Gegenüberstellung verschiedener Paradigmen zur Behandlung von Fehlern und Ausnahmen in verteilten Systemen

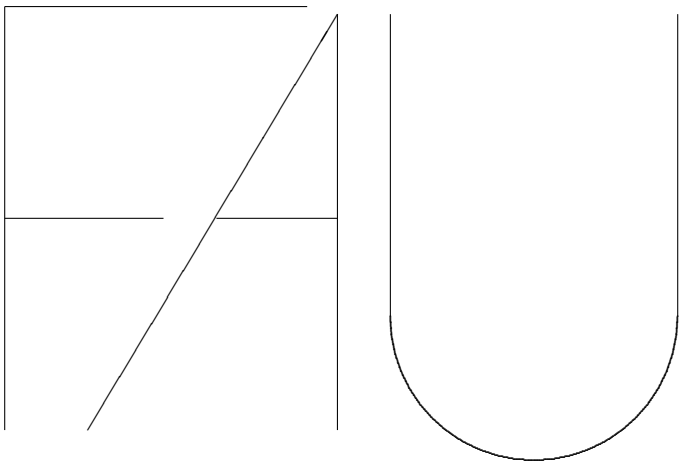
Bachelorarbeit

Alexander A. Schmidt

Lehrstuhl für Informatik 6
(Datenmanagement)

Department Informatik
Technische Fakultät

Friedrich Alexander-
Universität
Erlangen-Nürnberg



Gegenüberstellung verschiedener Paradigmen zur Behandlung von Fehlern und Ausnahmen in verteilten Systemen

Bachelorarbeit im Fach Informatik

vorgelegt von

Alexander A. Schmidt

geb. 02.01.1987 in München

angefertigt am

Department Informatik
Lehrstuhl für Informatik 6 (Datenmanagement)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Univ.-Prof. Dr.-Ing. habil. Richard Lenz
Dipl.-Inf. Christoph P. Neumann

Beginn der Arbeit: 12.01.2012
Abgabe der Arbeit: 30.04.2012

Erklärung zur Selbständigkeit

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Informatik 6 (Datenmanagement), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 30.04.2012

(Alexander A. Schmidt)

Kurzfassung

Gegenüberstellung verschiedener Paradigmen zur Behandlung von Fehlern und Ausnahmen in verteilten Systemen

Nachdem Goodenough im Jahr 1975 den Grundstein für den erfolgreichen Einsatz von Exceptions in lokalen Systemen gelegt hat, wird nun eine Weiterführung dieses Paradigmas in verteilten Systemen angestrebt, die eine besondere Herausforderung darstellen. In verteilten Systemen existieren viele diametral zueinander stehende Kriterien welche die Architektur des verteilten Systems bestimmen. Daraus ergeben sich unterschiedliche Forderungen an ein System, das Exceptionhandling in verteilten Systemen realisieren soll. Bisher findet sich in der Literatur noch keine Arbeit die sich damit beschäftigt einen Überblick über Exceptionhandling-Systeme für verteilte Systeme herzuzstellen. Die Ergebnisse nennenswerter Forschergruppen werden recherchiert, zusammengetragen und mittels einer Klassifikation für verteilte Systeme gegenübergestellt werden.

Im ersten Schritt der Arbeit wird kurz erschlossen warum Exceptions eine notwendige Erfindung waren. Es werden die unterschiedlichen Auslegungen und konzeptuelle Richtlinien im Bereich Exceptions dargestellt. Eine Vertiefung der konzeptuellen Richtlinien wird in einem Kapitel über Exception Pattern weitergeführt.

Für lokale Systeme wird eine umfassende Klassifikation aufgestellt und anhand dieser werden moderne Programmiersprachen in ihrem Leistungsumfang hinsichtlich. Exceptionhandling bemessen.

Anschließend erfolgt ein Einblick in die Implementierung von Exceptions und aufbauend darauf ein Abschnitt wie Exceptions auf einem lokalen System nach Leistung optimiert werden können.

Nachdem die erforderlichen Grundlagen für lokale Systeme geschaffen ist, wird eine umfassende erörternde Klassifikation für verteile Systeme aufgestellt und anhand dieser werden Ansätze, die den Stand der Technik bilden, in ihren Leistungen bewertet.

Abstract

Comparison of different paradigms for handling errors and exceptions in distributed systems

After Goodenough set the milestone for successfull usage of exceptions in local systems in 1975, exceptions should be set to good use in distributed systems. In distributed systems many criteria, which influence each other, have to be considered that decide on the realisation of an architecture and set up demands on a system that provides exception handling in distributed systems. Up to today there exists no paper in literature that tries to give an united overview about an exceptionhandling-system for distributed systems by collecting information from research groups and comparing them by an classification for distributed systems consulting in an evaluation. At first it is shown why exceptions have been introduced. After this there will be a discussion about different notions and conceptual design-rules of exceptions. A deeper insight in design-rules will be delivered in chapter about exception pattern.

An exhausting classification for local systems will be created and used to compare modern programming languages in their feature set concerning exception handling.

Concluding this an insight in the implementation of exceptions is given and based on this in the concluding section it is hown how exceptions can be tweaked for performance in local systems.

After the essential basics for local systems haven been established, an exhausting classification for distributed systems will be created and state-of-the-art approaches will be evaluted by their feature set.

At the end further research topics are pointed out which are interesting to pursue.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

1 Einführung

Softwareentwicklung ist zu einer ertragreichen und etablierten Industrie herangewachsen. Jedoch ist es in der Praxis noch ein junges wissenschaftliches Feld. Das Anwendungsfeld der Softwareentwicklung ist von der Erstellung validen, verifizierten und gut getesteten Programmcodes für spezifische Algorithmen, zu ausladenden Produktgruppen mit vollständiger Onlineintegration und zahlreichen Optionen, denen sich der Benutzer bedienen kann, übergegangen. Neu erscheinende Software wird stetig komplexer. Mit der Zunahme der Komplexität steigt auch die Wahrscheinlichkeit von Fehlern pro Lines of Code. Die Verifizierung bei komplexen und großen Softwareprojekten ist kaum möglich, weil die Anzahl möglicher Verzweigungen im Programmkontrollfluss zu groß geworden ist.

1.1 Motivation und Hintergründe

Statt statisch zu garantieren dass keine Fehler auftreten können (Verifikation), wird ein Mechanismus benötigt, der zur Laufzeit auftretende Fehler behandeln kann. Diese Mechanismen werden Ausnahmebehandlung oder auch Exceptionhandling genannt. Sie behandeln nicht nur Fehler, sondern liefern weitere Merkmale, die zur Robustheit und Sicherheit eines Programmes beitragen, die durch statische Verifikation nicht möglich wären. Durch Exceptionhandling können Produktionskosten gesenkt, Software stabiler und flexibel gestaltet werden. Exceptionhandling ist in der Informatik relativ neu, wurde aber im Einsatz von lokalen Systemen erschöpfend von Forschergruppen bearbeitet und die Forschung in diesem Gebiet als für abgeschlossen erklärt. Der nächste Schritt ist es, Exceptionhandling in verteilten Systemen zu erforschen. Es gibt eine große Anzahl unterschiedlicher verteilter Systemarchitekturen. Jede dieser verfolgt eine andere Konstellation von Kriterien die durch die Systemarchitektur erfüllt werden sollen. Zum Beispiel haben in einer Systemarchitektur kurze Antwortzeiten höchste Priorität, während in einer anderen Systemarchitektur die dezentrale Struktur und Unabhängigkeit der einzelnen Teilnehmer im Vordergrund stehen. Ein Exceptionhandling-Mechanismus, der in einem verteilten System im Einsatz ist, muss in Einklang mit den Kriterien der Systemarchitektur sein. Dies hat zur Folge, dass es unterschiedliche Exceptionhandling-

Mechanismen geben muss, so wie es unterschiedliche verteilte Systemearchitekturen gibt, um der Erfüllung der Kriterien gerecht zu werden.

1.2 Zielsetzungen

An welchen Exceptionhandling-Mechanismen wird geforscht? Gibt es für eine bestimmte verteilte Systemarchitektur mehrere Ansätze für einen Exceptionhandling-Mechanismus? Gibt es einen Mechanismus, der besser als alle anderen existierenden Mechanismen ist und als allgemeingültige Referenz für die bestimmte verteilte Systemarchitektur gelten kann? Das ultimative Ziel wäre die Schaffung eines Exceptionhandling-Mechanismus, der für den EInsatz mit jeder verteilten Systemarchitektur konfigurierbar ist. Solch ein Mechanismus würde die Forschung über Exceptionhandling in verteilten System abschließen. Der Beitrag dieser Arbeit konzentriert sich auf die Beantwortung dieser Fragen für die verteilte Systemarchitektur, in der lose gekoppelte Systeme miteinander agieren und ihre Unabhängigkeit gewahrt bleiben soll.

1.3 Methodik

Zur Einarbeitung in die Thematik erfolgt zu Beginn die Untersuchung der Frage weshalb Exceptions eingeführt wurden. Es existiert leider keine allgemein anerkannte Definition von Exceptions. Aus diesem Grund werden die Definitionen zu dem Begriff Exception von verschiedenen Forscher(gruppen) zur Abgrenzung gegenüber gestellt. Im weiteren Verlauf der Arbeit wird Exception meist losgelöst von der semantischen Bedeutung, und nur als Sprachkonstrukt betrachtet werden. An gewissen Stellen ist es erforderlich, die Semantik welche sich hinter dem Sprachkonstrukt Exception verbirgt zu benennen. Dies betrifft Implementierungsansätze, die nur für bestimmte Exceptions ausgelegt sind. Nach der Definition des Exceptionsbegriffs wird eine Abgrenzung zur Fehlerbehandlung vorgenommen. Dies ist eng mit der Definition der Exception verbunden.

Nachdem die theoretischen Grundlagen geschaffen wurde, wird erarbeitet wie das Sprachkonstrukt Exception in einer Programmmiersprache modelliert wird. Dazu gibt es die Befehlswörter ”try” und ”catch”, welche die Basisfunktionalität des Exceptionhandlings herstellen.

Bevor Exceptionhandling in verteilten Systemen betrachtet wird, muss Exceptionhandling auf lokaler Ebene verstanden sein. Deswegen wird ausgehend von der Vorlage von Buhr, die Klassifikation eines Exceptionhandlingmechanismus aufgestellt. Durch

diese Klassifikation findet eine komplette Analyse des Exceptionhandling-Prozesses statt. Zunächst kann Exceptionhandling in das Signalisieren, Weiterreichen und Behandeln von Exceptions eingeteilt werden. Diese einzelnen Phasen werden weiter analysiert und alle bekannten Alternativen für jedes Kriterium wird enumeriert.

Nach der Klassifikation, die eine Spezifikationscheckliste für jeden beliebigen Exceptionhandlingmechanismus darstelt, wird anhand von Java die Implementierung von dem try-catch und finally Konstrukt in Bytecode betrachtet. Dieser Schritt, dient dazu die anschließende Leistungsanalyse für lokale Systeme nachvollziehen zu können. In der Leistungsanalyse werden alternative Ansätze zur Implementierung von Exceptions besprochen und welchen Einfluss sie auf die Effizienz haben.

Danach werden die modernen Programmiersprachen C++, Python und Java in ihrem Leistungsumfang verglichen und ihre Unterschiede kenntlich gemacht.

Anschließend soll der Übergang von lokalen auf verteilte Systemen folgen. Dazu müssen zuerst Grundlagen gesammelt werden, welche verteilte Architekturen möglich sind. Die unterschiedlichen Architekturen für verteilte Systemen sind für unterschiedliche Anwendungszwecke und damit anderen Kriterien ausgerichtet (Zielkriterien). Der Exceptionhandlingmechanismus für verteilte Systeme muss auf der vorliegenden Architektur aufbauen und sollte möglichst geringen gegenteiligen Einfluss auf die Erfüllung der Zielkriterien ausüben.

