eine sehr detaillierte Programmanalyse möglich, die aber mit einem hohen Overhead, beispielsweise von über 20% für die Analyse von Speicherzugriffen, sehr aufwändig ist. Aufgrund der Implementierung können Traps und Systemaufrufe mit Valgrind nicht nachverfolgt werden.

Hardwarenahe Instrumentierung erfolgt in der Regel statisch, automatisiert und basierend auf dem Binärcode eines Programmes. Durch die hohe Anzahl der zu instrumentierenden Stellen (beispielsweise einzelne Registerzugriffe) ist eine manuelle Instrumentierung oder eine Instrumentierung zur Laufzeit nicht sinnvoll. Die Instrumentierung basiert auf dem Binärcode, da erst der Compiler aus dem Quellcode die tatsächlichen hardwarenahen Eigenschaften des Programms erstellt.

### Hypervisorinstrumentierung

Als Virtualisierungsschicht zwischen der Hardware und dem Betriebssystem bietet sich der Hypervisor als Ebene zur Instrumentierung an um Vorgänge im ausgeführten Betriebssystemkern oder im Hypervisor selbst zu analysieren. Neben verschiedenen Sampling-basierten Ansätzen zur Lastüberwachung (beispielsweise [76]) sind auch Ereignis-basierte Ansätze entwickelt worden.

Der Hypervisor überwacht und steuert die Ausführung von virtuellen Maschinen. Analog zu Systemaufrufen in Betriebssystemen existieren *Hypercalls*, die zu einer Aktivierung des Hypervisors und danach beispielsweise zur Aktivierung einer anderen virtuellen Maschine führt. Im Allgemeinen lassen sich die Aktionen des Hypervisors als Ereignisstrom aufnehmen - beispielsweise in VMware ESX [12] oder mit Hilfe von `xentrace` im Xen Hypervisor.

Durch Aggregation der Ereignisse können der Zustand des Gesamtsystems und die Ressourcennutzung der virtualisierten Umgebung bestimmt werden. In [56] werden entsprechende Metriken aus dem Ereignisstrom abgeleitet und das Verhalten des Hypervisors angepasst. Ziel ist eine bessere Performanz-Isolation zwischen einzelnen virtuellen Maschinen zu erreichen.

Betrachtet man auch Usermode-basierte Virtualisierungsansätze zur Ausführung von isolierten Betriebssysteminstanzen als Hypervisor, so wird dieser in der Regel von einer Anwendung in einem regulären Betriebssystem bereit gestellt. Zur Instrumentierung können dann die im nächsten Abschnitt vorgestellten Ansätze zur Betriebssysteminstrumentierung verwendet werden.

Ein Beispiel für *User Mode Linux* (UML) wird in [69] beschrieben: mit Hilfe der `ptrace`-Systemaufrufschnittstelle wird der vom UML-Hypervisor erzeugte Ereignisstrom überwacht. Durch Analyse des Ereignisstroms sollen Einbrüche in eine virtuelle Maschine erkannt werden.

Je nach Art und Implementierung des Hypervisors sind sowohl statische als auch dynamische Instrumentierung möglich.

### Betriebssysteminstrumentierung

Durch die privilegierte Position des Betriebssystems als exklusiver Ressourcenverwalter sind Vorgänge im Kern für reguläre Anwendungen verborgen. Aus diesem Grund bieten Betriebssysteme unterschiedliche Schnittstellen an, um den aktuellen Zustand des Systems abfragen zu können. Beispiele sind im Windows Umfeld die *Performance Counter* [84] oder die Windows Management Instrumentation [102]. In POSIX-konformen Betriebssystemen existieren beispielsweise das `proc`-Dateisystem und der `ptrace`-Systemaufruf [30, 60].

Um Vorgänge im Betriebssystemkern detailliert erfassen und analysieren zu können und zur Aufzeichnung von Ereignisketten, sind über die bereitgestellten Schnittstellen hinausgehende Instrumentierungsansätze notwendig. Auch diese Ansätze können in die Klassen *Statisch* und *Dynamisch* unterteilt werden.

Das Einfügen von Instrumentierungspunkten erfordert genaue Kenntnisse von Codebasis und Vorgängen im Kern. Das Lesen von Daten aus Speicherbereichen des Kernels sollte beispielsweise unter Berücksichtigung der korrekten Synchronisationsbeziehungen erfolgen [89]. Die Analyse, an welcher Stelle im Quellcode welches Ereignis (= Zustandsänderung) mit welchen Informationen angezeigt werden kann, ist komplex und zeitaufwändig.

**Statische Ansätze** zur Betriebssystemkerninstrumentierung basieren im Allgemeinen auf dem Quelltext des Betriebssystems. Dazu ist es erforderlich, dass dieser Quelltext verfügbar ist.

Die Instrumentierungsinfrastruktur *Event Tracing for Windows* (ETW) [5] stellt eine Menge von statisch in den Betriebssystemkern integrierte Instrumentierungspunkte zur Verfügung. Mit Hilfe von WMI können diese Punkte aktiviert und deaktiviert werden. Verschiedene Werkzeuge erlauben die Aufzeichnung von Ereignissen.

Weitere statische Ansätze sind für den Windows Kern nicht verfügbar, da der Quelltext nicht offen zugänglich ist. Der im Rahmen dieser Arbeit entwickelte *Windows Monitoring Kernel* (WMK) (siehe Kapitel 3) stellt basierend auf dem Windows Research Kernel eine weitere Möglichkeit zur Instrumentierung des Windows Kerns zur Verfügung.

Zur Instrumentierung von Linux-Systemen sind, da der Kernel-Quellcode frei verfügbar ist, mehrere Systeme entwickelt worden.

Um das Problem der Positionierung von Instrumentierungspunkten im komplexen Kernel-Quellcode zu vereinfachen, ist das Konzept der *Kernel Marker* [9] eingeführt worden. Instrumentierungspunkte werden durch Experten in den Quellcode eingefügt, mit einem eindeutigen Namen versehen und können von anderen Entwicklern aktiviert und verwendet werden.

Darauf aufbauend implementiert das *Linux Trace Toolkit* (LTTng) [42, 43, 109] die Möglichkeit zur Aufzeichnung von angezeigten Ereignissen - im Kernel, (Xen-)Hypervisor und im Usermode.

Mit verschiedenen Verfahren soll in LTTng der Einfluss der Instrumentierung auf das Gesamtsystem reduziert werden. In Multiprozessor-Systemen wird dies beispielsweise durch pro CPU exklusiv genutzte Puffer und Lock-freier Synchronisation erreicht. Durch die Verwendung von *Read-Copy-Update*, einem Mechanismus zum (Cache-)effizientem Lesen von Werten die selten geändert werden, zur Aktivierung und Deaktivierung von Instrumentierungspunkten wird der Einfluss inaktiver Instrumentierungspunkte reduziert.

*KLogger* [48] ist ein statischer, Quellcode-basierter Instrumentierungsansatz ebenfalls für den Linux-Kernel. Instrumentierungspunkte werden durch ein Preprozessormakro definiert, welches den Typ und die Daten eines Ereignisses benötigt, um Code zur Generierung eines entsprechenden Eintrags in der Logdatei zu erzeugen. Darüberhinaus wird die Struktur der Ereignisse in einer C-ähnlichen Syntax definiert.

Ein detaillierter Vergleich von Implementierungsaspekten des WMK und KLogger ist in Abschnitt 3.5.3 zu finden.

**Dynamische Ansätze** basieren in der Regel auf dem Binärcode eines Kerns und sind dementsprechend komplexer zu implementieren. Bei der Instrumentierung sind zwei Ansätze unterscheidbar: (1) Instrumentierung durch Überschreiben von vorhandenen Instruktionen und (2) Instrumentierung durch Umschreiben (*binary re-writing*) von vorhandenen Instruktionen.

Beim Überschreiben wird Code zum Aufrufen der tatsächlichen Instrumentierung eingefügt. Dieser Code muss, nachdem die Instrumentierungsaktion durchgeführt wurde, den Kontrollfluss wieder an die richtige Stelle leiten und dabei die überschriebenen Instruktionen zusätzlich ausführen.

Beim Umschreiben wird im Allgemeinen der vorhandene Code in einen neuen Speicherbereich übertragen und dabei um Instruktionen zur Instrumentierung ergänzt. Alle Referenzen auf

den original Codebereich müssen anschließend auf den umgeschriebenen Bereich umgesetzt werden.

In DTrace [35] werden mögliche Instrumentierungspunkte durch sogenannte *Event Provider* identifiziert und als *Event Probes* bei der DTrace Laufzeitumgebung registriert. Werden Probes durch *Event Consumer* tatsächlich verwendet, sorgt der entsprechende Provider für die Aktivierung der Instrumentierung. Diese Architektur trennt die eigentliche Instrumentierung von der Verarbeitung der Ereignisse.

Probes zeigen Ereignisse an und stellen eine definierte Menge von Ereignisparametern zur Verfügung. Ein Benutzer kann in der Sprache D Code spezifizieren, der ausgeführt wird, wenn ein bestimmtes Ereignis auftritt. D Skripte werden von der DTrace Laufzeitumgebung interpretiert. Durch statische Analyse vor der Ausführung und Parameterprüfung zur Laufzeit kann eine sichere Ausführung der Skripte garantiert werden.

DTrace ist für die Instrumentierung von produktiv eingesetzten Systemen vorgesehen. Demnach sind die Sicherheit der Instrumentierung und ein *zero probe effect* wichtige Ziele - nicht aktive Instrumentierungspunkte sollen keinen Einfluss auf das Systemverhalten haben.

Für Windows Systeme ist ein zu DTrace vergleichbarer Ansatz im Rahmen einer Masterarbeit entwickelt worden [98]. Für eine sichere Ausführung der Skripte im Kernel-Mode wurden eine Reihe von notwendigen Änderungen im Windows Kern identifiziert.

Ebenfalls für Windows wird in [79, 80] ein Verfahren zu dynamischen Instrumentierung von Funktionseintritt und -austritt beschrieben. Zum Einfügen von Instrumentierungspunkten werden dabei Füllbytes (`nop`-Instruktionen) vor Funktionen überschrieben. Diese Füllbytes sind von Microsoft eigentlich für das Einfügen von Patches vorgesehen. Das beschriebene Verfahren ist sowohl für Usermode-Programme als auch für den Windows Kern einsetzbar.

Mit Pin [74] stellt Intel ein generisches Werkzeug für die dynamische Instrumentierung von Usermode-Anwendungen zur Verfügung. PinOS [33] erweitert das Einsatzgebiet auf Betriebssystemkerne: eine zusätzliche Ausführungsschicht zwischen dem (Xen-)Hypervisor und dem Gastbetriebssystem führt eine JIT Übersetzung des ausgeführten Binärcodes durch. Dabei wird Instrumentierungscode dynamisch eingefügt. Bestimmte Codeteile die sich nicht auf diese Art umschreiben lassen - beispielsweise System- beziehungsweise Hypervisoraufrufe - werden durch die Laufzeitumgebung emuliert.