Im Fokus der Arbeit steht die Gegenüberstellung von Exceptionhandlingmechanismen, die auf lose-gekoppelte verteilte Systeme ausgerichtet sind. Das Zielkriterium, das so wenig wie möglich beeinträchtigt werden soll, ist die lose Kopplung.

1.4 Aufbau der Arbeit

Im ersten Schritt der Arbeit wird kurz erschlossen warum Exceptions eine notwendige Erfindung waren. (Kapitel 2) Anschließend daran werden die unterschiedlichen Auslegungen und konzeptuelle Richtlinien von Exceptions angeschlossen. (Kapitel 2.2) Eine Vertiefung der konzeptuellen Richtlinien wird im Kapitel über Exception Pattern weitergeführt werden. (Kapitel 9)

Für lokale Systeme wird eine umfassende Klassifikation aufgestellt (Kapitel 5) und anhand dieser werden moderne Programmiersprachen in ihrem Leistungsumfang (Kapitel 6) bzgl. Exceptionhandling bemessen werden.

Anschließend erfolgt noch ein Einblick in die Implementierung von Exceptions (Kapitel 7) und aufbauend darauf ein Abschnitt wie Exceptions auf einem lokalen System nach Leistung optimiert werden können. (Kapitel 8)

Nachdem die erforderlichen Grundlagen für lokale Systeme geschaffen ist, wird eine umfassende erörternde Klassifikation für verteile Systeme (Kapitel 10) aufgestellt und anhand dieser werden state-of-the-art Ansätze in ihren Leistungen bewertet werden.

Abschließend werden im Ausblick (Kapitel 11) noch weiterführende Richtungen in der Forschung aufgezeigt, geklärt welche Schritte noch zu tun sind, und welche Ziele für Exceptionhandling in verteilten Systemen von einer großen Zahl von Forschern angestrebt werden.

1.5 Leseanleitung

Wenn neue Begriffe eingeführt werden, auf die im weiteren Verlauf der Arbeit häufig Bezug genommen wird, so werden diese fett gedruckt dargestellt. Fachbegriffe von anderen Autoren werden auch fett gedruckt dargestellt. Die Großteil der Forschung in der Informatik ist in Englisch gehalten. Da diese Arbeit in Deutsch verfasst ist, wurden die meisten Begriffe wenn möglich ins Deutsche übersetzt. In einigen Fällen ist es aber nicht zweckdienlich die englischen Begriffe ins Deutsche zu überführen, wenn dadurch der Bezug zu anderer Literatur verloren geht und eine Zuordnung des Begriffes nur durch Nennung des ursprünglichen englischen Wortes hergestellt werden kann. In diesen Fällen wird gleich der englische Begriff als deutsches Ersatzwort verwendet. Auf Wortkombinationen aus deutschen und englischen Wörtern, wurde so weit wie nur möglich verzichtet.

2 Was sind Exception?

Um Exceptionhandling im richtigen Kontext einordnen zu können, soll die Frage beantwortet werden, wie es zur Entstehung der Exceptions gekommen ist. Deswegen wird im folgenden ein kurzer historischer Werdegang der Software-Entwicklung skizziert, aus der die Exceptions hervorgetreten sind. Anschließend wird eine Definition der Exception erörtert.

2.1 Historischer Kontext von Exceptions

Die Entwicklung der Exception kann in die drei folgenden Phasen eingeteilt werden. Zunächst wurde die konstruktive Fehlerbehandlung mit der analytischen Fehlerbehandlung ergänzt. Dies geschah durch die Erweiterung des Rückgabewertes zur Fehlerbehandlung. Dadurch wurde aber die Semantik des Rückgabewerts, die Unterscheidung in Ergebnisse und Fehler erschwert. Die Fehlerbehandlung ist durch diese Technik auch an seine Grenzen gestoßen, so dass in der dritten Phase eine Vereinheitlichung der Fehler-Signalisierung und Fehler-Behandlung durchgeführt wurde.

2.1.1 Von der konstruktiven zur analytischen Fehlerbehandlung

Zu der Zeit als man noch Lochkarten verwendete, erhielt die Lesemaschine nur Prüfbits, um zu erkennen ob die Lochkarten fehlerfrei erstellt waren. Mit dem vom Programmierer abgesegneten Programmcode hatte dies nichts mehr zu tun. Es war nur eine Überprüfung über die Konsistenz des Mediums. Man verließ sich darauf dass der Programmierer zuverlässigen Programmcode geschrieben hatte. Stimmte das Ergebnis der Berechnung nicht, so war die gebuchte Rechenzeit ohne brauchbares Ergebnis verstrichen und nach mühsamer Behebung des Irrtums im Programmcode musste erneut Rechenzeit gebucht werden. Schon damals war es teuer, wenn man im Entwurf nicht ordentlich gearbeitet hat.

Validierung und Verifikation waren nur in der Entwurfsphase (d.h. **offline**) möglich, nicht zur **runtime**. Mit dem Aufkommen der Multi-User Maschinen wurde die Programmstruktur verändert. Das Programm beanspruchte nicht mehr die gesamte Maschine für sich. Allgemeine generisierbare Funktionen wurden in das Betriebssystem ausgelagert. Um nun auf Abweichungen im Kontrollfluss des Programmes reagieren zu können, die u.a. durch konkurrierende Beanspruchung von Ressourcen auftraten, mussten die Betriebssystemkomponenten der aufrufenden Komponente mitteilen, ob die Aktion **erfolgreich** war oder **nicht**.

Hier wurde erstmals ein Mechanismus zur analytischen Fehlerbehandlung eingeführt. Somit wurden die boolschen Werte true und false zur Verzweigung in eine Ausnahmebehandlung verwendet. Alsbald wurden anstatt von boolschen Werten detailliertere Fehlercodes als Ganzzahlen zurückgegeben. Der Rückgabewert 0 bedeutet erfolgreiches Ausführen der Funktion, jeder andere Rückgabewert stellte einen Fehlercode dar. Doch oft galt auch eine zurückgegebene 1 als erfolgreiches Ausführen der Funktion. [LS98]

Dies etablierte sich sehr schnell zwischen allen Schichten, in welchen eine aufrufende Komponente einer aufgerufenen Komponente begegnete. Es konnten Komponenten in nahezu beliebiger Tiefe aufgerufen werden. Wenn aber eine irreparable Systemkonfiguration entstanden ist, dann wurde das Programm mit ”exit(1)” beendet und der Anwender erhielt auch einen Fehlercode, dessen Bedeutung er in dem Handbuch zur Software nachschlagen konnte.

2.1.2 Trennung von Ergebnissen und Fehler-Signalisierungen

Jedoch war dies unbefriedigend, da nur per Konvention, Pattern, erfolgreiche Fehlerbehandlung in einem Projekt mit mehreren Programmierern durchgesetzt werden konnte. Hier vermischen sich aber zwei unterschiedliche Semantiken miteinander, und es war erforderlich alle Fehlerwerte abzufragen, bevor man mit dem Ergebnis weiterrechnen konnte. Einen etwas spezielleren Fall stellen als Beispiel für diese Kategorie die Fließkommazahlendarstellung nach IEE-754 dar, die den Ergebniswertebereich mit dem Fehlerwert NaN (Not a Number) verknüpfen.Weitere arithmetische Berechnungen mit dem Wert NaN und beliebigen anderen Werten liefern stets den Fehlerwert NaN.

Wegen der Vermischung der Semantiken wurde nach einem Weg gesucht, auf andere Weise als dem Rückgabewert, dem Aufrufer mitzuteilen, welche Fehlerart aufgetreten ist. Es wurde eine globale Variable eingeführt (in der Programmiersprache C, hieß sie errno), die den Fehlerwert aufnimmt. Kann eine Funktion ihre Aufgabe nicht erfüllen, so schreibt sie in die globale Variable ihren spezifischen Fehlercode. Je nach Semantik

des Rückgabewertes der Funktion wird entweder kein Rückgabewert zurückgegeben, der eindeutig auf einen Fehler hinweist, oder es wird ein einziger Rückgabewert zurückgegeben, der eindeutig einen Fehler signalisiert oder es können unterschiedliche Fehlerwerte neben dem Wertebereich der Funktion zurückgegeben werden. Die Funktion kann im Fall eines Fehlers auch eine globale Variable entsprechend setzen, sodass durch Lesen dieser Variablen noch genauere Informationen zu dem Fehler ermittelt werden können.

2.1.3 Durchsetzung eines einheitlichen Fehler-Signalisierungs- und Behandlungsmechanismus

Diese zahlreichen Möglichkeiten der Fehler-Signalisierung vereitelt eine einheitliche Behandlung von Fehlern. Der Programmierer ist auf die Dokumentation der Schnittstellen angewiesen. Diese sind nur selten vollständig. Einige Schnittstellen erfordern nur eine Überprüfung des Rückgabewerts, wohingegen andere ein Lesen der globalen Fehlervariable erfordern. Eine Aktualisierung von fremden Schnittstellen ist nur nach ausgiebigem Blick der Dokumentation durchführbar. Veränderte oder neue Fehlercodes sind zu beachten und der Glaube dass die Dokumentation vollständig ist.

Dieser uneinheitliche Gebrauch von Fehler-Signalisierungen, die unzureichende Zuverlässigkeit auf Vollständigkeit der Schnittstellenbeschreibungen, sowie die uneinheitliche Behandlung der Fehler je nach Schnittstelle führten zu dem Wunsch der Vereinheitlichung der Fehler-Signalisierung und der Behandlung von Fehlern.

Exceptions stellen neben dem Rückgabe-Kontrollfluss einen zweiten, alternativen Weg bereit, um Fehler zu signalisieren. Der Kontrollfluss wird in Bereiche der aufrufenden Komponente geleitet, der die Logik enthält auf die zu erwartende aufzutretenden Fehler zu reagieren. Doch dieses Konzept der Exceptions geht über die Repräsentation von Fehlern heraus. Deswegen wurde auch der Begriff Exception geformt. Die Bedeutung der Exception wird im nächsten Abschnitt behandelt.

2.2 Definition

[LS79]: Der Begriff Exception grenzt sich zu dem Begriff Fehler ab, da eine Exception nicht impliziert dass etwas schief gelaufen ist. Diese Notwendigkeit wird aus der Erkenntnis gestützt dass ein Ereignis, dass von einer Prozedur als Fehler angesehen wird von einer anderen Prozedur nicht als Fehler wahrgenommen wird. Eine Exception impliziert, dass etwas Außergewöhnliches eingetreten ist, aber selbst dies könnte irreführend sein. So

könnten Exceptions dazu dienen, vorausgesetzt sie sind effizient genug, Informationen für normale und gewöhnliche Situationen zu übermitteln.

Yemini und Berry [YB85a] definieren eine **exceptional condition** in einer Operationsausführung als einen Zustand, der nicht die Vorbedingung für die Eingabe der Operation erfüllt.

Nach Knudsen [Knu87] ist der Auftritt einer Exception, ein Zustand, in dem die normale Berechnung nicht fortfahren kann. Zum Fortfahren ist eine außergewöhnliche Berechnung notwendig.

Nach Gehani [Geh92a] ist eine Exception ein Fehler oder ein Ereignis, das unerwartet oder unregelmäßig auftritt, wie beispielsweise eine Division durch Null oder ungenügender Arbeitsspeicher.

Nach Miller et. al [MT97] ist eine Exception ein abnormaler Berechnungszustand. Der Auftritt einer Exceptions ist die Instanziierung einer Exception. Eine Exception wird durch einen Fehler (einen invaliden Systemzustand, der nicht erlaubt ist), eine Abweichung (ein invalider Systemzustand, der erlaubt ist), eine Notifikation (einen Aufrufer darüber informiert, dass ein angenommener oder vorheriger Systemzustand sich geändert hat) oder ein Idiom (andere Verwendung, in der das Auftreten einer Exception selten statt abnormal ist, wie beispielsweise das Erkennen eines End Of File) ausgelöst.

Die im Allgemeinen akzeptierte Definition von Exception ist die Vereinigung von Fehler, außergewöhnlicher Fall, seltene Situation und ungewöhnliches Ereignis. [LS98]

Nach Goodenough [Goo75] haben exception conditions folgende Eigenschaften:

- die vollständige Bedeutung einer Exception ist nur außerhalb der Operation, der diese erkannt hat, möglich. (Dem stimmt auch Yemini et. al [YB85b] zu)
- dem Aufrufer kann es gestattet sein, die Operation zu terminieren, falls eine Exception erkannt wurde
- der Aufrufer entscheidet ob eine default-Antwort zu der Exception aufgerufen werden soll. Die default-Antwort ist innerhalb der Operation definiert, welche die Exception signalisiert.

Nach Goodenough [Goo75] dienen Exceptions dazu Operationen zu verallgemeinern, um sie in breiteren verschiedenen Kontexten benutzen zu können als das gewöhnlich der Fall ist. Dies wird durch Exceptions erreicht, da sie die Domäne der Operation (die Menge von Eingaben vergrößert auf denen Auswirkungen definiert sind) oder die Range der Operation (Effekte die erlangt werden, wenn bestimmte Eingaben verarbeitet werden) erweitern.

Die Verwendung von Exceptions können nach Goodenough [Goo75] klassifiziert werden. Demnach findet die Verwendung von Exceptions statt,

- um auf ein bevorstehendes oder tatsächliches Versagen einer Operation reagieren zu können
- um die Bedeutung eines validen Ergebnisses oder die Umstände, unter denen es gewonnen wurde, zu unterstreichen. Dadurch wird das Ergebnis einer Operation klassifiziert. In diesem Fall werden die Nachbedingungen der Operation erfüllt, aber der Aufrufer benötigt weitere Informationen, um das Ergebnis korrekt interpretieren zu können.
- um einem Aufrufer, die Operation beobachten (monitoring) lassen zu können, wie den Berechnungfortschritt zu messen oder zusätzliche Informationen anbieten zu können, sollten bestimmte Zustände eintreten. Wenn eine monitoring exception signalisiert wird, kann der Aufrufer eventuell eine Terminierung der Operation wünschen. Oft wird der Aufrufer gezwungen sein, die Operation fortfahren zu lassen, da es entweder nicht möglich oder ökonomisch nicht tragbar wäre, die Operation an jedem Beobachtungspunkt sauber zu beenden. Als Beispiel kann eine Suchoperation angeführt werden, die Teilergebnisse per Signalisierung einer Exception übergeben kann, und der Aufrufer daraufhin entscheiden kann, ob das nächste Ergebnis gesucht werden soll. Mittels Exceptions wird der Zustand der Operation gehalten und die Operation kann fortfahren ohne dieselben Berechnungen erneut ausführen zu müssen.

2.3 Definitionen

Hier werden alle grundlegende Begriffe für dieses und alle weiteren Kapitel festgelegt.

handler clause: Sammlung von Handlern auf gleicher Ebene, die dem selben Codeblock zugeordnet worden sind

guarded block: Codeblock, dem Handler zugeordnet worden sind

unguarded block: Codeblock, dem kein Handler zugeordnet worden ist

execution: Prozess der in einem Computer bzw. in einer virtuellen Maschine die Instruktionen eines Computerprogrammes ausführt. Sie besteht aus Zustandsinformationen, lokalen und globalen Daten, der aktuellen Ausführungsposition und routine activation records.

source execution: Prozess in Ausführung, der eine Exceptioninstanz geworfen hat

faulting execution: Prozess in Ausführung, der den Kontrollfluss nach Empfang einer Exceptioninstanz ändert

exceptional condition: Zustand eines Artefakts, der als außergewöhnlich erkannt wurde. Das Erkennen beinhaltet in der exception-controlled Technik das Werfen einer Exception.

exception: ist eine geeignete Datenrepräsentation, die das Auftreten einer exceptional condition signalisiert

explicit exception polling: ist eine Technik, in welcher das Abfragen, ob eine Exception vorliegt vom Programmierer explizit durchgeführt werden muss. Wird eine solche Abfrage vergessen, so wird die exceptional condition verborgen. Wird die Abfrage auf einen Rückgabwert ausgeführt, so wird möglicherweise mit einem falschen Zwischenergebnis weitergerechnet. Beinhaltet die Abfrage aber eine globale Variable, die bei Auftreten einer exceptional condition EC1 gesetzt wird und wird diese nicht auf einen Default-Wert zurückgesetzt ehe sie erneut durch eine weitere exceptional condition EC2 gesetzt werden kann, so kann eine darauffolgende Verzweigungsabfrage für EC2 fälschlicherweise eine exceptional condition für EC2 erkennen, obwohl es sich um die exceptional condition EC1 handelt.

exception-controlled: Technik, bei der nach dem Erkennen einer exceptional condition durch das Werfen einer Exception der Kontrollfluss - wenn nötig - implizit geändert wird.

point of raise: die Stelle im Kontrollfluss an dem eine Exception geworfen wird

2.4 Klassifizierung von Exceptionursachen

2.4.0.1 Klassifizierung nach Goodenough

Nach Goodenough [Goo75] lassen sich Exceptionsursachen in **range failures** und **domain failures** einteilen. Die Exceptionursache stellt bestimmte Anforderungen an ein Behandlungsmechanismus.

Range failure tritt auf, wenn eine Operation, entweder keine oder niemals in der Lage sein wird valide Ergebnisse (Nachbedingungen werden nicht erfüllt) ausgeben zu können. Zum Beispiel können keine validen Ergebnisse ausgegeben werden, wenn eine Leseoperation keinen Datensatz lesen kann, da das Dateiende schon erreicht wurde. Eine

Leseoperation könnte niemals in der Lage sein valide Ergebnisse auszugeben, wenn ein Paritätsfehler beim Lesen auftritt.

- Der Aufrufer muss die Operation abbrechen können. Die Operation muss signalisieren können, dass sie nicht mit der Ausführung fortfahren kann. Bei der Terminierung der Operation müssen alle bereits verursachte Auswirkungen, die Seiteneffekte auslösen würden, rückgängig gemacht werden können.
- Der Operation muss mitgeteilt werden können es erneut zu versuchen.
- Die Fähigkeit die Operation zu terminieren und Teilergebisse an den Aufrufer zu übergeben, gegebenfalls mit zusätzlichen Informationen um das Ergebnis korrekt interpretieren zu können.

Domain failure tritt auf, wenn eine Vorbedingung einer Operation nicht erfüllt wird. Der Aufrufer muss genügend Informationen von der Operation erhalten, damit er die Eingabe so anpassen kann, dass sie den Vorbedingungen genügen. Dazu kann es nötig sein, dass der Aufrufer auf die Operanden der Operation zugreifen kann.

2.4.0.2 Klassifizierung nach Eder et. al

Nach Eder et al. [EL95] können Exceptions in folgende vier Klassen eingeteilt werden.

unerwartete Exceptions Eine wichtige Klasse von Exceptions ist die Notwendigkeit die aktuelle Prozesstruktur von einem definierten Workflow während der Laufzeit verändern zu können, um sich neuen unerwarteten Anforderungen anpassen zu können.

erwartete Exceptions ist Versagen eines process work steps auf der Workflowebene. Erwartete Exceptions repräsentieren nicht den normalen Vorgang von Geschäftsabläufen, aber können sich häufig ereignen und deshalb sollten spezielle Mechanismen zu deren Behandlung bereitstehen. Meist ist es das Ziel einen Prozessfortschritt zu erreichen. Ist dies nicht möglich müssen zuvor getätigte Änderungen rückgängig gemacht werden, damit ein konsistenter Zustand erreicht wird und dann die Prozessausführung fortgesetzt werden kann, indem ein alternativer Pfad ausgeführt wird um den Prozess erfolgreich beenden zu können.

application failure sind Versagen durch Programmierfehler oder durch Verletzungen von Einschränkungen (Assertions). Meist wird dies durch unerwartete Eingaben ausgelöst. Der Prozess muss angehalten werden und ein Fachexperte (Mensch)

muss die Versagen behandeln. Der unterbrochene Prozess, in dem sich das Versagen ereignete, muss neugestartet werden.

basic failure ist Versagen auf der Systemebene, wie beispielsweise Netzwerkversagen. Die Behandlung dieser Versagen wird von entsprechenden Komponenten des darunterliegenden Systems übernommen. Typisches Versagen in dem Datenbankbereich sind Deadlocks, Verbindungsprobleme oder Medienversagen.

2.5 Exceptionhandler-Aktionen

Nach Berry [BKvSY80], Cocco et al. [NC82], Goodenough [Goo75] und Yemini [Lac91] können folgende Aktionen von Exceptionhandlern auf eine Exception erfolgen.

setze den Exceptionmelder fort Führe Aktionen aus und setze danach die Operation am **point of raise** fort.

terminiere den Exceptionmelder Führe Aktionen aus, und gebe danach ein Ersatzergebnis von dem geforderten Datentyp zurück. Hat die Operation keinen Rückgabewert, so kann sofort nach dem Konstrukt der dem Aufruf der Operation folgt, gesprungen werden, aber es müssen zur Erfüllung des Ziels der Operation alternative Ressourcen, Algorithmen, etc. benutzt werden.

Wiederhole den Exceptionmelder Führe Aktionen aus und rufe den Exceptionmelder erneut auf.

reiche die Exception weiter Führe Aktionen aus und erlaube es dem Aufrufer des Exceptionsmelders auf die Exception zu reagieren.

Übertrage den Kontrollfluss Führe Aktionen aus und leite den Kontrollfluss danach zu einer anderen Stelle im Programm. Dies beinhaltet das Ausführen von Aktionen und das anschließende Terminieren eines geschlossenen Konstruktes welches den Exceptionmelder enthält.

2.6 Entwurfregeln für den Einsatz von Exceptions

[LS79]: Um zu verstehen wie eine Prozedur implementiert wurde, sollen allein die Spezifikationen der verwendeten Prozeduren genügen, aber nicht deren Implementierungen. Die Spezifikation muss daher alle Exceptions, die signalisiert werden können, aufzählen. Dies

gilt auch für die Exceptions von Prozeduren auf einer tieferen Ebene, sofern dies nicht durch die Verwendung des einschrittigen Weiterreichungsmechanimus unterbunden wird.

Kienzle [Kie08] schlägt vor, Exceptions nicht - wie es oft vorkommt - nur in der Implementierungsphase des Softwareentwicklungszyklus zu verwenden, sondern Exceptions im gesamten Lebenszyklus zu verwenden. Eine Exception ist so ähnlich wie ein Objekt. Beide haben ihren Ursprung auf der Programmiersprachebene. Objekte werden mittlerweile in allen Phasen der Softwareentwicklung verwendet. Ein Objekt ist eindeutig, und wird durch einen Zustand und ein zugeordnetes Verhalten definiert. In der Anforderungsphase repräsentieren Objekte Domänenkonzepte, wie beispielsweise externe Aktoren. Auf der Softwarearchitekturebene repräsentieren Objekte Komponenten, aus denen das System besteht. Zwischen den Objekten der einzelnen Phasen bestehen keine Eins-zu-Einsabbildungen und die Bedeutungen der Objekte ändern sich durch Abbildung auf die nächste Phase. Doch durch Objekte wurde eine Verfolgbarkeit durch alle Phasen hindurch geschaffen.

Romanovsky und Lemos [dLR01], sowie Rubira et al. [RdLFF05] sind der Auffassung, dass durch Integrierung von Exceptions in den gesamten Software-Lebenszyklus ähnliche Vorteile erzielt werden können. Romanovsky und Lemos [dLR01] schildern die Vorgehensweise, dass in jeder Phase des Software-Lebenszyklus eine Klasse von Exceptions definiert wird. In der Anforderungsphase werden anwendungsrelevante Exceptions definiert. In der Entwurfsphase werden entwurfsspezifgische Exceptions definiert. In jeder Phase werden auch entsprechende Handler definiert. Es kann notwendig sein übergreifende Handler zu definieren, die anwendungsrelevante und entwurfsspezifische Exceptions behandeln.