Ein ähnlicher Ansatz wird auch in JIFL [78] verfolgt: Linux Kernelcode wird dynamisch umgeschrieben und dabei durch Instrumentierungscode ergänzt. JIFL versucht die Instrumentierung durch verschiedene Verfahren zu optimieren. Es wird beispielsweise versucht - falls möglich - den Instrumentierungscode direkt in den Kontrollfluß des umgeschriebenen Codeblocks einzufügen. Weiterhin wird versucht, den Einfluss des Instrumentierungscodes - beispielsweise die Anzahl der zu sichernden Registerwerte - zu reduzieren.

Der Linuxkern stellt mit KProbes eine Infrastruktur zur dynamischen Instrumentierung durch Instruktionsüberschreibung bereit. Ein Instrumentierungspunkt wird an einer bestimmten Stelle im Kernelcode eingefügt, indem Code registriert wird, der vor und nach der Ausführung des eigentlichen Instrumentierungscodes ausgeführt wird. Die Instruktion am Instrumentierungspunkt wird durch eine Trap-Anweisung ersetzt. Wird ein solcher Trap ausgelöst, wird die Laufzeitumgebung den entsprechenden Instrumentierungscode aufrufen.

Die KProbes-Infrastruktur wird in Systemtap [46] verwendet, um dynamische Instrumentierung ähnlich zu DTrace zu implementieren. Die Verwendung von Traps zur Anzeige von auftretenden Ereignissen führt im Vergleich zu den dargestellten JIT-Ansätzen zu einem erhöhten Overhead.

**Instrumentierung von Laufzeitumgebungen verwalteter Sprachen**

Laufzeitumgebungen verwalteter Sprachen interpretieren Programme, die in einem entsprechenden Bytecode-Format vorliegen, und führen diese aus - wichtige Vertreter sind die Laufzeitumgebungen für Java- und .NET-Programme. Die Indirektion über eine Laufzeitumgebung ermöglicht die Ausführung von Programmen auf unterschiedlichen Hardwareplattformen ohne Anpassungen, wenn eine entsprechende Laufzeitumgebung für diese Plattform verfügbar ist.

Analog zur Instrumentierung eines Betriebssystemkerns zur Analyse von nativ ausgeführten Anwendungen ermöglicht die Instrumentierung der Laufzeitumgebung die Analyse in ihr ausgeführter Anwendungen. Mit verschiedenen Mechanismen lassen sich so Ströme von Ereignissen während der Ausführung aufzeichnen - auch dann wenn der Quellcode der Anwendung nicht verfügbar ist.

Der interpretierte Bytecode einer Anwendung weist, verglichen beispielsweise mit nativem x86-Code, in der Regel eine geringere Komplexität auf. Statische und dynamische Ansätze zur Instrumentierung basierend auf dem Bytecode sind dementsprechend einfacher zu implementieren und werden im Abschnitt Anwendungsinstrumentierung betrachtet.

Vorgänge innerhalb der Laufzeitumgebung wie beispielsweise *Garbage Collection* sind außerhalb schwer nachvollziehbar. Für eine genaue Systemanalyse kann daher eine Instrumentierung der Laufzeitumgebung selbst notwendig sein - Mechanismen, die dazu zur Verfügung stehen, sind beispielsweise selbstdefinierte Klassenlader, Reflection und verschiedene Debug-Optionen der Laufzeitumgebung.

Am Beispiel von Java werden nachfolgend drei Ansätze zur Instrumentierung der Java Laufzeitumgebung kurz dargestellt.

Die *Java Management Extension* [8] stellt einen Ansatz zur Verwaltung und Überwachung von komponentenbasierten Java-Anwendungen dar. Dem Java-Komponentenmodell folgend können *Managed Beans* (MBeans) definiert werden, die den Zustand relevanter Komponenten kapseln. Für die Laufzeitumgebung von Java sind spezielle Plattform MBeans (MXBeans) verfügbar, die Aspekte wie Speicher Pools, Garbage Collection, Thread-Verwendung und JIT Kompilierung zugänglich machen und zur Analyse der ausgeführten Anwendungen verwendet werden können.

Die HotSpot VM Java Laufzeitumgebung für Solaris stellt Probes für DTrace [35] bereit. Mit Hilfe dieser Probes [3] können Vorgänge in der Laufzeitumgebung beobachtet und analysiert werden. Es können Ereignisse aus den Bereichen Initialisierung der Laufzeitumgebung, Thread-Verwendung, Klassenverwaltung (Laden/Entladen), Garbage Collection, JIT Kompilierung (welche Methoden) und Synchronisation verarbeitet werden. Weiterhin kann der Kontrollfluss durch ein Programm mit Hilfe von Ereignissen, die Methoden-Eintritt und -Austritt anzeigen, analysiert werden. Mit Hilfe von DTrace ist auch eine Verknüpfung von Ereignissen innerhalb der Laufzeitumgebung mit Ereignissen aus dem Betriebssystemkern möglich.

Verwaltete Sprachen verfügen im Allgemeinen über eine umfangreiche Standardbibliothek, die Funktionalität aus verschiedenen Bereichen (beispielsweise Netzwerkkommunikation oder Datenstrukturen) bereitstellt. Die Standardbibliothek kann als Bestandteil der Laufzeitumgebung betrachtet werden, insbesondere wenn ihr Quellcode nicht verfügbar ist. Die Instrumentierung dieser Bibliothek (auch ohne Quellcode) kann über dynamische oder statische Manipulation des Bytecodes erfolgen. Für Java ist in [28] ein Verfahren vorgestellt, mit dem sämtliche ausgeführten Klassen instrumentiert werden können. Die Standardklassen werden dabei sta-

tisch vor der Ausführung instrumentiert - anwendungsspezifische Klassen werden dynamisch instrumentiert.

Ein weiterer Ansatz wird in [49] beschrieben: zu einer bestehenden Vererbungsstruktur von Klassen (beispielsweise den Standardbibliotheksklassen) wird eine isomorphe Zwillingsstruktur erstellt. Diese wird erst bei Bedarf erstellt, lädt den Binärcode der Originalklassen und fügt Instrumentierungscode ein. Durch einen veränderten Klassenlader werden unveränderte Anwendungen mit den alternativen, instrumentierten Klassen ausgeführt.

Verwaltete Sprachen bieten eine Plattform mit in der Regel standardisierten und gut dokumentierten Eigenschaften. Basierend auf dem Bytecode der Programme und Debug-Optionen der Laufzeitumgebung können Ereignisströme aufgezeichnet und analysiert werden.

**Anwendungsinstrumentierung**

Anwendungen stellen die oberste Schicht der möglichen Instrumentierungsebenen dar. Vorgänge in Anwendungen lassen sich im Allgemeinen durch Instrumentierung von darunterliegenden Schichten analysieren.

Die direkte Instrumentierung einer Anwendung kann verwendet werden um spezifische Abläufe in der Implementierung selbst zu analysieren. Der Entwickler der Anwendung hat eine genaue Vorstellung über die Abläufe und kennt die Stellen im Programm, die für eine bestimmte Analyse relevant sind. Daher kann gezielter instrumentiert werden.

Es sind zwei Fälle zu unterscheiden: (1) Die Anwendung wurde direkt mit Mechanismen zur Aufzeichnung von Ereignissen ausgestattet - beispielsweise in Form verschiedener Logdateien, die (mit konfigurierbarem Detaillierungsgrad) Aktivitäten der Anwendung protokollieren. (2) Die Anwendung enthält keine solche Mechanismen. In diesem Fall ist weiter zu unterscheiden ob der Quelltext der Anwendung verfügbar ist oder nicht.

Im ersten Fall ist im Design der Anwendung die Analyse von internen Vorgängen bereits berücksichtigt worden. Zur Durchführung einer Analyse wird die Aufzeichnung entsprechend aktiviert. Für die Implementierung solcher Mechanismen steht der vollständige Sprachumfang der Programmiersprache zur Verfügung, in der die Anwendung entwickelt wurde.

Darüber hinaus sind verschiedene Bibliotheken entwickelt worden, die eine Integration von Tracing in Anwendungsprogrammen vereinfachen. Ein Beispiel ist das *Enterprise Instrumentation Framework* (EIF) von Microsoft [4]. Dieses Framework stellt Tools zur Verwendung der Windows üblichen Mechanismen zur Aufzeichnung von Ereignissen zur Verfügung - beispielsweise für die Windows Ereignislogdatei oder die *Windows Management Instrumentation* (WMI).

Im zweiten Fall, wenn die Anwendung entweder nicht mit Mitteln zur Analyse ausgestattet wurde oder die vorhandenen Konzepte nicht ausreichen, müssen Instrumentierungspunkte nachträglich eingefügt werden - basierend auf dem Quelltext (wenn verfügbar) oder basierend auf dem Binärcode.

Ein Ansatz, der zur Instrumentierung bestehender Anwendungen verwendet werden kann, ist die Aspekt-orientierte Programmierung (AOP). Dabei werden sogenannte Aspekte unabhängig entwickelt und mit dem eigentlichen Programm verwoben - mit Hilfe von Pointcuts werden die Stellen im Quelltext spezifiziert, die durch einen Aspekt ergänzt werden sollen. So ist es beispielsweise möglich Methodenaufrufe zu instrumentieren und nachzuverfolgen. Das AOP Konzept ist für eine Vielzahl von Plattformen und Programmiersprachen entwickelt worden. Beispiele sind AspectJ [64] für Java (statisch, Quellcode-basiert) oder Loom [93] für .NET Sprachen (statisch und dynamisch).

Unabhängig von AOP sind - vor allem für verwaltete Sprachen - verschiedene Ansätze [54, 34] zur Instrumentierung von Anwendungen in ihrer Binärcode-Repräsentation entwickelt worden.

Für die Instrumentierung von Anwendungen stehen eine Vielzahl von unterschiedlichen Ansätzen zur Verfügung. Anwendungen können dabei statisch und dynamisch instrumentiert werden. Da der Quellcode verfügbar ist, bietet sich aus der Sicht eines Entwicklers eine quellcode-basierte, statische Instrumentierung an. Entsprechender Programmcode kann auch im Produktiveinsatz weiter im Programm enthalten sein - und bei Bedarf zur Fehleranalyse aktiviert werden.

### 6.1.3 Diskussion - Einordnung des WMK

Der *Windows Monitoring Kernel* (WMK) (siehe auch Kapitel 3) ist im Rahmen dieser Arbeit als Werkzeug für die dynamische Analyse des Windows Kerns entwickelt worden - primär also als Forschungswerkzeug. Der Einsatz von Instrumentierungstechniken in Forschungsprojekten weist wesentlich einfachere Rahmenbedingungen auf. Insbesondere die Anforderung an die Verfügbarkeit des untersuchten Systems sind im Allgemeinen geringer - Systeme können durchaus häufiger neu gestartet werden.