Exceptions in der Anforderungsphase repräsentieren jede potentiell auftretende Situation, in der ein System gehindert werden kann die ihr vertraglich vorgeschriebene Operation zu erfüllen. Nach Kienzle [Kie08] können auf dieser Abstraktionsebene Exceptions in zwei Klassen eingeteilt werden. Die Kontext-betreffenden Exceptions ändern den Kontext in dem das System operiert. Gewisse Kontextänderungen unterbrechen die komplette Erfüllung eines Services oder erfordern die Ausführung außergewöhnlicher Services, die als Exceptionhandler angesehen werden können. Ein Beispiel wäre ein Fahrstuhl, der bei Feueralarm nicht die ausgewählte Etage anfährt, sondern ins Erdgeschoss fährt und die Türen öffnet. Die andere Klasse sind service-betreffende Exceptions, welche die komplette Erfüllung eines bestimmten Services oder Ziels gefährden. Ursachen dafür sind unter anderem, ein Systemzustand, der die Bereitstellung des Services verhindert, ein Versagen eines anderen Services von dem ein Ergebnis erwartet wird, ein Versagen der Kommunikationsleitung, sowie ein Verletzten des Interaktionsprotokolls.

Während der Spezifikationsphase werden die Schnittstellen des Systems beschrieben. Auf dieser Abstraktionsebene liegen die gesendeten Nachrichten zwischen den Schnittstellen im Fokus. Aus diesem Grund ist eine Exception in der Spezifikationsphase eine außergewöhnliche ausgetauschte Nachricht zwischen dem System und seiner Umgebung, oder das Ausbleiben einer normalen Nachricht. Diese Exceptions können in folgenden drei Fällen auftreten. Zum einen wenn eine außergewöhnliche Situation in der Umgebung auftritt, die die Sicherheit oder Zuverlässigkeit von bestimmten Services bedroht. Wird dies erkannt muss der (außergewöhnliche) Aktor das System durch eine geeignete Exception-Nachricht informieren. Der zweite Fall besteht darin, dass das System einen Service eines anderen Aktors verwendet und dieser den Service nicht erbringen kann. Der dritte Fall ist, wenn das System den Service nicht wie vertraglich festgelegt erfüllen kann.

Während der Softwarearchitekturphase ist eine Exception jede außergewöhnliche Ausgabe, die von einer Softwarekomponente produziert wird um einer anderen Komponente anzuzeigen, dass eine außergewöhnliche Situation eingetreten ist.

Während der Implementierungsphase wird eine Exception signalisiert, wenn eine außergewöhnliche Situation eintritt, die die Ausführung von weiteren Anweisungen verhindert.

Nach Yemini et. al [YB85b] sollte ein Aufrufer nicht vor dem Aufruf einer Operation überprüfen, ob die Vorbedingungen für diese erfüllt sind, da dies zu schwerfälligen Code führt. Die Überprüfung wird meisten von den sowieso von der Operation durchgeführt. Einige Überprüfungen können nicht sinnvoll vor dem Prozeduraufruf stattfinden. Programmzuverlässigkeit fordert, dass eine Operation so entworfen wird damit alle Eingabedaten des korrekten Datentyps ohne abzustürzen behandelt werden können, auch wenn die Vorbedingungen nicht efüllt sind.

2.7 Warum Exceptionhandling

[LS79]: Neben dem echten Ergebnisobjektes kann ein Tag mitgeführt werden, welchesden Beendigungsgrund der Prozedur angibt. Dies führt aber dazu, dass nach jedem Auruf ein Bedingungstest folgen muss der bestimmt welcher Beendigungsgrund vorlag. Dies führt dazu, dass solche Programme schwer zu lesen sind, wahrscheinlich auch noch ineffizient sind und folglich es dazu führt, dass Programmierer entmutigt werden solche Verfahren zu benutzen.

2.8 Ursachen von Exceptions

Nach Kienzle [Kie08] sind die Ursachen von Exceptions verschieden je nachdem welche Abstraktionsebene eines Systems betrachtet wird.

2.9 Was sind Fehler?

Nach P.M. Melliar-Smith [MSR77] hängt die Definition eines fehlerhaften Zustandes von der Aufteilung des Algorithmus eines System in normale Algorithmen und abnormale Algorithmen ab, wobei die abnormalen Algorithmen auch als Fehlerbehandlungsalgorithmen bezeichnet werden. Die Aufteilung und somit auch die Bestimmung eines Zustandes als fehlerhaften Zustand ist ein Frage der Beurteilung.

2.10 Erkennung von Fehlern

Nach P.M. Melliar-Smith [MSR77] kann Versagen stattfinden, was darauf schließen lässt, dass ein Fehler existiert. Nach dem Feststellen von Versagen liegen meist noch zu wenig Informationen vor, um eine Informationseinheit als Fehler ausfindig zu machen. Beispielsweise weicht die Sequenz interner Zustand bei einem algorithmischen Fehler von dem korrekten System ab einem gewissen Punkt ab. Die Fehlerursache ist ein algorithmische Fehler, aber durch den prozeduralen Aufruf der Algorithmen und der Annahme dass es keinen einzigartigen korrekten Algorithmus für ein Problem gibt, kann dieser nur subjektiv in einem der Algorithmen bestimmt werden, aber nicht an einer einzelnen Stelle alleine. In vielen System ist die Behandlung von Fehlern notwendig, aber die Fehlerursachen die diese Fehler verursachen zu beheben - obwohl erstrebenswert - ist nicht notwendigerweise erforderlich um mit der Operation fortzufahren.

2.11 Unterschied zu fault-tolerance Programmierung

Nach Knudsen [Knu00] bezeichnet **fault tolerant programming** eine Technik um auf Zustände reagieren zu können, die nicht spezifiziert sind. Dazu zählen Programmierfehler, nichtbehandelte Exceptions, sowie unerwartete Systemzustände. Werden diese 'Fehler' innerhalb einer Abstraktion behandelt oder sind sie Teil der Schnittstelle und

können behandelt werden, so wird dies als Exceptionhandling bezeichnet. Fault-tolerant programming beschäftigt sich mit Fehlerbehandlung in allen verbleibenden Fällen. Dazu zählen schlechtentworfene Systeme, fehlerhafte Zustände im Exceptionhandlingcode und Fehler die außerhalb des Systems oder Frameworks ihren Ursprung haben.

3 Wortschatz

An dieser Stelle wird der Wortschatz eingeführt der durchgängig in der vorliegenden Arbeit verwendet wird. Der Wortschatz stammt zum Teil aus folgenden Quellen: [LS79] [MSR77] [XRR98]

Prozedur
bezeichnet einen Programmtext, entweder in einer Hochsprache oder in Maschinencode

Aufruf / Invocation
ist ein textueller Ausschnitt einer Prozedur, der eine andere Prozedur aufruft

Aufrufer
Eine Invocation kann eine Prozedur aufrufen. Die Prozedur der Invocation wird als Aufrufer bezeichnet.

Handler
Ein Programmtext, der dafür gedacht ist ausgeführt zu werden wenn eine Exception geworfen wurde, wird als Handler bezeichnet.

Aktivierung
Der Aufruf einer Prozedur führt zu deren Aktivierung. Eine Aktivierung kann eine Exception signalisieren.

Signalisieren
steht für die deutsche Bezeichnung des englischen throw, raise bzw. signal. Selten wird in dieser Arbeit dafür auch ”aufgetreten” verwendet. (Anmerkung: Von den drei Begriffen wurde signal ausgewählt, da die deutsche Entsprechung einfach zu finden war, und die Bedeutung dieses Begriffs allgemein gehalten ist. Throw und raise werden meist nur mit dem termination model verbunden. Resume wird für das resumption model bzw. retry für das retrying model verwendet.)

Exceptionmelder
bezeichnet eine Aktivierung die eine Exception signalisiert hat.

Exceptionempfänger
bezeichnet eine Aktivierung die eine Exception durch einen Exceptionhandler bzw. eine catch-clause behandelt.

Versagen
Versagen tritt auf wenn ein System seine Dienstleistung nicht wie spezifiziert erbringt. Versagen ist damit ein Ereignis. Versagen bezeichnet nur das Ereignis ohne Implikation dass das Ereignis als solches erkannt wurde.

fehlerhafter Zustand
Ein interner Zustand eines Systems wird als fehlerhafter Zustand bezeichnet wenn der Zustand sich in Übereinstimmung mit der Spezifikation befindet, aber eine weitere Ausführung des normalen Algorithmuses zu einem Versagen führen würde und nicht als Folgefehler angesehen werden kann.

Fehler / Error
Der Begriff Fehler bezeichnet den Teil des fehlerhaften Zustands, der nicht korrekt ist. Somit ist der Fehler eine Informationseinheit.

Fehlerursache / fault
Die Fehlerursache ist die mechanische oder algorithmische Ursache für einen Fehler.

Exceptionkontext
Ein Exceptionkontext bezeichnet einen Abschnitt in dem Exceptions vom selben Typ auf die gleiche Weise behandelt werden. Diese Exceptionkontexte sind meistens Codeblöcke oder die Rümpfe von Prozeduren.

Atomic-Action
Die Aktivität einer Gruppe von Komponenten oder Objekten bildet eine Atomic-Action wenn keine Interaktionen zwischen dieser Gruppe und dem Rest des Systems für die Zeitdauer der Aktivität bestehen.

4 Exception-Grundverständnis

Als Grundlage für dieses Kapitel dient [Ull, Kapitel 6].

Das bisher übliche Vorgehen beim Entwurf einer Methode war, dass sobald eine Überprüfung (check) ergeben hat, dass die Methode ihre Aufgabe nicht mehr erfüllen kann, diese mittels der return Anweisung beendet wurde. Dem return wird meistens ein spezieller Wert übergeben, der auf die Art des Fehlers hinweist.

Anstelle dieser Überprüfungen, die meistens aus if-Anweisungen und if-Rumpf (Codeblock, der ausgeführt wird, wenn der Ausdruck im if zu true ausgewertet wird) bestehen, werden semantische Einheiten gebildet, die größtenteils den if-Rümpfen entsprechen. Es sind Einheiten, die von dem anderen Code getrennt als zusammengehörig ausgesondert werden. Jede dieser gebildeten Einheiten wird mit einem try-Befehl umschlossen, sodass diese semantische Einheiten, auch als try-Blocks bezeichnet werden. Tritt eine Exception auf, so soll der Kontrollfluss zwingend in einen anderen geeigneten semantischen Codeblock umgeleitet werden, der auf diese Exception reagieren kann. Dieser Codeblock wird durch das catch-Konstrukt kenntlich gemacht, und wird deshalb auch als catch-Block bezeichnet. Der catch-Block behandelt die aufgetretene Exception und wird aus diesem Grund auch als Exceptionhandler bezeichnet. Das beschriebene try-catch Konstrukt ist in der Abbildung 4.1 dargestellt.

```
1 try
2 {
3  // Programmcode, der eine Exception ausloesen kann
4 }
5 catch ( ... )
6 {
7   // Programmcode zum Behandeln der Exception
8 }
```

Bild 4.1: Ein Try-Catch-Block in Java

In den meisten Programmiersprachen gibt es mittlerweile die Funktion, dass nach dem Auftreten oder nicht-Auftreten einer Exception Programmcode ausgeführt wird. Diese Programmzeilen werden folglich immer ausgeführt. Dieser Codeblock wird durch das

finally-Konstrukt kenntlich gemacht und als finally-Block bezeichnet. Ein Beispiel ist in der Abbildung 4.2 zu sehen.

```
1  try
2  {
3    // Programmcode, der eine Exception auslösen kann
4  }
5  catch ( ... )
6  {
7    // Programmcode zum Behandeln der Exception
8  }
9  finally
10 {
11   // Programmcode der bedingungslos ausgeführt wird
12 }
```

Bild 4.2: Ein Try-Catch-Block mit anschließendem finally-Block in Java

Der Sinn des finally-Blocks ist es Datenbankverbindungen, Handler zu Dateien bzw. anderen Diensten zu schließen und Aufräumarbeiten durchführen zu können. Beim Auftreten einer Exception ist neben dem Exceptiontyp und der Exceptionnachricht (eine für Menschen, meist an Programmierer gerichtete textuelle Nachricht), der Stacktrace informativ. Der Stacktrace listet die Folge aller Methodenaufrufe auf. Dem ist zu entnehmen, in welcher Methode die Exception auftrat und welcher Kontrollfluss (in diesem Fall Pfad) zu der Fehlerstelle geführt hat.

In den meisten Programmiersprachen existiert ein allgemeines Objekt (meistens eine Klasse) von der alle anderen Exceptiontypen abgeleitet werden. Zur Kenntlichmachung wird in der gesamten Arbeit die allgemeinste Exceptionklasse als BaseExceptionType bezeichnet. Von dieser werden alle anderen Exceptiontypen abgeleitet. Jede Programmiersprache beinhaltet einen fundamentalen Satz an Exceptiontypen, die sich von dieser Klasse ableiten. Es können aber auch beliebig weitere Exceptiontypen durch Ableiten von dieser Klasse entworfen werden.

Der aufmerksame Leser wird festgestellt haben, dass nach dem Befehlswort catch in den runden Klammern nur Auslassungspunkte zu sehen sind. An deren Stelle muss der Exceptiontyp angegeben werden, von dem Klasseninstanzen aufgefangen und auf die im catch-Block reagiert werden soll. Auf den Exceptiontyp folgt noch ein Variablenbezeichner. Durch diesen ist ein Zugriff auf die Exceptioninstanz möglich. Ist die Exception eine abgeleitete Klasse von BaseExceptionType so sind Methoden vorhanden, um den Typ der Exception, die Exceptionnachricht und den Stacktrace abzufragen. Ein try-Block kann mehr als einen Exceptiontyp werfen. Demnach können auf einen try-Block mehrere

catch-Blöcke stehen, die auf unterschiedliche Exceptiontypen reagieren können. Für jede aufzutretene Exception innerhalb eines try-Blocks werden ebenso viele catch-Blöcke benötigt. Wenn die Reaktion auf verschiedene Exceptiontypen dieselbe ist, so kann die Oberklasse der beiden Exceptiontypen gewählt werden, und somit alle abgeleiteten Exceptiontypen von dieser Klasse mit einem einzigen catch-Block behandelt werden.

Wird nun eine Exception geworfen, dann wird ein Matching durchgeführt, das heißt es wird ein catch-Block gesucht der zu dem Exceptiontyp passt. Dafür muss es sich entweder um genau den bezeichneten Exceptiontyp oder um eine von ihm abgeleitete Exceptionklasse handelt.Das matching wird beginnend mit dem ersten catch-Block nach dem try-Block durchgeführt, bis entweder ein passender catch-Block gefunden, oder das Ende erreicht wurde. So wird maximal nur ein catch-Block ausgeführt. Wird kein passender catch-Block gefunden, dann wird die Exception im Stacktrace an die aufrufende Methode weitergereicht. Hat diese auch keinen passenden catch-Block, so geht dies immer weiter, bis der Stacktrace leer ist, und der letzte Aufrufende das Betriebssystem ist. Dies führt zu einem Absturz des Programms, meist mit einer generischen Fehlermeldung und dem Anzeigen des ursprünglichen Stacktraces.

Der matching Algorithmus kann wie folgend ausgenutzt werden. Ist die Exceptionhierachie wie in 4.3 gegeben, in der von der BaseExceptionType der neue Exceptiontyp ExceptionABC erstellt wurde, und von diesem drei direkte Klassen abgeleitet wurden: ExceptionA, ExceptionB und ExceptionC.

```
BaseExceptionType
|
+-- ExceptionABC
       |
      +--ExceptionA
      +--ExceptionB
      +--ExceptionC
```

Bild 4.3: Beispielhafte benutzerdefinierte Exceptionhierachie

Wenn auf alle drei Exceptions gleich reagiert werden soll, reicht ein ”catch(ExceptionABC)” aus. Soll aber auf ExceptionB anders reagiert werden, so müsste nun für jeden Exceptiontyp ein eigener catch-Block erstellt werden, in dem für ExceptionA und ExceptionC derselbe redundante Code verwendet werden würde, wie in 4.4. Nun kann die Abarbeitungsreihenfolge des matching Algorithmus verwendet werden und wie in der Abbildung 4.5 zuerst der Spezialfall ”catch(ExceptionB)” abgefangen

werden, wenn dieser Eintrag nicht passt, nur dann würde der nächste allgemeine Eintrag "catch(ExceptionABC)" aufgerufen werden.

```
1  try
2  {
3    // Programmcode, der eine Exception vom Typ ExceptionABC auslösen
       kann
4  }
5  catch ( ExceptionA ex)
6  {
7    // Programmcode zum Behandeln der Exception
8  }
9  catch ( ExceptionB ex)
10 {
11   // Programmcode zum Behandeln der Exception
12 }
13 catch ( ExceptionC ex)
14 {
15   // Programmcode zum Behandeln der Exception
16 }
```

Bild 4.4: Catch-Blöcke für jede auftretbare Exception

```
1  try
2  {
3    // Programmcode, der eine Exception vom Typ ExceptionABC auslösen
       kann
4  }
5  catch ( ExceptionB ex)
6  {
7    // Programmcode zum Behandeln der Exception
8  }
9  catch ( ExceptionABC ex)
10 {
11   // Programmcode zum Behandeln der Exception
12 }
```

Bild 4.5: Catch-Blöcke für den Spezialfall ExceptionB und danach alle weiteren Exception mit der Oberklasse abfangen

5 Klassifizierung eines Exception Handling Mechanismus

Um den Leistungsumfang moderner Programmiersprachen in lokalen Systemen bewerten zu können, muss zunächst ein Referenzmodell erstellt werden. In diesem Abschnitt wird eine Klassifikation für lokale Systeme entworfen. Diese teilt Exceptionhandling in mehrere Phasen ein. Die erste Phase ist das Erkennen einer **exceptional condition** (außergewöhnliche Ausnahme) und das Anzeigen dieser durch **Signalisierung dieser Exception**. Die zweite Phase ist das Weiterreichen (auch Propagating) der Exception bis ein Exception-Handler gefunden wird. Die letzte Phase besteht in der Bestimmung des geeignetesten Exceptionhandlers und dessen Ausführung, d.h. die eigentliche Behandlung der Exception.

Als Grundlage für dieses Kapitel dient die Klassifikation von Buhr et al. [BM00].

5.1 Eigenschaften eines Exception Handling Mechanismus

Ein Exception Handling Mechanismus sollte folgende Eigenschaften besitzen:

- Softwaretests werden unterstützt und können einfach durchgeführt werden
- Der Kontrollfluss kann explizit geändert werden
- Nicht vollständig abgeschlossene Operationen werden nicht fortgeführt
- Erweiterungen sind mit minimalem Aufwand und Änderungen verbunden

Die ersten beide Punkte dienen dabei der Lesbarkeit und der Programmierbarkeit.

5.2 Übersicht

Ein Exception Handling Mechanismus beinhaltet das Werfen (raise bzw. throw), Weitereichen (propagating) und Behandeln (catch) von Exceptions.

5.3 Signalisieren von Exceptions

Durch das Signalisieren einer Exception wird ein außergewöhnlicher Zustand angezeigt. Dieser soll nicht durch den gewöhnlichen Kontrollfluss behandelt werden, deshalb wird nach dem Signalisieren einer Exception der Kontrollfluss geändert. Dies muss aber nicht in derselben execution stattfinden.

Es können zwei Arten von executions unterschieden werden. Zum einen die **source execution**, in der die Exceptioninstanz signalisiert wird, und zum anderen die **faulting execution**, die aufgrund der signalisierten Exceptioninstanz den Kontrollfluss ändert.

Demnach kann das Signalisieren von Exceptions synchron oder asynchron erfolgen. Synchrone Exceptions ändern den Kontrollfluss in der selben execution, in der sie signalisiert wurden. Source execution und faulting execution fallen zusammen.

Asynchrone Exceptions werden in einer source execution signalisiert, aber in der faulting execution wird der Kontrollfluss geändert. In der faulting execution wird der Kontrollfluss in einen Handler geleitet und somit findet hier das Weitereichen von Exceptions statt.

5.4 Weiterreichen von Exceptions

Die Weiterreichung von Exceptions kann durch drei Entscheidungen bestimmt werden. Sollen Exceptions weitergereicht werden, so muss entschieden werden, wo der Kontrollfluss fortgeführt werden soll, nachdem die Exception behandelt worden ist. Dies wird durch den Weiterreichungsmechanismus entschieden. Als nächstes ist noch festzulegen, wie der Pfad gebildet werden soll, entlang dessen die Exception weitergereicht wird, bis eine passende **handler clause** gefunden wird. Diesen gibt das Weiterreichungsmodell vor. Zuletzt folgt die Auswahl des geeignetesten Exceptionhandler aus der gefundenen handler clause.

5.4.1 Weiterreichungsmechanismus / Kontrollfluss in der Quelle

Es gibt drei Lösungen um einen Weiterreichungsmechanismus umzusetzen: Werfen, Wiederaufnehmen oder ein Mechanismus, indem Werfen und Wiederaufnahme möglich ist. Der Weiterreichungsmechanismus bestimmt in der Quelle welchen Weg der Kontrollfluss nach Auftreten einer Exception in dieser geht.

5.4.1.1 Werfen / throw

Bei **Werfen** wird nach der Behandlung der Exception nicht zum **point of raise** zurückgesprungen. Auf der Suche nach dem Handler wird der Stack abgearbeitet. Dabei werden die einzelnen stack frames entfernt (**stack unwinding**) bis ein passender Handler gefunden worden ist. Dies entspricht dem Zerstören der Codeblöcke die zwischen dem Werfen und dem **Exception-Handler** liegen. Dieses stack unwinding kann stattfinden, wenn der Handler gefunden wurde, während der Ausführung des Handler oder nach der Ausführung des Handler.

5.4.1.2 Wiederaufnehmen / resume

Bei **Wiederaufnehmen** wird mit der Ausführung am **point of raise** fortgefahren. Sollte eine **Wiederaufname** nicht möglich sein, so kann eine neue Exception geworfen werden, welche die **Werfen**-Semantik trägt. Dies erfolgt beispielsweise in VMS mittels des **unwind**-Befehls.

Die Idee liegt nahe nur das **Wiederaufnehmen** Konzept zu verwenden und **Werfen** durch einen Aufruf von dem unwind-Befehl zu bewerkstelligen. Mit diesem Ansatz wird die Fähigkeit verloren in der Quelle zu spezifizieren, ob **Werfen** oder **Wiederaufnehmen** durchgesetzt werden soll. Dadurch könnten unsichere **Wiederaufnahmen** auftreten.

5.4.1.3 Verwendung

Welcher Mechanismus verwendet wird kann entweder explizit durch ein Befehlswort beim Signalisieren der Exception (**quellenbestimmte Weitereichungsmechanismus-Auswahl**), oder bei der Deklaration der Exception (**exceptionbestimmte Weitereichungsmechanismus-Auswahl**) angegeben werden.

Die Angabe mittels Befehlswort wird von keinem in der Literatur bekannten System unterstützt. Durch die Angabe des zu verwendenden Mechanismus bei der Deklaration der Exception findet eine Partitionierung der Exceptions statt. µSystem und Exceptional C sind Vertreter für diese Variante.