Der WMK und die implementierte Laufzeitumgebung sollen die Konzepte und den allgemeinen Ansatz verdeutlichen und als Prototyp in verschiedenen Szenarien erprobt werden. Die entwickelten Konzepte - insbesondere zur online Verarbeitung von Ereignisströmen - sind jedoch auch für den Einsatz in Produktivsystemen geeignet.

Produktiv eingesetzte Systeme sollen *im Fehlerfall* instrumentiert werden können - *im Normalfall* soll das System jedoch mit maximaler Leistung ausgeführt werden. Die Instrumentierungspunkte sollen, wenn sie nicht aktiviert sind, keinerlei Einfluss auf das System ausüben - in [35] wird dies als *zero probe effect* bezeichnet. Der vorgestellte WMK bietet in seiner momentanen Form nur statische Instrumentierung - die Instrumentierungspunkte werden zur Kompilierzeit in das System eingefügt. Die Aktivierung erfolgt über eine Ereignismaske, die für jeden Ereignistyp ein Bit enthält. Dieses Bit wird geprüft, wenn ein Instrumentierungspunkt ausgeführt wird. Ist es gesetzt, wird das Ereignis aufgezeichnet, sonst nicht. Da die Prüfung dieses Bits auch bei deaktivierten Ereignissen erfolgt, bietet die Implementierungsstrategie des WMK keinen *zero probe effect*.

Sind Neustarts des zu untersuchenden Systems zulässig, bietet die statische Instrumentierung des WMK eine höhere Flexibilität: Ereignisse können in nahezu beliebigen Punkten im System, in beliebigen Ausführungskontexten angezeigt und aufgezeichnet werden. Ansätze zur dynamischen Instrumentierung unterliegen Einschränkungen bei der Art und der Position von Instrumentierungspunkten. In vielen Systemen entspricht die dynamische Instrumentierung nur einer Aktivierung von bereits im System integrierten Instrumentierungspunkten - eine Ergänzung um weitere Ereignistypen ist nicht möglich.

Der WMK stellt über Systemaufrufe eine Schnittstelle zur Aufzeichnung von Ereignissen bereit. Daher können auf allen beschriebenen Instrumentierungsschichten oberhalb des Betriebssystemkerns Ereignisse mit dem WMK aufgezeichnet werden.

Bezogen auf die Laufzeitumgebung zur sofortigen Verarbeitung von Ereignisströmen ist eine wichtige Anforderung an die Instrumentierungsinfrastruktur, dass auftretende Ereignisse *synchron* - also im Kontext der Ereignis-auslösenden Aktivität - aufgezeichnet werden. Diese Anforderung ist im WMK erfüllt.

Zusammenfassend ist weiterhin festzustellen, dass der WMK die erste zu KLogger [48] vergleichbare Implementierung für den Betriebssystemkern von Windows ist.

## 6.2 Verarbeitung von Ereignisströmen

Die online Verarbeitung von Ereignisströmen ist im Kontext von Betriebssystemkernen bisher selten untersucht worden. Im Bereich der Wirtschaftsinformatik, speziell in der Lieferkettenanalyse, spielen Ereignisströme bei der Anwendung von RFID-Technik eine große Rolle: Sensoren erfassen eine Vielzahl von Warenbewegungen und liefern entsprechende Meldungen als Ereignisse an Auswertungssysteme.

Die Probleme bei der Ereignisverarbeitung im Betriebssystem und in der Anwendung der RFID-Technik sind in manchen Punkten ähnlich: (1) Eine Vielzahl von Ereignissen kann potentiell zu einem sehr hohen Datenvolumen von mehreren tausend Ereignissen pro Sekunde führen. (2) Die Ereignisströme sind eventuell unvollständig (defekte Sensoren) oder inkonsistent (falsche Reihenfolge).

In anderen Aspekten unterscheiden sich beide Typen von Ereignisverarbeitung: (1) Die Typen der potentiell auftretenden RFID-Sensor Ereignisse sind bekannt. (2) Die fehlerhafte Verarbeitung von Ereignissen im Betriebssystem kann zu einem Ausfall des Gesamtsystems führen. (3) Die Typen der generierten Ereignisse können im Betriebssystemkern beeinflusst werden. (4) Die Ereignisverarbeitung im Betriebssystemkern kann synchron erfolgen.

Der Hauptteil der vorliegenden Arbeit beschreibt eine Laufzeitumgebung zur Verarbeitung von Ereignisströmen im Betriebssystemkern. In verschiedenen anderen Anwendungsbereichen sind ähnliche Ansätze entwickelt worden, die in diesem Abschnitt kurz dargestellt werden.

### 6.2.1 Forschungskontext

Das Konzept *Ereignis* wird in unterschiedlichen Gebieten verwendet. In [53] werden die Konzepte aus den Bereichen der (aktiven) Datenbanken und der Wissensrepräsentation gegenüber gestellt. Der grundlegende Unterschied ergibt sich aus der Betrachtung von Ereignissen als Zeitpunkte (Zeitpunkt-Semantik) oder als Zeiträume (Intervall-Semantik) (siehe auch Abschnitt 2.4.1), sowie der Art der Beschreibung von Beziehungen zwischen Ereignissen.

Eine formale Betrachtung des Sequenz-Operators in Ereignisströmen ist in [105] vorgenommen worden. Dabei werden notwendige und wünschenswerte Axiome definiert, die das zeitliche Modell einer Ereignisverarbeitung erfüllen sollte. Die identifizierten Probleme mit bestehenden Systemen ergeben sich aus der Berücksichtigung komplexer, zusammengesetzter Ereignisse, deren Zeitstempel als Intervall beschrieben werden muss. Weiterhin wird ein asynchrones Bearbeitungsmodell angenommen: die Verarbeitung der Ereignisse erfolgt logisch getrennt von ihrer Erzeugung. Das in der vorliegenden Arbeit angenommene Modell für Ereignisströme genügt den dargestellten notwendigen Axiomen.

Gemeinsamkeiten und Unterschiede zwischen der Verarbeitung von Ereignissen (beziehungsweise von Ereignisketten) und ganz allgemein der Verarbeitung von Datenströmen werden in [37] untersucht. Ein grundlegendes Paradigma ist dabei der *Event-Condition-Action* (ECA) Ansatz - die elementaren Ereignisse, geltende Bedingungen und auszuführende Aktionen werden getrennt betrachtet. Dieser Ansatz ist flexibel in unterschiedlichen Einsatzgebieten anwendbar. Die Verarbeitung von Datenströmen beruht nicht auf der Trennung von Ereignis und Aktion -

vielmehr steht die Analyse von in der Regel kontinuierlich erzeugten und sehr großen Datenmengen im Mittelpunkt.

Beide Gebiete stellen also unterschiedliche Betrachtungsweisen sehr ähnlicher Problemstellungen dar: ECA erfordert die kontinuierliche Überwachung auftretender Ereignisse in Kombination mit der Prüfung von Bedingungen. Die Verarbeitung von Datenströmen erfordert die effiziente Kombination der gesammelten Daten.

Nach [37] existieren Unterschiede zwischen beiden Ansätzen in den folgenden Bereichen:

**Verarbeitungsmodus:** Bei der Verarbeitung von Datenströmen wird in der Regel ein Bereich des Datenstromes betrachtet, welcher das aktuellste Datenelement und eine vorgegebene Anzahl von vorherigen Elementen enthält. Die Größe des Bereiches richtet sich dabei entweder nach einer festgelegten Anzahl von Datenelementen oder nach einem Zeitraum, beispielsweise alle Datenelemente der letzten zwei Minuten. Bei der Erkennung von Mustern in einem Ereignisstrom muss dagegen mit einer Menge von unter Umständen nur teilweise erkannten Mustern operiert werden, da nicht vorhersagbar ist, wie groß ein entsprechender Arbeitsbereich sein muss um alle spezifizierten Muster zu erkennen.

Die Beschränkung auf einen bestimmten Bereich von Datenelementen und die Verwendung entsprechender Operatoren ermöglicht die Einhaltung bestimmter Verarbeitungsgarantien, also QoS-Anforderungen, bei der Verarbeitung von Datenströmen. Die Verarbeitung von Ereignisströmen erfolgt in der Regel nach einem *best-effort* Ansatz - auftretende Ereignisse werden sofort verarbeitet - der keine entsprechenden Qualitätsaussagen garantieren kann.

**Operatoren:** Bei der Verarbeitung von Ereignisströmen werden in der Regel einzelne Elemente betrachtet, also die Lagebeziehungen (zum Beispiel Sequenz und Negation) zwischen Ereignissen. In der Verarbeitung von Datenströmen werden in den meisten Fällen Operationen auf den Parametern einzelner Datenelemente durchgeführt, beispielsweise ein Zähler verwaltet oder ein Durchschnittswert ermittelt. Dementsprechend werden unterschiedliche Sprachkonzepte für die Beschreibung der Verarbeitung von Strömen bereitgestellt.

In der Ereignisstromverarbeitung werden Muster beschrieben und Bedingungen definiert. Die Verarbeitung von Datenströmen erfolgt in der Regel in der Form von SQL-ähnlichen Anfragen an den Datenstrom.

**Berechnungsmodell:** Die Verwendung des ECA-Ansatzes gibt drei Klassen von Berechnungen zur Verarbeitung von Ereignisströmen vor: Berechnungen auf der Ebene einzelner Ereignisse, Berechnung auf der Ebene der Ereignisparameter zur Prüfung von Bedingungen und Berechnung auf der Ebene der auszuführenden Aktionen. Die Verarbeitung von Datenströmen gibt in der Regel zwei unterschiedliche Klassen von Berechnung vor: Berechnungen auf den Datenelementen des aktuellen Bereichs und Berechnungen zur Manipulation des aktuellen Bereiches. Die Schwerpunkte der Verarbeitung sind dementsprechend unterschiedlich - Berechnungen auf der Ebene einzelner Elemente werden bei der Verarbeitung von Datenströmen nicht durchgeführt, (explizite) Berechnungen zur Anpassung des aktuell betrachteten Bereiches von Elementen existieren dementsprechend nicht bei der Verarbeitung von Ereignisströmen.

Die in der vorliegenden Arbeit entwickelten Konzepte lassen sich ebenfalls in das ECA-Schema einfügen: der Windows Monitoring Kernel stellt die Ereignisse zur Verfügung - Bedingungen

und Aktionen werden in der Laufzeitumgebung geprüft und ausgelöst. Darüber hinaus sind Gemeinsamkeiten zu beiden vorgestellten Ansätzen zu finden, die sich aus dem Einsatz innerhalb des Betriebssystemkerns ergeben. Das Konzept des Ausführungskontexts von Ereignissen führt zu implizit vorhandenen Beziehungen zwischen auftretenden Ereignissen, die in der Regel bei der Verarbeitung von Ereignisströmen nicht vorhanden sind. Das verwendete Instrumentierungsframework kann darüber hinaus Konsistenzeigenschaften der aufgezeichneten Ereignisströme sicherstellen, die es ermöglichen Operationen auf Parametern der auftretenden Ereignisse durchzuführen.

### 6.2.2 Sprachen zur Beschreibung von Ereigniskonstellationen

Spezifikationssprachen zur Beschreibung von Mustern sind in verschiedenen Kontexten entwickelt worden. In [71] sind grundlegende Aspekte von *Event Pattern Languages* (EPL) - also von Sprachen zur Beschreibung von Mustern in einem Ereignisstrom - aufgeführt: Ausdrucksmächtigkeit, Einfachheit der Notation, Präzise Semantik und Skalierbare Mustererkennung.

In einer konkreten Implementierung müssen diese Punkte ausgeglichen werden: komplexe Operatoren verlangen unter Umständen komplexere Berechnungen während der Mustererkennung. Komplexere Berechnungen könnten eine gute Skalierbarkeit verhindern.

Die Sprachmittel, die zur Beschreibung von Ereigniskonstellationen verwendet werden können, hängen direkt von den Eigenschaften der verarbeiteten Ereignisse ab. Beispielsweise sind Operatoren, die explizit auf die Kausalität zwischen Ereignissen Bezug nehmen, nur dann möglich, wenn die entsprechenden Informationen auch zu jedem Ereignis verfügbar sind.

Prinzipiell lassen sich drei Bestandteile von Sprachen zur Beschreibung der Verarbeitung von Ereignisströmen unterscheiden:

**Ereignisspezifikation:** Mit der Ereignisspezifikation wird die Menge der potentiell zu verarbeitenden Ereignisse beschrieben. Zu jedem Ereignistyp werden die verfügbaren Informationen spezifiziert. Dabei können die Dimensionen Zeit, Daten und Kausalität unterschieden werden. Die verfügbaren Informationen bestimmen die Möglichkeiten, die bei der Muster- und Regelspezifikation verfügbar sind.

**Musterspezifikation:** Die Musterspezifikation erfolgt mit Hilfe von Operatoren, die die Struktur der zu erkennenden Konstellation beschreiben können. Mögliche Operatoren lassen sich in die Klassen *Relational*, *Kausal* und *Temporal* einordnen.

**Regel-/Reaktionsspezifikation:** Für die Spezifikation von Regeln können zusätzliche Sprachelemente definiert sein, die beispielsweise einen Zeitrahmen für die Mustererkennung vorgeben. Weitere Möglichkeiten sind die Spezifikation von Rückgabewerten oder von Aktionen, die ausgeführt werden sollen, wenn das beschriebene Muster erkannt wird. Ganz allgemein fallen jene Sprachkonstrukte unter den Punkt Regelspezifikation, die nicht direkt der Definition der Muster*struktur* dienen.

Jede Sprache kann unterschiedliche Sprachmittel bereitstellen, und in verschiedenen Systemen sind auch ganz unterschiedliche Operatoren vorgeschlagen worden. Welche Menge von Operatoren tatsächlich sinnvoll und notwendig ist, hängt vom Anwendungsgebiet und auch vom subjektiven Empfinden des Entwicklers ab.

Interessant ist die folgende Feststellung von Luckham ([71], Seite 167): *„It is unclear which of the logical and set operators are the most useful. Implementation of efficient pattern matching for some of these operators is challenging and demands smart algorithms.“*

Diese Feststellung wird durch zahlreiche Forschungsarbeiten und unterschiedliche Ansätze zur Beschreibung von Ereigniskonstellationen untermauert: manche bereitgestellte Operatoren sind problemlos durch andere nachzubilden - andere wiederum führen tatsächlich zu einer kompakteren Spezifikation von Mustern oder zu erweiterten Beschreibungsmöglichkeiten. Die Menge der Operatoren, die für einen bestimmten Anwendungsfall optimal ist, hängt eben von diesem Anwendungsfall ab.

Verwandte Arbeiten sind vor allem in den folgenden drei Bereichen zu finden:

**Aktive Datenbanken:** Datenbankoperationen werden als Ereignisse aufgefasst, zu komplexeren Konstellationen kombiniert und als Grundlage für Aktionen verwendet. Entsprechende Ansätze werden vor allem in Geschäftsanwendungen verwendet, beispielsweise bei der Verwaltung von Lieferketten.

Beispiele sind Cayuga [31] und Snoop [36].

**Verarbeitung von RFID-Ereignissen:** In einer Umgebung, die mit Sensoren ausgestattet ist, können Bewegungen von Objekten mit RFID-Chip verfolgt werden. Die Sensoren erzeugen dabei Ereignisse, wenn Objekte erkannt werden. Je nach konkreter Anwendung wird dann auf erkannte Ereigniskonstellationen reagiert, beispielsweise wenn sich bestimmte Objekte gleichzeitig in der Nähe eines bestimmten Sensors befinden.

Beispiele sind Cascadia [104] und SASE [57, 44]. Zu SASE sind in [20, 108] Implementierungsdetails veröffentlicht worden.

**Autonomic Computing:** Bei der selbstständigen Verwaltung von komplexen Systemen treten ebenfalls eine Vielzahl von Ereignissen auf, die eine Reaktion erfordern.

Ein Beispiel ist in Amit (*Active Middleware Technology*) zu finden: Der *Situation Manager* [16, 17, 18] erkennt Konstellationen, die eine Reaktion erfordern.

Neben den dargestellten Systemen für konkrete Anwendungsfälle sind allgemeine Ansätze entwickelt worden, die konzeptionell Ereigniserzeugung und Ereignisverarbeitung trennen und damit in verschiedenen Bereichen eingesetzt werden können.

Mit XChangeEQ [32] können als XML-Daten vorliegende Ereignisse verarbeitet werden. Weitere generische Ansätze sind CEDR (*Complex Event Detection and Response*) [26] und EventScript [40].

Mit GEM [75] und dem Rapide-Framework [73, 72] sind Ansätze vorgestellt worden, die zur Systemspezifikation (insbesondere von dynamischen Aspekten) mit Hilfe von Ereigniskonstellationen verwendet werden können.

In den verwandten Arbeiten werden verschiedene Operatoren zur Beschreibung von Ereigniskonstellationen entwickelt. In Cayuga [31] wird beispielsweise der `FOLD`-Operator in CEDR [26] der `CANCELWHEN`-Operator beschrieben. Mit `FOLD` können Iterationen über die Elemente eines Ereignisstroms beschrieben werden. Entsprechend aggregierte Werte können zusammen mit den Ereignisdaten für andere Bedingungen verwendet werden. `CANCEL-WHEN` ermöglicht den Abbruch einer Mustererkennung, wenn eine bestimmte alternative Ereigniskonstellation auftritt.

Im Allgemeinen verwenden alle Systeme entweder eine SQL-ähnliche Syntax oder reguläre Ausdrücke zur Beschreibung von Ereigniskonstellationen und Bedingungen. Komplexe Operatoren können die Beschreibung bestimmter Konstellationen vereinfachen und werden von der im Rahmen dieser Arbeit entwickelten Laufzeitumgebung nicht unterstützt.

Weiterhin existieren verwandte Arbeiten bei denen beschriebene Ereigniskonstellationen nicht *erkannt* sondern *erzwungen* werden sollen: In [103] werden beispielsweise Interaktionsprotokolle basierend auf Ereignisketten spezifiziert. Damit wird der umgekehrte Weg beschrieben: eine angegebene Aktion darf nicht ausgeführt werden, wenn vorher nicht eine bestimmte Abfolge von Ereignissen auftrat. Solche Ansätze werden in dieser Arbeit nicht weiter betrachtet.

### 6.2.3 Diskussion - Einordnung der entwickelten Laufzeitumgebung

Die entwickelte Laufzeitumgebung zur sofortigen Verarbeitung von Ereignisströmen im Betriebssystemkern besteht aus zwei Komponenten: (1) einer Sprachspezifikation mit dazugehörigem Compiler und (2) der eigentlichen Laufzeitumgebung, die übersetzte Regeln laden und verarbeiten kann.

Die Spezifikationssprache zur Beschreibung von Ereigniskonstellationen und Aktionen stellt elementare Sprachkonstrukte bereit. Diese Sprachkonstrukte waren für die durchgeführten Fallstudien ausreichend. Für komplexere Operatoren (beispielsweise `CANCELWHEN`) ist im Kontext der Ereignisverarbeitung im Betriebssystemkern kein Anwendungsfall identifiziert worden.

Auf eine zusätzliche Spezifikationssprache für Ereignistypen ist in der vorliegenden Arbeit verzichtet worden. Durch die enge Verbindung zum Windows Monitoring Kernel war das direkte Einlesen der C-Datenstrukturen für vorhandene Ereignistypen der nahe liegende Weg.

Sprachumfang und Laufzeitumgebung sind am ehesten mit SASE [57, 44] vergleichbar. Synchrone Ereignisverarbeitung im Betriebssystemkern erforderten jedoch andere Implementierungsansätze bei der Pufferverwaltung und Synchronisation.

Das Konzept der Mustersemantik (siehe Abschnitt 4.2.2) ermöglicht eine genaue Spezifikation, wie sich die Konstellationserkennung verhalten soll. Dies ist vor allem notwendig, wenn zur Spezifikationszeit nicht alle potentiell auftretenden Ereignistypen bekannt sind.

In verwandten Arbeiten sind unterschiedliche Fragestellungen im Umfeld der Ereignisverarbeitung untersucht worden. Die im Rahmen dieser Arbeit durchgeführte quantitative Analyse, basierend auf absorbierenden Markov-Ketten, bietet einen neuen Ansatz zur Analyse. Grundlage ist dabei ein sinnvolles stochastisches Modell des Ereignisstroms im System.

Die entwickelte Laufzeitumgebung bietet einen neuen Ansatz zur Verarbeitung von Ereignissen direkt im Betriebssystemkern. Durch die sofortige Verarbeitung im Kontext der ereignisauslösenden Aktivität ist eine synchrone Reaktion auf erkannte Ereigniskonstellationen möglich - ohne explizite Logdatei und mit begrenzbarem Speicherverbrauch bei der Verarbeitung.

## 6.3 Zusammenfassung

Es existieren verschiedene Ansätze unterschiedlicher Komplexität zur Instrumentierung von Softwaresystemen. Ein Grundproblem von Tracing-basierter Systembeobachtung ist die Unvorhersagbarkeit der Anzahl auftretender Ereignisse. Im Rahmen dieser Arbeit wurde ein System zur statischen, Quellcode-basierten Instrumentierung des Windows Kerns entwickelt.