5.4.1.4 Randbedingungen

5.4.1.4.1 einschritter oder mehrschrittiger Weiterreichungsmechanismus

Liskov und Snyder [LS79] treten für einen **einschrittigen** im Gegensatz zu einem **mehrschrittigen Weiterreichungsmechanismus** ein. Ihrer Auffassung nach realisiert eine

Prozedur eine Abbildung von einer Eingabemenge in eine Ausgabemenge. Der Aufrufer muss nur wissen, welche Abbildung durchgeführt wird, aber nicht wie (Implementierung) diese durchgeführt wird. Demnach ist es angemessen, dass der Aufrufer die Exceptions kennt, die bei Aufruf der Prozedur laut Schnittstellenbeschreibung auftreten können.

Er sollte aber nichts über die Exceptions wissen müssen, die von den Prozeduren, die zur Implementierung der aufgerufenen Prozedur dienen, stammen, da dies zu den Implementierungsdetails der Prozedur gehört. Die Lösung um die Behandlung einer Exception weiterzureichen besteht darin, dass die Prozedur eine eigene Exception wirft, die Teil ihrer Schnittstellenbeschreibung ist.

5.4.1.4.2 Automatische oder explizite Weiterreichung

Nach Yemini et al. [YB85b] verletzt die **automatische Weiterreichung** von unbehandelten Exceptions entlang der Aufruferkette das Prinzip des **information hiding**, wenn Exceptions über mehre Abstraktionsebenen weitergereicht werden. Einige Exceptions können Details der Implementierung des Exceptionsmelders preisgeben, wenn diese nicht zur Weiterreichung außerhalb des Moduls gedacht sind, aber weitergereicht werden, da sie nicht im Modul durch einen Exceptionhandler behandelt wurden.

Auch wenn nicht jeder Prozeduraufruf eine Änderung der Abstraktionsebene darstellt, so kann eine sichtbare Funktion f innerhalb eines Moduls mehrere versteckte Funktionen in beliebiger Aufruftiefe aufrufen. Wenn der Aufrufer von f eine Exception von einer der versteckten Funktionen erhalten würde, die nicht in eine abstraktions-relevante Form von f übersetzt wurde, so kann kein passender Exceptionhandler gefunden werden. Der sauberste, einheitlichste Weg zu garantieren, dass Exceptions richtig übersetzt werden ist es, dass automatische Weiterreichung verboten wird.

Die explizite Weiterreichung wird von Liskov und Snyder [LS79], Goodenough [Goo75], Yemini et al. [YB85b], Levin [Lev77] und Cocco et al. [NC82] befürwortet. Automatische Weiterreichung hätte die Folge, dass der Exceptiontyp auch außerhalb des Bereiches verwendet werden kann, in dem er sichtbar ist. Das Deaktivieren von automatischer Weiterreichung erlaubt statische Überprüfungen, die folgende Vorteile hat:

- Jede Exception wird mit der korrekten Menge von Parametern signalisiert
- Jeder Exceptionhandler wird mit der korrekten Menge von formalen Parametern definiert
- Nur Exceptions, die von dem Exceptionmelder definiert wurden, können signalisiert werden (dies ist die Durchsetzung des Prinzips der expliziten Weiterreichung)

- Überprüfen, ob alle Exceptions die in einem Bereich signalisiert werden können, auch in dem Bereich behandelt werden

5.4.2 Weiterreichungsmodell / Exceptionhandler-Suchpfad

Das Weiterreichungsmodell legt fest auf welche Weise nach einem Handler gesucht wird. Genauer gesagt wird durch das Modell die Reihenfolge festgelegt nach der die **handler clauses** durchsucht werden. Es entscheidet aber nicht welcher Handler in einer handler clause ausgewählt wird.

Es gibt zwei verschiedene Ansätze für ein Weiterreichungsmodell. Die **dynamische Weiterreichung** und die **statische Weiterreichung**.

5.4.2.1 Dynamische Weiterreichung

Der call stack wird von oben nach unten durchsucht. Oben auf liegt der Stackframe mit dem engsten Kontext zu der geworfenen Exception. Jeder darunterliegende Stackframe ist mit einem mehr und mehr allgemeinerwerdenden Kontext verbunden. Das Behandeln von Exceptions ist laut Buhr et al. [BM00] in engerem Kontext am einfachsten. Dieser Ansatz minimiert die Anzahl der Stack Unwindings falls die Exception dynamisch geworfen wird. Zu den Nachteilen dieses Ansatzes zählt aber das **Sichtbarkeitsproblem**, die **dynamische Handlerauswahl** und das **rekursive Fortsetzen**.

- Das **Sichtbarkeitsproblem** beschreibt den Umstand, dass eine geworfene Exception durch Methodenaufrufe hinweg, in denen sie nicht definiert ist (z.B. in der Signatur steht), weitergereicht werden kann, bis zu einem Methodenaufruf in der diese wieder definiert ist (durch Aufnahme in der Signatur oder in einem Exceptionhandler) und in der sich auch der entsprechende Handler befindet. Allgemein gesprochen kann eine geworfene Exception durch verschiedene lexikalische Bereiche weitergereicht werden, in denen sie unsichtbar ist. Dies ist nach Buhr et al. [BM00] unerwünscht, da dadurch eine Routine indirekt Exception weiterreicht, die sie nicht kennt. Einige Entwickler von Programmiersprachen sind der Auffassung, dass Exceptions niemals in einen Bereich weitergereicht werden sollen, in denen sie unsichtbar sind. Falls dies dennoch erlaubt wird, so sollte die Exception in eine allgemeine failure Exception konvertiert werden.
- Die **dynamische Handlerauswahl** hat zur Folge, dass nicht statisch vor Ausführung des Programms bestimmt werden kann, welcher Handler bei Eintreten

einer bestimmten exceptional condition aufgerufen wird. Somit ist es schwierig den Ansatz zu benutzen und den Kontrollfluss zurückzuverfolgen.

- Das **rekursive Fortsetzen** bereitet Probleme, da kein Stack Unwinding stattfindet, und somit alle Handler sich im selben Kontext befinden. Bei Aufruf eines resume Handler sind alle anderen Resume Handler immer noch aktiv und es kann zu zyklischen Handleraufrufen kommen. Die einfachsten Fälle sind dabei, dass sich der selbe Handler rekursiv aufruft, oder dass sich zwei Handler gegenseitig endlos aufrufen.

  Dies ist schwer zu entdecken, da die Entscheidung des Handlers dynamisch erfolgt. Weitere Schwierigkeiten, kommen hinzu falls asynchrones Fortsetzen unterstützt werden soll.

  In Mesa [MMS78] existiert eine Lösung, die aber als erfolglos betrachtet wird, da sie nicht nachvollziehbar arbeitet.

5.4.2.2 statische Weiterreichung - impliziert Werfen / throw

Die statische Weiterreichung ist ein Vorschlag von Knudsen [Knu00] [Knu87] der die Probleme der dynamischen Weiterreichung eliminieren soll. Er wurde aber in der Literatur zum größten Teil ignoriert. Der Ansatz durchsucht lexikalische Hierachien und basiert auf dem sequel Konstrukt von Tennent [Ten77].

Eine sequel ist eine Routine einschließlich Parametern, die nach ihrer Ausführung nicht den Kontrollfluss an die Stelle nach ihrem Aufruf weiterleitet, sondern zu dem Ende des Codeblocks, in dem sie aufgerufen worden ist. Dies entspricht dem termination model mit einem Weiterreichen der Exceptions entlang einer Hierachie und somit einer statischen Weiterreichung.

Virtuelle oder Default-Sequels [BM00] können für Aufräumarbeiten verwendet werden.

Dieser Ansatz eignet sich sehr gut für monolithische Programme, aber sobald der statische Kontext von Modulen bzw. Bibliotheken und Benutzercode besteht, ist dieser Ansatz nicht mehr anwendbar.

5.4.3 Exceptionhandler-Auswahl

Die Exceptionhandler-Auswahl bezeichnet das Verfahren, wie der beste Exceptionhandler aus einer **handler clause** ausgewählt wird. Eine **handler clause** kann mehrere Exceptionhandler beinhalten die eine signalisierte Exception behandeln können. Die Auswahl des Exceptionhandlers kann mittels der Kriterien **Übereinstimmung**, **Kontextnähe**,

Spezifizität und **Handlerposition** getroffen werden. Die Priorität der Kriterien sollte auch in dieser Reihenfolge Anwendung finden.

- **Übereinstimmung** wird nur benötigt wenn der Exception-Handling-Mechanismus verschiedene Weitereichungsmechanismen 5.4.1 und Partitionierung der Exceptionhandler verwendet. Übereinstimmung sorgt dafür, dass nur ein Handler der mit dem gewählten Weitereichnusmechanismus übereinstimmt ausgewählt wird.
- **Kontextnähe** wählt den Exceptionhandler aus, der auf dem Stack von dem **point of raise** am kürzesten entfernt und in der Lage ist die Exception zu behandeln.
- **Spezifizität** beurteilt unter einer Menge von geeigneten Exceptionhandlern, welcher von ihnen spezifischer ist. Ein Vergleich von zwei Handlern ergibt folgende Regeln. Behandeln beide **denselben Exceptiontyp** so ist derjenige, der Bedinungszweige verwendet spezifischer. Verwenden beide **(nicht) Bedingungszweige** so ist der Handler der den konkreten Exceptiontyp behandelt spezifischer, als derjenige der nur eine Oberklasse des Exceptiontyp behandelt.

  Jedoch wird eine Aussage schwierig, wenn ein Handler den konkreten Exceptiontyp, und der andere eine Oberklasse des Exceptiontyps dafür aber Bedingungszweige verwendet. In diesem Fall wären beide Handler gleich spezifisch.
- **Handlerposition** in einer **handler clause** wird als letztes Kriterium herangezogen, wenn beispielsweise gleiche Spezifizität besteht. Nach diesem Kriterium wird der zuerst aufgeführte, passende Exceptionhandler in einer handler clause ausgewählt.

5.5 Behandeln von Exceptions - Strategie der Korrektur

Die letzte Schritt eines Exceptionhandlings-Mechanismus ist die Ausführung des Exceptionhandlers. Die Ausführung, d.h. die eigentliche Behandlung von Exceptions kann nach der **Behandlungsart**, dem **Behandlungskontext** und der **Signalisierungs-Synchronität** unterschieden werden.

Weil die **Behandlungsart** von dem Weg des Kontrollflusses, d.h. dem Kontrollflussmechanismus, bestimmt wird, und der Weg des Kontrollflusses leichter als die Behandlungsart zu differenzieren ist, findet die Unterscheidung der Behandlungsart indirekt über die Unterscheidung des Kontrollflussmechanismus statt.

5.5.1 Exception-Kontrollflussmechanismen

Nach Buhr et al. [BM00] gibt es vier verschiedene Modelle, die bestimmen wie mit einer Exception vom Werfen bis zur Behandlung umgegangen wird. Der hauptsächliche Unterschied zwischen den Modellen besteht darin wohin der Kontrollfluss geleitet wird. Und dies bestimmt welche Art von Korrektur (**Behandlungsart**) der Handler für diesen Fall durchführen muss.

5.5.1.1 Non-local transfer model

Der Kontrollfluss wird durch das Anspringen von labels geändert. Diese Sprungmarken sind nicht statisch, denn ein label besteht aus einem transfer point und einem Zeiger auf ein activation record auf dem Stack. Die Technik wird mittels den C-Befehlen setjmp und longjmp realisiert.

Dieses Modell ist ungeeignet, da ein beliebiges Springen im Programmcode erlaubt ist und es so sehr fehleranfällig ist. Außerdem ist keine Compilerunterstützung gegeben. Der Compiler erkennt nicht, dass es sich bei diesem Pattern um Exceptions handelt und es können ungültige Zustände durch Optimierungen eintreten.