Die Verarbeitung von Ereigsnisströmen ist von der konkreten Instrumentierung unabhängig. In verwandten Arbeiten sind unterschiedliche Sprachkonstrukte zur Beschreibung von Ereigniskonstellationen in solchen Strömen vorgeschlagen worden. Ein *minimal* notwendiger Sprachumfang ist bisher nicht identifiziert worden und abhängig von den konkreten Anwendungsfällen.

Kapitel 7

# Zusammenfassung und Ausblick

*In der vorliegenden Arbeit wird ein neuer Ansatz zur Verarbeitung von Ereignisströmen im Betriebssystemkern vorgestellt. Eine Spezifikationssprache erlaubt die Beschreibung von Ereigniskonstellationen über unterschiedliche Ausführungskontexte hinweg. Solche Beschreibungen werden in einen deterministischen, endlichen Automaten übersetzt und von einer Laufzeitumgebung im Betriebssystemkern zur Analyse des Ereignisstroms verwendet. Die Laufzeitumgebung ermöglicht die sofortige Reaktion auf erkannte Konstellationen - entweder durch Ausführung von Skripten im Kernelmode oder durch einen Aufruf anwendungsspezifischer Callbacks.*

**Zusammenfassung**

In Kapitel 1 und 2 werden die Problemstellung definiert und die Grundlagen der Arbeit dargestellt. Vor- und Nachteile von Tracing als Technik zur Systembeobachtung werden diskutiert. Die Aufnahme vollständiger Ketten von auftretenden Ereignissen in einem System kann für die Analyse von Vorgängen hilfreiche Informationen liefern, die mit Sampling basierten Ansätzen nicht erfasst werden können. Auf der anderen Seite führt Tracing zu einer unvorhersagbaren - da Last-abhängigen - Datenmenge.

In Kapitel 3 wird der *Windows Monitoring Kernel* (WMK) vorgestellt. Der WMK ist die erste Instrumentierungsinfrastruktur für die statische, Quellcode-basierte Instrumentierung des Windows Kerns. Beschrieben werden Architektur und verfügbare Ereignistypen. Durch Lock-freie Synchronisation und Pufferverwaltung ermöglicht der WMK die effiziente Aufzeichnung von Ereignissen im System, bei geringer Systembeeinflussung.

In Kapitel 4, dem Kern der Arbeit, wird eine Laufzeitumgebung zur Verarbeitung von Ereignisströmen im Betriebssystemkern vorgestellt. Beschrieben werden eine Spezifikationssprache für Ereigniskonstellationen, die Umwandlung solcher Spezifikationen in deterministische, endliche Automaten und die Laufzeitumgebung zur online Verarbeitung von Ereignisströmen. Weiterhin werden Möglichkeiten zur sofortigen Reaktion auf erkannte Konstellationen gezeigt. Das Konzept der entwickelten Laufzeitumgebung ist dabei unabhängig von der konkret verwendeten Instrumentierungsinfrastruktur.

Fallstudien aus unterschiedlichen Anwendungsgebieten werden in Kapitel 5 dargestellt. Die Verwendung der entwickelten Konzepte im Rahmen eines Softwareprüfstands wird gezeigt.

In Kapitel 6 werden die entwickelte Instrumentierungsinfrastruktur und die Laufzeitumgebung zur Verarbeitung von Ereignisströmen im Kern in den Kontext verwandter Forschungsarbeiten eingeordnet. Ereignisstromverarbeitung in anderen Gebieten (beispielsweise in aktiven

Datenbanken oder bei der RFID-basierten Lagerhaltung) wird diskutiert und die Unterschiede zur Verarbeitung von Ereignissen im Betriebssystemkern herausgestellt.

---

**Ausblick**

Die im Rahmen dieser Arbeit entwickelten Prototypen lassen sich im Hinblick auf ihre Implementierung in einigen Bereichen verbessern. Die Ereignisaufzeichnung kann optimiert werden, indem beispielsweise die Pufferverwaltung verbessert und der Einfluss von Caching-Effekten bei der Synchronisation untersucht werden. Die Laufzeitumgebung weist Verbesserungsmöglichkeiten beim Sprachumfang der Spezifikations- und Skriptsprache, sowie bei der Anbindung weiterer Instrumentierungsinfrastrukturen auf.

Ganz allgemein sind im Gebiet der Ereigsnisstromverarbeitung aufbauend auf die in der vorliegenden Arbeit entwickelten Konzepte weitere Fragestellungen zu untersuchen.

Nach Ansicht des Autors sind die Möglichkeiten zur Tracing-basierten Systembeobachtung und -analyse noch nicht in allen Bereichen voll ausgeschöpft. Die Verwendung von Tracing ist sinnvoll, da in dem aufgezeichneten Strom von Ereignissen in der Regel Hinweise auf die Lösung eines identifizierten Problems bereits vorhanden sind - eine Eigenschaft, die bei der Verwendung von Sampling-basierten Ansätzen nicht gegeben ist.

Die Kernprobleme von Tracing sind dann (1) die Systembeeinflußung, (2) die große Menge von anfallenden Daten und (3) die Identifikation von kritischen Ereigniskonstellationen.

Die Systembeeinflussung und die Datenmenge können reduziert werden indem, wie in der vorliegenden Arbeit gezeigt, nur die Informationen gespeichert werden, die für eine konkrete Analyse tatsächlich benötigt werden.

In weiterführenden Ansätzen kann beispielsweise untersucht werden, ob die Systembeeinflussung durch die Verwendung dedizierter CPUs für die Ereignisaufzeichnung reduziert werden kann. In einem solchen Szenario steht diese CPU nicht für Berechnungsaufgaben zur Verfügung - die aufgezeichneten Ereignisströme führen jedoch zu einem besseren Verständnis des aktuellen Systemzustandes und ermöglichen die gezielte Optimierung der Abläufe auf anderen CPUs. Durch die zunehmende Verbreitung von CPUs mit mehreren Kernen scheint ein solcher Ansatz denkbar. Ein weiterer Ansatz zur Reduzierung der Systembeeinflussung ist die kontinuierliche Aufzeichnung auftretender Ereignisse in einer Art Ringpuffer. Bei der Erkennung eines konkreten Problems enthält dieser Ringpuffer die „Entstehungsgeschichte“ der problematischen Situation. Durch Analyse des Ringpufferinhalts ist es dann unter Umständen möglich die Ursache des Problems zu identifizieren. Untersuchungsaspekte sind dann beispielsweise die notwendige Größe des Ringpuffers und das Programmiermodell für ein solches System.

Die Identifikation von kritischen Ereigniskonstellationen ist ebenfalls ein Feld für weiterführende Arbeiten. In der hier vorgestellten Form kann die Laufzeitumgebung zur Erkennung von *bekannten* und *beschreibbaren* Ereigniskonstellationen verwendet werden.

Die Ableitung von Ereigniskonstellationen, die eine Anpassung des Systems erfordern kann beispielsweise ebenfalls auf der kontinuierlichen Systembeobachtung basieren: bei der regulären Ausführung des Systems werden Ereignisfolgen aufgezeichnet, die das Systemverhalten im Normalfall beschreiben. Durch Verfahren der künstlichen Intelligenz können darauf aufbauend Muster abgeleitet werden, die abweichendes Verhalten beschreiben.

Für die effiziente Beschreibung von Konstellationen ist zu untersuchen, welche Sprachkonstrukte tatsächlich notwendig sind. In unterschiedlichen Systemen sind eine Vielzahl von Operatoren zur Beschreibung von Beziehungen zwischen Ereignissen vorgeschlagen worden. Darüberhinaus wurden unterschiedliche Ansätze für das Gesamtkonzept der Musterverarbeitung entwickelt: von einfachen Ereignis-Bedingung-Aktion Schemata hin zu fein-granularen Reaktionen auf schon erkannte Teilmuster. Eine konsistente theoretische Grundlage ist bisher nicht aufgestellt worden, so dass nicht klar ist, welcher Ansatz und welche Sprachkonstrukte tatsächlich notwendig sind. Die wichtigsten Punkte, die konsistent definiert werden sollten, sind:

**Zeitbezug:** Neben der Festlegung, ob die Zeitstempel von auftretenden Ereignissen Intervall- oder Zeitpunkt-Semantik besitzen sollten, ist die grundsätzliche Eigenschaft dieser Information zu definieren: Muss zwischen einem Erkennungszeitraum und einem Gültigkeitszeitraum eines Ereignisses unterschieden werden? Wann werden die Zeitstempel definiert? Können sie nachträglich geändert werden oder sind sie konstant?

**Kausalität:** Die Erfassung von Kausalitätsbeziehungen zwischen Ereignissen ist komplex und aufwändig - vor allem in verteilten Systemen. Die explizite Speicherung von Abhängigkeiten kann jedoch die Analyse von Ursachen von Ereignisfolgen erleichtern. Je nach gewähltem Ansatz ist entweder die Erzeugung von Ereignissen oder die Analyse von Ereignisfolgen aufwändiger.

**Operatoren:** Die Beschreibung von Ereigniskonstellationen kann mit beliebig komplexen Sprachkonstrukten erfolgen. In den meisten Fällen werden Konzepte ähnlich zu regulären Ausdrücken verwendet. Die genaue Definition von komplexen Operatoren - beispielsweise der Negation - ist in existierenden Systemen unterschiedlich.

Ob sich für diese Punkte tatsächlich eine einheitliche Festlegung finden lässt ist offen. Wahrscheinlich sind alle genannten Konzepte stark vom konkreten Anwendungsfall abhängig. Ansätze für ein konsistentes Verständnis sind beispielsweise in Form von häufig auftretenden Anwendungsmustern zu finden.

Die in dieser Arbeit vorgestellten Konzepte stellen die Grundlage für Anwendungen dar, die auf erkannte Ereigniskonstellationen reagieren möchten. Dies ist vor allem im Kontext des *Autonomic Computing* ein vielversprechender Ansatz. Die Definition von Konstellationen, die eine Reaktion erfordern, ist dabei ein Gebiet für weiterführende Arbeiten. Eine Menge generischer Muster zur Beschreibung problematischer Konstellationen kann beispielsweise als Grundlage für die Selbst-Heilung dienen. Auch notwendige Aktionen der Selbst-Konfiguration lassen sich basierend auf Tracing ableiten: Neu hinzugefügte Ressourcen oder ausfallende Komponenten lassen sich mit Hilfe eines kontinuierlich aufgenommenen Ereignisstroms identifizieren.

Wird der Systemzustand mit Hilfe von Sampling beobachtet, kann das Systemverhalten mit Hilfe eines Regelkreises beeinflusst werden. Die vorgestellte Laufzeitumgebung kann als Grundlage für einen ähnlichen Ansatz verwendet werden, der allerdings die Systembeobachtung mit Tracing-Ansätzen durchführt.