5.5.1.2 Termination model

Die Strategie, die dem termination model zu Grunde liegt, besteht darin, Exceptions und damit den fehlerhaften Zustand zu behandeln, indem ein konsistenter Zustand hergestellt wird und der Codeblock der seinen Auftrag aufgrund des fehlerhaften Zustandes nicht erfüllen konnte, durch Ausführung eines alternativen Blocks, der den Auftrag erfüllen kann, übersprungen wird. Der Kontrollfluss wird vom guarded block in den Handler geleitet. Nach der Ausführung des Handlers wird der Kontrollfluss nach dem guarded block fortgesetzt. Somit verhält sich das termination model wie eine alternative Operation zu der im guarded block Festgelegten. Bevor eine Exception signalisiert wird, müssen nach Liskov [LS79] von dem Exceptionmelder stets Aufräumarbeiten durchgeführt werden. Das Signalisieren einer Exception beendet den Exceptionmelder.

5.5.1.3 Retrying model

Die retrying model Strategie ist es, einen konsistenten Zustand herzustellen, und die fehlgeschlagene Operation erneut auszuführen.

Der Kontrollfluss wird vom guarded block in den Handler geleitet. Nach der Ausführung des Handlers wird der Kontrollfluss erneut bei dem restart point fortgesetzt. Der restart

point ist sinnigerweise der Anfang des guarded blocks. Aus Sicht der Implementierung bietet sich auch kein anderer Punkt an. Dieses Model findet in Mesa [MMS78], Exceptional C [Geh92b] und Eiffel [Mey92] Verwendung.

In diesem Modell schleichen sich bei falscher Verwendung leicht subtile Fehler ein. Bei der Verwendung von mehr als einem Handler in einem guarded block muss der selbe restart point gewählt werden. Dies schränkt die Flexibilität enorm ein, da nur noch Handler dem guarded block zugeordnet werden können, die denselben restart point haben. Hohe Flexibilität wäre gegeben, wenn jeder Handler einen eigenen restart point wählen könnte. Dieses Modell kann wie Gehani [Geh92b] gezeigt hat leicht durch eine Schleife und dem termination model nachgeahmt werden.

5.5.1.4 Resumption model

Die resumption model Strategie ist es einen konsistenten Zustand herzustellen, Korrekturmaßnahmen vorzunehmen, und an die Stelle im Codeblock zurückzuspringen, welche die Exception signalisiert hat und dort die Ausführung fortsetzen.

Der Kontrollfluss wird vom Punkt, an dem die Exception geworfen wurde, in den Handler geleitet. Nach der Ausführung des Handlers wird der Kontrollfluss an dem Punkt an dem die Exception geworfen wurde fortgesetzt. Semantisch ist dies identisch zu einem Funktionsaufruf. Im Gegensatz zu einem Funktionsaufruf wird der Handler aber dynamisch bestimmt. Dies könnte nachgeahmt werden, indem einem Funktionsaufruf als Argumente Funktionen übergeben werden.

Die Nachteile an dem Modell sind, dass veralteter Code komplett umgeschrieben werden muss, eine Wiederverwendung kaum möglich ist und dass normale Ausführung und Exceptionhandling nicht voneinander getrennt sind.

Nach Liskov und Snyder [LS79] wird durch das resumption model eine größere Abhängigkeit erzeugt. Normalerweise ist die aufrufende Prozedur von der aufgerufenen Prozedur abhängig, da die aufgerufene Prozedur eine Abbildung durchführen soll. Im resumption model ist zusätzlich die aufgerufene Prozedur (kann Exception auslösen) von der aufrufenden Prozedur (besitzt Handler) abhängig, wenn eine Exception auftritt. Die Spezifikation einer Prozedur enthält je einen Abschnitt für den normalen Fall und für jeden Exception-Fall. Dies gilt nicht nur für das termination, sondern auch für das resumption model, da nicht immer alle Korrekturmaßnahmen durchgeführt werden können, und notfalls doch ein Abbruch erfolgen muss. Durch die gegenseitige Abhängigkeit zwischen den Prozeduren müssen diese neben den gerade erwähnten termination states

auch aufgezählt werden. Es ist erforderlich das Verhalten, das von den Handlern erwartet wird wenn Exceptions signalisiert werden zu dokumentieren.

[LS79]: Drei Arten von Signalen werden unterschieden, die Fällen zugeordnet werden, in denen der Exceptionmelder nicht fortgesetzt wird, fortgesetzt werden muss oder in denen Fortsetzung optional ist. Im Fall dass kein Handler im Aufrufer ausgeführt wird, wird ein Mechanismus zur Verfügung gestellt, der es dem Exceptionmelder erlaubt "Äufräumarbeiten" durchzuführen (z.B. nichtlokale Variablen in einen konsistenten Zustand überzuführen). Zusätzlich exisitiert ein Standardmechanismus, der es dem Exceptionmelder erlaubt eigene Exceptions zu behandeln, falls das der Aufrufer nicht erledigt.

Es gibt noch weitere Lösungen zu dem resumption model. Das in Mesa verwirklichte basiert auf den Thesen von Goodenough [Goo75] und ist komplexer. Eine andere Lösung [Geh92b] [BMZ92] sind dynamische Funktionsaufrufe, die auf verschachtelten Routinen beruhen. Ein ganz einfaches Modell ist auch ohne verschachtelte Routinen implementierbar.

Das **resumption model** leidet darunter, dass kein **stack unwinding** stattfindet und so alle Kontexte und damit auch die **Exception-Handler** erhalten bleiben. So kann es vorkommen, dass in einem resume-Handler erneut eine Exception vom selben Typ geworfen wird und derselbe Handler zur Behandlung dieser Exception erneut aufgerufen wird. Auch mit abwechselnden Aufrufen zweier oder mehrerer Funktionen ist eine solche Rekursion denkbar. Um eine solche potentielle unendliche Aufrufreihe zu unterbinden, muss das resumption model mit einer entsprechenden Lösung ausgestattet werden.

Im weiteren wird der Lösungsansatz von Mesa und von Peter A. Buhr und W.Y. Russell Mok [BM00] vorgestellt.

- **Mesa resumption model** Dieser Ansatz basiert darauf, dass alle aufgerufenen Exception-Handler markiert werden. Markierte Exception-Handler können nicht mehr benutzt werden und so wird eine Rekursion vermieden.

  Bei diesem Konzept unterliegen Programmierer oft einer von zwei fälschlichen Ansichten. Zum einen der Annahme, dass ein Exception-Handler existiert, obwohl dem nicht so ist. Zum anderen die Annahme, dass eine Rekursion möglich sei.

  Ein Nachteil an dieser Lösung ist, dass wenn im selben lexikalischen Bereich eine Exception vom selben Typ geworfen wird, unterschiedliche Exception-Handler ausgeführt werden, wenn die **Wiederaufnahme** im **guarded block** oder im Exception-Handler erfolgt. Im Fall wenn eine Exception im **guarded block** signalisiert wird, wird ein Exception-Handler der nach diesem Codeblock folgt aufgerufen.

Im Fall der Signalisierung einer Exception in einem Exception-Handler wird ein Exception-Handler, der vor dem Codeblock des davor aufgerufenen Exception-Handlers steht, aufgerufen.

- **Resumtpion model nach Buhr** Dieses Modell führt erstmal den neuen Bergriff **consequent event** ein, der entweder eine synchron signalisierte Exception oder eine andere signalisierte Exception bezeichnet, die als Folge ersterer signalisiert wurde. Damit ist eine asynchron signalisierte Exception in einer source execution kein consequent event in der faulting execution, da die außergewöhnliche Situation in der source execution auftrat.

  In diesem Ansatz werden nicht nur alle aufgerufenen Exception-Handler markiert, sondern alle Exception-Handler die nach der Suche nach einem geeigneten Exception-Handler untersucht wurden, ob sie die notwendigen Kriterien erfüllen. Dies entspricht semantisch dem unwinding der Exception-Handler im termination model.

- **Restart als resumption model** Nach Christophe Dony [CDUV05] werden in der Lisp Terminologie eine Liste möglicher **resumption points** als **restarts** bezeichnet. Restarts bieten lexikalische Reperatur und Fortsetzungsmöglichkeiten an. Restarts decken einen Bereich von dem **point of raise** bis zu dem kontextnähesten Exception-Handler ab, das sie verändern können und einen nicht-lokalen Sprung in diese Umgebung ausführen können. Restarts haben auch dynamische Erweiterung und **gobal scope**. Nach Dony bietet nur Common Lisp und Dylan restarts an, die eine Auswahl von resumption points zulassen.

5.5.1.5 Vergleich der Modelle

Liskov und Snyder [LS79] bevorzugen, das **termination model** gegenüber dem **resumption model**, da das termination model einfacher zu handhaben ist. Stroustrup [Str94] bestätigt auch, dass es einfacher ist das termination models als das resumption model zu implementieren. Beide Modelle bieten die gleiche Ausdruckskraft und die Fälle sind gering, die mittels des termination model schwierig und mittels des resumption model leicht zu modellieren sind.

Nach Miller et. al [MT97] wird das termination model bevorzugt, da es sicherer ist das Objekt, das eine Exception signalisiert hat, zu zerstören und die Berechnung erneut zu starten, als den Versuch zu unternehmen das Objekt in einen konsistenten Zustand zu überführen. Es ist kaum möglich den Systemzustand für alle Situationen hervorzusehen,

in denen der Exceptionhandler aufgerufen werden kann. Bevor mehr Probleme durch inkorrekte Annahmen provoziert werden sollten Aufräumarbeiten und ein bekannter Systemzustand hergestellt werden.

Nach Lacourte [Lac91] wird die Korrektheit einer Schnittstelle nach Aufruf eines Exceptionhandlers außerhalb der gekapselten Implementierung verletzt. Die Argumentation ist Folgende: wird eine Exception signalisiert, so werden die geschachtelten Aufrufer gespeichert. Auf einer beliebigen Ebene des Stacks kann ein Wert geändert und die Ausführung auf einer anderen Ebene des Stacks fortgesetzt werden. Dies impliziert, dass der Teil im gespeicherten Stack, zwischen der Ebene - in der der Wert geändert wurde, und die Ebene - in der die Ausführung fortgesetzt wurde, valide ist. Dies setzt eine vollständige Analyse des betroffenen Codes voraus. Der Aufruf des Exceptionhandlers muss daher aus der Sicht gesehen werden, dass dieser in den verschachtelten Prozeduraufrufen ausgeführt wird. Das Ziel von Kapselung ist es, formal oder informal die Korrektheit einer Implementierung von einer Schnittstelle zu beweisen. Die Korrektheit einer Schnittstelle darf nicht von der Implementierung der Objekte abhängen, die diese benutzen. In diesem Fall, müsste die signalisierende Methode geprüft werden, indem die Schnittstelle des Handlers benutzt wird und der Exceptionhandler als Argument der Schnittstelle übergeben wird. In einer objekt-orientierten Umgebung ist eine Prozedur immer mit einem Objekt verknüpft. Ein resumption Handler muss deshalb ein **closure** sein, der eine Prozedur und eine Umgebung beinhaltet. Dies ist notwendig, damit die aufgerufene Methode, die Schnittstelle für die erwartete **closure** (Exceptionhandler) anbieten kann, um die Validität der vorgeschlagenen Handler zu garantieren.

Nach Lacourte [Lac91] wird das Konstrukt **closure**, das in den Lisp-basierten Programmiersprachen vorkommt, benötigt um resumption handlers verwenden zu können, da sich damit Codeblöcke darstellen lassen, die eine Umgebung beinhalten.

5.5.2 Asynchrone Exceptions

Bisher wurden Exceptions unter der Annahme betrachtet, dass der gesamte Programmcode nur von einem Thread ausgeführt wird. Werden Exceptions signalisiert, so unterbrechen sie den laufenden Kontrollfluss und das Weiterreichen von Exceptions wird angestoßen. Laufen aber mehrere Threads auf einem System oder in verteilten Systemen, so kann ein Thread eine **exceptional condition** erkennen, von der auch ein anderer Thread betroffen sein kann. In diesem Fall muss der andere Thread durch eine Exception darauf aufmerksam gemacht werden. Da diese Signalisierung der Exception von außen (außerhalb

des eigenen Kontrollflusses des Threads) kommt, kann diese Signalisierung nur asynchron stattfinden. Es handelt sich um asynchrone Exceptions.