Die Reaktion auf erkannte Ereigniskonstellationen kann auch im Bereich der Aspekt- beziehungsweise Kontext-Orientierten Programmierung angewendet werden. In der Aspekt-Orientierten Programmierung werden Aspekte die viele Programmteile betreffen (zum Beispiel Logging oder Tracing) getrennt von der eigentlichen Programmlogik entwickelt und später miteinander verwoben. Die Spezifikation der Codestellen, die durch einen Aspekt ergänzt werden, wird im Allgemeinen Quellcode-basiert durchgeführt. In [25, 23] sind Ansätze vorgestellt, die Aspekte basierend auf dem Programmablauf (*trace*) ausführen. Die in dieser Arbeit

vorgestellten Konzepte können diese Idee weiterführen und auf Ereigniskonstellationen erweitern. In der Kontext-Orientierten Programmierung [41] wird Programmcode in Abhängigkeit eines Kontexts aktiviert und deaktiviert. Ereigniskonstellationen können zur Erkennung des aktuellen Kontexts und zur Erkennung von Kontextänderungen verwendet werden.

Speziell im Bereich der Betriebssysteme bieten die dargestellten Konzepte weitere Möglichkeiten. Denkbar ist eine Ereignisstrom-basierte Betriebssystemimplementierung. Dabei stellt der Kern eine Reihe von instrumentierten Abstraktionen bereit - beispielsweise Synchronisationsobjekte, die jede *Acquire*- und *Release*-Operation als Ereignis anzeigen. Komplexes Verhalten des Betriebssystemkerns kann dann als Reaktion auf erkannten Konstellationen implementiert und unter Umständen zur Laufzeit angepasst oder ausgetauscht werden. Untersucht werden muss, welche grundlegenden Abstraktionen und welche Konzepte zur Reaktion auf erkannte Konstellationen ein solcher Kern bereitstellen sollte.

# Literaturverzeichnis


[1] *Apache HTTP Server Project*. http://httpd.apache.org/.

[2] *Coco/R*. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/.

[3] *DTrace Probes in HotSpot VM*. http://java.sun.com/javase/6/docs/technotes/guides/vm/dtrace.html.

[4] *Enterprise Instrumentation Framework*. http://msdn.microsoft.com/en-us/library/ms979206.aspx.

[5] *Event Tracing for Windows*. http://www.microsoft.com/whdc/devtools/tools/EventTracing.mspx.

[6] *GNU Compiler Collection*. http://gcc.gnu.org/.

[7] *Httperf - tool for measuring web server performance*. http://www.hpl.hp.com/research/linux/httperf/.

[8] *Java Management Extensions (JMX)*. http://java.sun.com/javase/6/docs/technotes/guides/jmx/index.html.

[9] *Linux Kernel Markers*. http://lwn.net/Articles/245671/.

[10] *MPI Portable Instrumentation Library*. http://www.csm.ornl.gov/picl/mpicl.html.

[11] *Oprofile*. http://oprofile.sourceforge.net/.

[12] *VMware ESX Server 3: Ready Time Observations*. http://www.vmware.com/pdf/esx3_ready_time.pdf.

[13] *WRK@HPI Blog*. http://www.dcl.hpi.uni-potsdam.de/research/WRK/.

[14] *Linux: High-Res Timers and Tickless Kernel*. http://kerneltrap.org/node/6750, 2006.

[15] Abdelzaher, Tarek F.; Stankovic, John A.; Lu, Chenyang; Zhang, Ronghua; Lu, Ying: *Feedback Performance Control in Software Systems*. In *Proceedings of the IEEE Control Systems*, Bd. 23, S. 74–90. 2003.

[16] Adi, Asaf; Botzer, David; Etzion, Opher: *Semantic Event Model and its Implication on Situation Detection*. In *ECIS 2000 Proceedings*. 2000.

[17] ADI, ASAF; ETZION, OPHER: *The situation manager rule language*. In *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*. 2002.

[18] ADI, ASAF; ETZION, OPHER: *Amit - the situation manager*. The VLDB Journal, 13(2):177–203, 2004. ISSN 1066-8888.

[19] AGARWALA, SANDIP; SCHWAN, KARSTEN: *SysProf: Online Distributed Behavior Diagnosis through Fine-grain System Monitoring*. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, S. 8. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2540-7.

[20] AGRAWAL, JAGRATI; DIAO, YANLEI; GYLLSTROM, DANIEL; IMMERMAN, NEIL: *Efficient Pattern Matching over Event Streams*. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, S. 147–160. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-102-6.

[21] AHLUWALIA, JASWINDER; KRÜGER, INGOLF H.; PHILLIPS, WALTER; MEISINGER, MICHAEL: *Model-based run-time monitoring of end-to-end deadlines*. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, S. 100–109. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-091-4.

[22] AHO, ALFRED V.; LAM, MONICA S.; SETHI, RAVI; ULLMAN, JEFFREY D.: *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2. Aufl., 2007. ISBN 978-0321547989.

[23] ALLAN, CHRIS; AVGUSTINOV, PAVEL; CHRISTENSEN, ASKE SIMON; HENDREN, LAURIE; KUZINS, SASCHA; LHOTÁK, ONDŘEJ; DE MOOR, OEGE; SERENI, DAMIEN; SITTAMPALAM, GANESH; TIBBLE, JULIAN: *Adding trace matching with free variables to AspectJ*. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, S. 345–364. ACM, New York, NY, USA, 2005. ISBN 1-59593-031-0.

[24] ASTROM, KARL JOHAN; WITTENMARK, BJORN: *Computer-Controlled Systems: Theory and Design*. Prentice Hall, 3. Aufl., 1996. ISBN 978-0133148992.

[25] AVGUSTINOV, PAVEL; BODDEN, ERIC; HAJIYEV, ELNAR; HENDREN, LAURIE; LHOTÁK, ONDŘEJ; DE MOOR, OEGE; ONGKINGCO, NEIL; SERENI, DAMIEN; SITTAMPALAM, GANESH; TIBBLE, JULIAN; VERBAERE, MATHIEU: *Aspects for Trace Monitoring*. In Klaus Havelund; Manuel Nunez; Grigore Rosu; Burkhart Wolff, Hg., *Formal Approaches to Testing Systems and Runtime Verification*, Lecture Notes in Computer Science. Springer, 2006.

[26] BARGA, ROGER S.; GOLDSTEIN, JONATHAN; ALI, MOHAMED; HONG, MINGSHENG: *Consistent Streaming Through Time: A Vision for Event Stream Processing*. CoRR, 2006.

[27] BARHAM, PAUL; DONNELLY, AUSTIN; ISAACS, REBECCA; MORTIER, RICHARD: *Using magpie for request extraction and workload modelling*. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, S. 18–18. USENIX Association, Berkeley, CA, USA, 2004.

[28] BINDER, WALTER; HULAAS, JARLE; MORET, PHILIPPE: *Reengineering Standard Java Runtime Systems through Dynamic Bytecode Instrumentation*. In *SCAM '07: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, S. 91–100. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2880-5.

[29] BLANQUER, JOSEP M.; BATCHELLI, ANTONI; SCHAUSER, KLAUS; WOLSKI, RICH: *QoS for internet services: done right*. In *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, S. 8. ACM, New York, NY, USA, 2004.

[30] BOVET, DANIEL P.; CESATI, MARCO: *Understanding the Linux Kernel*. O'Reilly Media, 2005. ISBN 978-0596005658.

[31] BRENNA, LARS; DEMERS, ALAN; GEHRKE, JOHANNES; HONG, MINGSHENG; OSSHER, JOEL; PANDA, BISWANATH; RIEDEWALD, MIREK; THATTE, MOHIT; WHITE, WALKER: *Cayuga: a high-performance event processing engine*. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, S. 1100–1102. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-686-8.

[32] BRY, FRANÇOIS; ECKERT, MICHAEL: *Rule-Based Composite Event Queries: The Language XChange EQ and its Semantics*. In *In Proc. Int. Conf. on Web Reasoning and Rule Systems*. Springer, 2007.

[33] BUNGALE, PRASHANTH P.; LUK, CHI-KEUNG: *PinOS: a programmable framework for whole-system dynamic instrumentation*. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, S. 137–147. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-630-1.

[34] CABRAL, BRUNO; MARQUES, PAULO; SILVA, LUÍS: *RAIL: code instrumentation for .NET*. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, S. 1282–1287. ACM, New York, NY, USA, 2005. ISBN 1-58113-964-0.

[35] CANTRILL, BRYAN; SHAPIRO, MICHAEL W.; LEVENTHAL, ADAM H.: *Dynamic Instrumentation of Production Systems*. In *USENIX Annual Technical Conference, General Track*, S. 15–28. 2004.

[36] CHAKRAVARTHY, S.; MISHRA, D.: *Snoop: an expressive event specification language for active databases*. Data Knowl. Eng., 14(1):1–26, 1994. ISSN 0169-023X.

[37] CHAKRAVARTHY, SHARMA; ADAIKKALAVAN, RAMAN: *Events and streams: harnessing and unleashing their synergy!* In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, S. 1–12. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-090-6.

[38] CHANDA, ANUPAM; COX, ALAN L.; ZWAENEPOEL, WILLY: *Whodunit: transactional profiling for multi-tier applications*. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, S. 17–30. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-636-3.

[39] CHU, HAOHUA; NAHRSTEDT, KLARA: *A Soft Real Time Scheduling Server in UNIX Operating System*. In *Interactive Distributed Multimedia Systems and Telecommunication Services*, S. 153–162. 1997.

[40] COHEN, NORMAN H.; KALLEBERG, KARL TRYGVE: *EventScript: An Event-processing Language based on Regular Expressions with Actions*. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, S. 111–120. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-104-0.

[41] COSTANZA, PASCAL; HIRSCHFELD, ROBERT: *Language constructs for context-oriented programming: an overview of ContextL*. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, S. 1–10. ACM, New York, NY, USA, 2005.

[42] DESNOYERS, MATHIEU; DAGENAIS, MICHEL: *LTTng: Tracing across execution layers, from the Hypervisor to user-space*. In *Proceedings of the Linux Symposium*. 2008.

[43] DESNOYERS, MATTHIEU: *Low-Impact Operating System Tracing*. Doktorarbeit, Ecole Polytechnique de Montreal, 2009.

[44] DIAO, YANLEI; IMMERMAN, NEIL; GYLLSTROM, DANIEL: *Sase+: An agile Language for Kleene Closure over Event Streams*, 2007.

[45] EGGERT, LARS; TOUCH, JOSEPH D.: *Idletime scheduling with preemption intervals*. In *SOSP '05: Proceedings of the 20th ACM symposium on Operating systems principles*, S. 249–262. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-079-5.

[46] EIGLER, FRANK CH.; PRASAD, VARA; COHEN, WILL; NGUYEN, HIEN; HUNT, MARTIN; KENISTON, JIM; CHEN, BRAD: *Architecture of systemtap: a Linux trace/probe tool*. `http://sourceware.org/systemtap/archpaper.pdf`, 2005.

[47] ELNIKETY, SAMEH; NAHUM, ERICH; TRACEY, JOHN; ZWAENEPOEL, WILLY: *A method for transparent admission control and request scheduling in e-commerce web sites*. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, S. 276–286. ACM Press, New York, NY, USA, 2004. ISBN 1-58113-844-X.

[48] ETSION, YOAV; TSAFRIR, DAN; KIRKPATRICK, SCOTT; FEITELSON, DROR G.: *Fine Grained Kernel Logging with KLogger: Experience and Insights*. In *Proceedings of the EuroSys 2007*, S. 259–272. 2007.

[49] FACTOR, MICHAEL; SCHUSTER, ASSAF; SHAGIN, KONSTANTIN: *Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach*. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, S. 288–300. ACM, New York, NY, USA, 2004. ISBN 1-58113-831-9.

[50] FÖLLINGER, OTTO: *Regelungstechnik: Einführung in die Methoden und ihre Anwendung*. Hüthig Verlag, 10. Aufl., 2008. ISBN 978-3-7785-2970-6.

[51] FOX, ARMANDO; KICIMAN, EMRE; PATTERSON, DAVID: *Combining statistical monitoring and predictable recovery for self-management*. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, S. 49–53. ACM, New York, NY, USA, 2004. ISBN 1-58113-989-6.

[52] FRIEBEL, THOMAS; BIEMUELLER, SEBASTIAN: *How to Deal with Lock Holder Preemption*. `http://www.amd64.org/fileadmin/user_upload/pub/2008-Friebel-LHP-GI_OS.pdf`, 2008.

[53] GALTON, ANTONY; AUGUSTO, JUAN CARLOS: *Two Approaches to Event Definition*. In *DEXA '02: Proceedings of the 13th International Conference on Database and Expert Systems Applications*, S. 547–556. Springer-Verlag, London, UK, 2002. ISBN 3-540-44126-3.

[54] GOLDBERG, ALLEN; HAVELUND, KLAUS: *Instrumentation of Java Bytecode for Runtime Analysis*. In *In Proc. Formal Techniques for Java-like Programs, Volume 408 of Technical Reports from ETH Zürich*. 2003.

[55] GRINSTEAD, CHARLES M.; SNELL, LAURIE J.: *Grinstead and Snell's Introduction to Probability*. American Mathematical Society, version dated 4 july 2006 Aufl., 2006.

[56] GUPTA, DIWAKER; CHERKASOVA, LUDMILA; GARDNER, ROB; VAHDAT, AMIN: *Enforcing performance isolation across virtual machines in Xen*. In *Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, S. 342–362. Springer-Verlag New York, Inc., New York, NY, USA, 2006.

[57] GYLLSTROM, DANIEL; WU, EUGENE; CHAE, HEE-JIN; DIAO, YANLEI; STAHLBERG, PATRICK; ANDERSON, GORDON: *SASE: Complex Event Processing over Streams*. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research*. 2006.

[58] HRISCHUK, CURTIS E.; WOODSIDE, C. MURRAY; ROLIA, JEROME A.; IVERSEN, ROD: *Trace-Based Load Characterization for Generating Performance Software Models*. IEEE Trans. Softw. Eng., 25(1):122–135, 1999. ISSN 0098-5589.

[59] IBM: *An architectural blueprint for autonomic computing*. 2006.
URL `http://www-03.ibm.com/autonomic/pdfs/ACBlueprintWhitePaperV7.pdf`

[60] IEEE, THE OPEN GROUP: *The Open Group Base Specifications Issue 7 / IEEE Std 1003.1 (POSIX)*, 2008.
URL `http://www.opengroup.org/onlinepubs/9699919799/`

[61] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2007.

[62] ISRAR, TAUSEEF A.; LAU, DANNY H.; FRANKS, GREG; WOODSIDE, MURRAY: *Automatic generation of layered queuing software performance models from commonly available traces*. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, S. 147–158. ACM, New York, NY, USA, 2005. ISBN 1-59593-087-6.

[63] KEPHART, JEFFREY O.: *Research challenges of autonomic computing*. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, S. 15–22. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-963-2.

[64] KICZALES, GREGOR; HILSDALE, ERIK; HUGUNIN, JIM; KERSTEN, MIK; PALM, JEFFREY; GRISWOLD, WILLIAM G.: *An Overview of AspectJ*. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, S. 327–353. Springer-Verlag, London, UK, 2001. ISBN 3-540-42206-4.

[65] KOSKINEN, ERIC; JANNOTTI, JOHN: *BorderPatrol: isolating events for black-box tracing*. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, S. 191–203. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-013-5.

[66] LARSON, ULF E.; JONSSON, ERLAND; LINDSKOG, STEFAN: *A Structured Overview of Data Collection with a Focus on Intrusion Detection*. Techn. Ber., Göteborg University, 2008.

[67] LEFEBVRE, GEOFFREY; CULLY, BRENDAN; FEELEY, MICHAEL J.; HUTCHINSON, NORMAN C.; WARFIELD, ANDREW: *Tralfamadore: unifying source code and execution experience*. In *EuroSys '09: Proceedings of the fourth ACM european conference on Computer systems*, S. 199–204. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-482-9.

[68] LIN, CHIHHAN; CHU, HAOHUA; NAHRSTEDT, KLARA: *A Soft Real-time Scheduling Server on the Windows NT*. In *WINSYM'98: Proceedings of the 2nd conference on USENIX Windows NT Symposium*. 1998.

[69] LITTY, LIONEL: *Hypervisor-based Intrusion Detection*. Diplomarbeit, University of Toronto, 2005.

[70] LIU, YAN; GORTON, IAN: *Implementing Adaptive Performance Management in Server Applications*. In *SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, S. 12. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2973-9.

[71] LUCKHAM, DAVID: *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional, 2002. ISBN 0201727897.

[72] LUCKHAM, DAVID C.: *Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events.* Techn. Ber., Stanford, CA, USA, 1996.

[73] LUCKHAM, DAVID C.; KENNEY, JOHN J.; AUGUSTIN, LARRY M.; VERA, JAMES; BRYAN, DOUG; MANN, WALTER: *Specification and Analysis of System Architecture Using Rapide*. IEEE Trans. Softw. Eng., 21(4):336–355, 1995. ISSN 0098-5589.

[74] LUK, CHI-KEUNG; COHN, ROBERT; MUTH, ROBERT; PATIL, HARISH; KLAUSER, ARTUR; LOWNEY, GEOFF; WALLACE, STEVEN; REDDI, VIJAY JANAPA; HAZELWOOD, KIM: *Pin: building customized program analysis tools with dynamic instrumentation*. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, S. 190–200. ACM, New York, NY, USA, 2005. ISBN 1-59593-056-6.

[75] MANSOURI-SAMANI, MASOUD; SLOMAN, MORRIS; SLOMAN, MORRIS: *GEM: A Generalised Event Monitoring Language for Distributed Systems*. IEE/IOP/BCS Distributed Systems Engineering Journal, 4, 1997.

[76] MENON, ARAVIND; SANTOS, JOSE RENATO; TURNER, YOSHIO; JANAKIRAMAN, G. (JOHN); ZWAENEPOEL, WILLY: *Diagnosing performance overheads in the xen virtual machine environment*. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, S. 13–23. ACM, New York, NY, USA, 2005. ISBN 1-59593-047-7.

[77] NETHERCOTE, NICHOLAS; SEWARD, JULIAN: *Valgrind: a framework for heavyweight dynamic binary instrumentation*. SIGPLAN Not., 42(6):89–100, 2007. ISSN 0362-1340.

[78] OLSZEWSKI, MAREK; MIERLE, KEIR; CZAJKOWSKI, ADAM; BROWN, ANGELA DEMKE: *JIT instrumentation: a novel approach to dynamically instrument operating systems*. SIGOPS Oper. Syst. Rev., 41(3):3–16, 2007. ISSN 0163-5980.

[79] PASSING, JOHANNES: *Dynamic Tracing of Windows NT Kernel Mode Components*. Diplomarbeit, Hasso-Plattner-Institut an der Universität Potsdam, 2008.

[80] PASSING, JOHANNES; SCHMIDT, ALEXANDER; VON LÖWIS, MARTIN; POLZE, ANDREAS: *Ntrace: Function Boundary Tracing for Windows on IA-32*. In *Proceedings of the 16th Working Conference on Reverse Engineering*. 2009.

[81] POLZE, ANDREAS: *How to Partition a Workstation*. In *Proceedings of 8th IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*. 1996.

[82] POLZE, ANDREAS; PROBERT, DAVE: *Teaching Operating Systems: The Windows Case*. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, S. 298–302. ACM Press, New York, NY, USA, 2006. ISBN 1-59593-259-3.

[83] RICHLING, JAN; POLZE, ANDREAS: *Scheduling Server for Predictable Computing - an Experimental Evaluation*. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*. 1997.

[84] RUSSINOVICH, MARK E.; SOLOMON, DAVID: *Microsoft Windows Internals*. Microsoft Press, 4. Aufl., 2005. ISBN 978-0735619173.

[85] SALEHIE, MAZEIAR; TAHVILDARI, LADAN: *Autonomic computing: emerging trends and open problems*. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, S. 1–7. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-039-6.

[86] SCHÖBEL, MICHAEL: *Measuring Spin-Locks*. `http://www.dcl.hpi.uni-potsdam.de/research/WRK/2008/11/measuring-spin-locks/`, 2008.

[87] SCHÖBEL, MICHAEL; POLZE, ANDREAS: *A Runtime Environment for Online Processing of Operating System Kernel Events*. In *Proceeding of the Seventh International Workshop on Dynamic Analysis*. 2009.

[88] SCHMIDT, ALEXANDER: *A Performance Issue in Windows Timer Management?* `http://www.dcl.hpi.uni-potsdam.de/research/WRK/2007/08/a-performance-issue-in-windows-timer-management/`, 2007.

[89] SCHMIDT, ALEXANDER; VON LÖWIS, MARTIN; POLZE, ANDREAS: *KStruct: Preserving Consistency Through C Annotations*. In *PLOS '09: Proceedings of the 5th workshop on Programming languages and operating systems*. 2009.

[90] SCHMIDT, ALEXANDER; SCHÖBEL, MICHAEL: *Analyzing System Behavior: How the Operating System Can Help*. In *Proceedings of INFORMATIK 2007, LNI Nr. 110, Band 2*. 2007.

[91] SCHÖBEL, MICHAEL: *The Windows Monitoring Kernel*. In *Proceedings of the 2nd Ph.D. retreat of the HPI Research School on Service-Oriented Systems Engineering*. 2008.

[92] SCHÖBEL, MICHAEL; POLZE, ANDREAS: *Kernel-mode scheduling server for CPU partitioning: a case study using the Windows research kernel*. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, S. 1700–1704. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-753-7.

[93] SCHULT, WOLFGANG: *Architektur komponentenbasierter Systeme mit LOOM: Aspekte, Muster, Werkzeuge*. Doktorarbeit, Hasso-Plattner-Institut an der Universität Potsdam, 2009.

[94] SHANNON, CLAUDE ELWOOD: *Communication in the Presence of Noise*. Proceedings of the IRE, 37(1):10–21, 1949.
URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1697831`

[95] SHAPIRO, MICHAEL W.: *Self-healing in modern operating systems*. Queue, 2(9):66–75, 2005. ISSN 1542-7730.

[96] SHIN, MICHAEL E.; COOKE, DANIEL: *Connector-based self-healing mechanism for components of a reliable system*. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, S. 1–7. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-039-6.

[97] SLOTHOUBER, LOUIS P.: *A Model of Web Server Performance*, 1995.

[98] SOBANIA, JAN-ARNE: *A Tracing-Infrastructure for the Windows Research Kernel*. Diplomarbeit, Hasso-Plattner-Institut an der Universität Potsdam, 2009.

[99] TOY, BONNIE: *Linpack Benchmark Code*. http://www.netlib.org/benchmark/linpack-pc.c, 1994.

[100] TRÖGER, PETER: *CPU partitioning on Windows NT and SUN Solaris*, 2001.

[101] TSAFRIR, DAN; ETSION, YOAV; FEITELSON, DROR G.: *General Purpose Timing: The Failure of Periodic Timers*. Techn. Ber., School of Computer Science and Engineering, The Hebrew University of Jerusalem, 2005.

[102] TUNSTALL, CRAIG; COLE, GWYN: *Developing WMI Solutions: A Guide to Windows Management Instrumentation*. Pearson Education, Inc., 1. Aufl., 2002. ISBN 978-0201616132.

[103] WALKER, ROBERT J.; VIGGERS, KEVIN: *Implementing protocols via declarative event patterns*. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, S. 159–169. ACM, New York, NY, USA, 2004. ISBN 1-58113-855-5.

[104] WELBOURNE, EVAN; KHOUSSAINOVA, NODIRA; LETCHNER, JULIE; LI, YANG; BALAZINSKA, MAGDALENA; BORRIELLO, GAETANO; SUCIU, DAN: *Cascadia: A System for Specifying, Detecting, and Managing RFID Events*. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, S. 281–294. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-139-2.

[105] WHITE, WALKER; RIEDEWALD, MIREK; GEHRKE, JOHANNES; DEMERS, ALAN: *What is "next" in event processing?* In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, S. 263–272. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-685-1.

[106] WILE, DAVID S.: *Patterns of self-management*. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, S. 110–114. ACM Press, New York, NY, USA, 2004. ISBN 1-58113-989-6.

[107] WONG, CHEE SIANG; TAN, IAN; KUMARI, ROSALIND DEENA; WEY, FUN: *Towards achieving fairness in the Linux scheduler*. SIGOPS Oper. Syst. Rev., 42(5):34–43, 2008. ISSN 0163-5980.

[108] WU, EUGENE; DIAO, YANLEI; RIZVI, SHARIQ: *High-performance complex event processing over streams*. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, S. 407–418. ACM, New York, NY, USA, 2006. ISBN 1-59593-434-0.

[109] YAGHMOUR, KARIM; DAGENAIS, MICHEL R.: *Measuring and Characterizing System Behavior Using Kernel-Level Event Logging*. In *Proceedings of the USENIX Annual Technical Conference*, S. 13–26. 2000.

[110] YANG, LIANG; GOHAD, TUSHAR; GHOSH, PAVEL; SINHA, DEVESH; SEN, ARUNABHA; RICHA, ANDREA: *Resource mapping and scheduling for heterogeneous network processor systems*. In *ANCS '05: Proceedings of the 2005 symposium on Architecture for networking and communications systems*, S. 19–28. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-082-5.

[111] ZHANG, RONGHUA; LU, CHENYANG; ABDELZAHER, TAREK F.; STANKOVIC, JOHN A.: *ControlWare: A Middleware Architecture for Feedback Control of Software Performance*. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, S. 301. IEEE Computer Society, Washington, DC, USA, 2002. ISBN 0-7695-1585-1.

ANHANG A

# WMK EREIGNISTYPEN

Der *Windows Monitoring Kernel* (WMK) wird in Kapitel 3 beschrieben. In diesem Anhang werden die Datenstrukturen der verfügbaren Ereignistypen dokumentiert.

## Allgemeine Ereignisinformationen

**Listing A.1:** WMK Ereignisheader

```
typedef struct _WMK_EVENT_HEADER {

  // general information
  ULARGE_INTEGER    TimeStamp;
  UCHAR             Type;

  // execution context
  UCHAR             CpuId;
  HANDLE            ProcessId;
  HANDLE            ThreadId;

  // event data
  USHORT            BodySize;

} WMK_EVENT_HEADER, *PWMK_EVENT_HEADER;
```

**Listing A.2:** WMK Ereignisnummern

```
#define WMK_E_NODATA_EVENT_ID            0
#define WMK_E_CUSTOM_EVENT_ID            1

#define WMK_E_PROCESS_CREATION_ID        2
#define WMK_E_PROCESS_TERMINATION_ID     3
#define WMK_E_THREAD_CREATION_ID         4
#define WMK_E_THREAD_TERMINATION_ID      5
#define WMK_E_WAIT_EVENT_ID              6
#define WMK_E_SYSCALL_ID                 7
#define WMK_E_CONTEXT_SWITCH_ID          8
#define WMK_E_WAIT_RELEASE_ID            9
#define WMK_E_SYSCALL_EXIT_ID           10
#define WMK_E_QUANTUM_END_ID            11
#define WMK_E_CREATE_OBJECT_ID          12
```

```
#define WMK_E_CREATE_FILE_ID           13
#define WMK_E_IO_TRANSFER_ID           14
#define WMK_E_LOCK_ID                  15
#define WMK_E_PAGE_FAULT_ID            16
// ...

#define WMK_EVENT_COUNT                17

#define WMK_E_UNKNOWN                  0xFE
#define WMK_CHECKPOINT_ID              0xFF // MAXUCHAR
```

## Thread- und Prozess-Erzeugung

**Listing A.3:** WMK Thread Erzeugung

```
typedef struct _WMK_E_THREAD_CREATION {

  HANDLE            ProcessId;
  HANDLE            ThreadId;
  PVOID             ThreadObjAddress;
  PVOID             OffsetIntoImage;
  PVOID             ThreadStartAddress;
  USHORT            StringLength;

} WMK_E_THREAD_CREATION, *PWMK_E_THREAD_CREATION;
```

**Listing A.4:** WMK Prozess Erzeugung

```
typedef struct _WMK_E_PROCESS_CREATION {

  HANDLE            ProcessId;
  HANDLE            ParentProcessId;
  PVOID             ProcessObjAddress;
  PVOID             SectionBaseAddress;

} WMK_E_PROCESS_CREATION, *PWMK_E_PROCESS_CREATION;
```

## Objekterzeugung

**Listing A.5:** WMK Erzeugung einer Objektreferenz

```
typedef struct _WMK_E_CREATE_OBJECT {

  PVOID             Object;
  USHORT            ObjectTypeLength;
  USHORT            ObjectNameLength;

} WMK_E_CREATE_OBJECT, *PWMK_E_CREATE_OBJECT;
```

**Listing A.6:** WMK Erzeugung einer Dateireferenz

```
typedef struct _WMK_E_CREATE_FILE {

  int                  Status;
  USHORT               FileNameLength;

} WMK_E_CREATE_FILE, *PWMK_E_CREATE_FILE;
```

### Systemaufrufe

**Listing A.7:** WMK Systemaufruf

```
typedef struct _WMK_E_SYSCALL {

  ULONG                SyscallNr;

} WMK_E_SYSCALL, *PWMK_E_SYSCALL;
```

### Scheduling

**Listing A.8:** WMK Kontextwechsel

```
typedef struct _WMK_E_CONTEXT_SWITCH {

  char                 Priority;
  char                 Quantum;

} WMK_E_CONTEXT_SWITCH, *PWMK_E_CONTEXT_SWITCH;
```

### Synchronisation

**Listing A.9:** WMK Synchronisationsaufruf

```
typedef struct _WMK_E_WAIT_EVENT {

  UCHAR                Type;
  BOOLEAN              Flag; // == TRUE  <-> start waiting event
                             // == FALSE <-> finished waiting event
  LONG                 ReturnValue;
  PVOID                Object;
  ULONG                MultipleObjectsDataSize;

} WMK_E_WAIT_EVENT, *PWMK_E_WAIT_EVENT;
```

Listing A.10: WMK Freigabe eines Synchronisationsobjektes

```
typedef struct _WMK_E_WAIT_RELEASE {

  PVOID             Object;
  LONG              PriorityBoost;
  UCHAR             Type;

} WMK_E_WAIT_RELEASE, *PWMK_E_WAIT_RELEASE;
```

Listing A.11: WMK feingranulare Synchronisation

```
#define WMK_LOCK_FLAG_MUTEX       0x0000
#define WMK_LOCK_FLAG_SPINLOCK    0x0001
#define WMK_LOCK_FLAG_ERESOURCE   0x0002
#define WMK_LOCK_FLAG_PUSHLOCK    0x0003

#define WMK_LOCK_FLAG_LOCK        0x0010
#define WMK_LOCK_FLAG_RELEASE     0x0020

#define WMK_LOCK_FLAG_EXCLUSIVE   0x0040
#define WMK_LOCK_FLAG_SHARED      0x0080

#define WMK_LOCK_FLAG_TRY         0x0100
#define WMK_LOCK_FLAG_UNSAFE      0x0200

typedef struct _WMK_E_LOCK {

  PVOID             ObjAddress;
  USHORT            Flags;

} WMK_E_LOCK, *PWMK_E_LOCK;
```

## Eingabe und Ausgabe

Listing A.12: WMK Ein- und Ausgabeoperationen

```
typedef struct _WMK_E_IO_TRANSFER {

  USHORT            IOType;
  ULONG             TransferCount;

} WMK_E_IO_TRANSFER, *PWMK_E_IO_TRANSFER;
```

## Seitenzugriffsfehler

**Listing A.13:** WMK Seitenzugriffsfehler

```
#define WMK_PAGE_FAULT_UNKNOWN      0
#define WMK_PAGE_FAULT_PAGEFILE     1
#define WMK_PAGE_FAULT_MAPPEDFILE   2
#define WMK_PAGE_FAULT_TRANSITION   3
#define WMK_PAGE_FAULT_PROTOPTE     4
#define WMK_PAGE_FAULT_DEMANDZERO   5

typedef struct _WMK_E_PAGE_FAULT {

  PVOID             VirtualAddress;
  USHORT            Type;

} WMK_E_PAGE_FAULT, *PWMK_E_PAGE_FAULT;
```