5.5.2.1 Verschiedene executions

Im Fall von verschiedenenen Executions muss festgelegt sein welche Execution für die Weiterreichung einer Exception verantwortlich ist. In einer **coroutine-Umgebung** ist dafür die faulting execution zuständig und sorgt dafür, dass es zu keinen Missverständnissen kommen kann.

In einer **concurrent-Umgebung** ist es die bessere Lösung wenn auch hier die faulting execution für die Weiterreichung der Exceptions zuständig ist. Wäre in dieser Umgebung die source execution verantwortlich, so müsste diese die faulting execution ändern und den runtime stack sowie den Programmzähler ändern. Die letzten beiden Ressourcen müssen dazu noch gemeinsam verteilte Ressourcen sein und so wäre in diesem Fall noch ein Sperren dieser durch einen Lock nötig. Um solche Leistungseinbußen zu vermeiden wird die erste Variante bevorzugt. [BM00]

Nach Romanovsky et al. [XRR98] kann in einer **concurrent-Umgebung** eine zusätzliche Dimension des **Weiterreichungsmodells** erforderlich sein. In dem Fall von verschachtelten **Atomic-Action**s wird es nötig, dass eine Exception zur nächsthöheren Atomic-Action weitergereicht wird. In dieser Atomic-Action können mehr **Komponenten** beteiligt sein als in der verschachtelten, und in diesem Fall muss die Exception zu diesen Komponenten weitergereicht werden.

5.5.2.2 Kommunikation

Asynchrone Exceptions erfordern eine Kommunikation, die ohne gemeinsame Ressourcen auskommt. Auf der Seite der source execution steht **blockierende** und **nicht-blockierende** Kommunikation zur Verfügung. Die blockierende-Variante leidet darunter dass der Empfänger möglicherweise niemals oder zu selten prüft, ob eine asynchrone Exception signalisiert wurde. Die nicht-blockierende Variante hat den Vorteil, dass keine Verzögerung vorhanden ist. Sie ist daher die ausgewählte Option.

Auf der Seite der faulting execution kann kein expliziter Empfangsbefehl stehen wie in der normalen nicht-blockierenden Kommunikation, da dadurch ein außergewöhnlicher Kontrollfluss signalisiert wird und dieser nicht im normalen Kontrollfluss stehen sollte. Aus diesem Grund muss ein Handler für asynchrone Exceptions installiert werden. Diese Vorgehensweise sorgt für Transparenz gegenüber dem Programmierer, erfordert aber

die Verwendung von polling, um die Exception nicht sofort aber zeitnah in der faulting execution signalisieren zu können.

Das **explicit polling** erfordert expliziten Code. Dies gibt dem Programmierer die Möglichkeit zu bestimmen wann asynchrone Exceptions signalisiert werden können, hat aber zugleich den Nachteil, dass Exceptions auch ignoriert oder verzögert werden können. Dahingegen geschieht das **implicit polling** automatisch und stellt so eine einfachere Schnittstelle zur Verfügung. Der Nachteil ist, dass ohne Code schwer zu entscheiden ist wie hoch die Frequenz des Pollings sein soll. Ist sie zu niedrig wird die Verzögerung des Exceptionhandling zu hoch. Ist die Frequenz zu hoch, so führt dies zu Leistungseinbußen.

Aus diesen Vor- und Nachteilen der beiden Pollingstrategien kann keine absolute Entscheidung für eine Strategie getroffen werden. Ein Exceptionhandlingmechanismus muss daher beide Strategien anbieten und automatisch die implicit polling Strategie verwenden. Dadurch wird es einfacher, da asynchrone Exceptions nicht ignoriert werden und zur Leistungsoptimierung eine niedrige Frquenz benutzt wird. Es steht aber die Option zur Verfügung dies auszuschalten und zusätzliche Polling nach der expliziten Polling-Strategie zu betreiben. [BM00]

5.5.2.3 Reentrant-Problem

Implicit Polling führt dazu dass die Zustellung der asynchronen Exception nicht deterministisch ist. Der Thread der für die Berechnung entlang des happy-path verantwortlich ist kann deshalb jederzeit vom Exception-Handler ”gestohlen” werden und hinterlässt so die Berechnung in einem inkonsistenten Zustand. Wird nun ein Exception-Handler gefunden und ruft rekursiv die inkonsistente Berechnung auf so spricht man von dem **Reentrant-Problem**. Die Lösung besteht darin asynchrone Exceptions kurzzeitig zu deaktivieren.

Es gibt drei Ansätze um Exceptions zu deaktivieren: individuelle-Exception-Deaktivierung, hierachische-Exception-Deaktivierung und prioritätengesteuerte-Exception-Deaktivierung.

Die **individuelle-Exception-Deaktivierung** deaktiviert nur Exceptions von dem explizit angegebenen Exceptiontyp, d.h. alle abgeleiteten Exceptiontypen müssen explizit aufgelistet werden, sollen auch sie deaktiviert werden. Die **hierachische-Exception-Deaktivierung** deaktiviert alle Exceptions von dem angegebenen Exceptiontyp einschließlich aller von ihr abgeleiteten Exceptiontypen. Die **prioritätengesteuerte-Exception-Deaktivierung** basiert darauf, dass Exceptiontypen Prioritäten zugeordnet

werden und Exceptions mit einer niedrigen als der angegebenen Priorität deaktiviert werden.

Der Vorteil der **individuelle-Exception-Deaktivierung** ist, dass sie sehr effizient ist. Werden aber neue abgleiteten Exceptiontypen eingeführt, so muss dies an vielen Stellen im Code stehen ohne dass Unterstützung dafür im Allgemeinen von der IDE angeboten werden kann. Dieses Problem wird durch die **hierachische-Exception-Deaktivierung** gelöst mit dem Nachteil dass die Implementierung komplexer und die Laufzeiten höher sind. Dasselbe Problem wird auch durch die **prioritätengesteuerte-Exception-Deaktivierung** gelöst aber mit anderen Nachteilen. Die Schwierigkeit liegt darin Prioritäten den einzelnen Exceptiontypen zuzuweisen und setzt die Kenntniss über die außergewöhnliche Natur jedes einzelnen Exceptiontyps voraus. Dies führt dazu dass das System weniger wartbar und weniger leicht verständlich ist. Da prioritätengesteuerte-Exception-Deaktivierung schon zu komplex ist um eine Bereicherung und Vereinfacherung darzustellen, wird diese von Buhr [BM00] abegelehnt und eine Kombination mit der hierachische-Exception-Deaktivierung steht nicht mehr in einer sinnvollen Position.

5.5.2.4 mehrere wartende asynchrone Exceptions

Durch das Polling kann es vorkommen dass zwischen zwei Polls mehrere asynchrone Exceptions auftreten. Aus Gründen der Robustheit darf keine dieser Exceptions ignoriert werden.

Laut Buhr [BM00] ist die angemessenste Lösung für diesen Fall noch nicht gefunden wurde und bedarf noch einigen Erfahrungen auf diesem Gebiet. Deswegen scheint eine FIFO Warteschlangelösung momentan am geeignesten. Die Einfachheit des Verständnisses und die geringen Implementierungskosten sprechen zusätzlich dafür. Wegen dem möglichen Fall, dass eine deaktivierte Exception die Bearbeitung der nach ihr in der Liste stehenden Exceptions verzögern kann, wird in diesem Fall eine nicht nach der Reihenfolge verlaufende Abarbeitung der Exceptions angestrebt. Ein Programmierer muss aber nun damit rechnen, dass selbst wenn zwei Exceptions dieselbe source execution und faulting execution haben, ihre Reihenfolge nicht garantiert wird. Dies kann als unvernünftig angesehen werden da sich vor allem kausales Ordnen von Exceptions gerade in verteilter Programmierung als vorteilig erwiesen hat.

5.5.2.5 Atomic-Action oder Conversation

Campbell und Randell [CR86] haben einen allgemeinen Exceptionhandlingansatz entworfen basierend auf **atomic action**s, die die Interaktion einer Gruppe von Prozessen oder

Threads einschließt. Mit jeder Aktion ist eine **handler clause** verknüpft, die einige oder alle Exceptiontypen abdeckt. Wird eine Exception signalisiert werden die geeigneten Exception-Handler ausgeführt und sind gemeinsam verantwortlich dafür das System **kooperativ** in einen gültigen Zustand zu überführen.

Für den Fall gleichzeitiger Signalisierung von Exceptions unterschiedlicher Exceptiontypen wird zu deren Auflösung zu einer einzigen Exception das **exception-tree-concept** verwendet. Es handelt sich um einen Baum, der alle Exceptiontypen enthält die mit einer Atomic-Action verknüpft sind und stellt diese unter eine partielle Ordnung in einer Weise, dass dem höheren Exceptiontyp ein Exception-Handler zugeordnet ist, der dafür gedacht ist auch alle Exceptiontypen in den unteren Ebenen des Baumes behandeln zu können.

5.5.3 Exceptionhandler-Kontext

Es gibt zwei Möglichkeiten für den statischen Kontext eines Exceptionhandlers: **exception-source context** und **handler-only context**. Bei dem exception-source context teilen sich die **guarded block** und der Exceptionhandler denselben statischen Kontext, dass es dem Handler ermöglicht wird auf lokale Variablen des guarded blocks zuzugreifen, wie es beispielsweise in Ada möglich ist. Im handler-only context hingegen hat der Exceptionhandler seinen eigenen statischen Kontext und kann deswegen nicht auf Variablen aus dem guarded block zugreifen, wie es beispielsweise in C++ realisiert ist. Dies hat den Vorteil, dass eine bessere Benutzung von Registern möglich ist. Aber beide Ansätze können den jeweils anderen nachahmen. [BM00]

Ein Exceptionhandler mit resume Semantik entspricht dem Aufruf einer Prozedur im guarded block und erfordert, dass der lexikalische Kontext des Handlers auf lokale Variablen im statischen Bereich zugreifen kann. Dies wird durch **lexical links** [BM00] bewerkstelligt, dabei handelt es sich um Zeiger auf semantische Kontexte, die dynamisch nach globalen Referenzen durchschritten werden. Die Erfordernis lexical links zu benutzen führt zu Komplexität und Verwirrung, da auf unerwartete Werte zugegriffen werden kann, die sich durch den Unterschied ergeben zwischen dynamischen und statischen Kontexten ergibt. Lexical Links werden wegen dem Verschachteln von Aufrufen, nicht wegen dem resumption model per se benötigt.

5.5.4 passendes Handling - zwischen Weiterreichungsmechanismus und Exception-Kontrollflussmechanismus

In der Quelle kann eine Exception mit throw oder resume Semantik geworfen werden und als Handlerimplementierung kann zwischen terminate und resume Semantik gewählt werden. Es werden nun alle Kombinationen besprochen und bestimmt welche Kombinationen sich als sinnvoll herausstellen.

Ein throw in der Quelle und ein Handler mit terminate Semantik passen, die Quelle möchte einen alternativen Kontrollfluss ansteuern und der Handler führt diesen Kontrollfluss aus und überspringt mittels terminate Semantik den ersten Kontrollfluss indem sich eine exceptional condition ereignet hat. Ein resume in der Quelle und ein Handler mit resume Semantik passen, die Quelle ist auf eine exceptional condition gestoßen, möchte durch einen Handler eine Korrektur vornehmen lassen und dann am point of raise mit der Ausführung fortfahren. Genau dies bewirkt der resume Handler. Ein throw in der Quelle und ein Handler mit resume Semantik kann nicht funktionieren, da durch throw ein Stack Unwinding stattfindet und der Kontext einschließlich point of raise zerstört wird, der mittels resume wieder angesprungen werden soll.

Ein resume in der Quelle und ein Handler mit terminate Semantik funktioniert bedingt. Es gibt zwei Fälle, im ersten Fall kann die terminate Semantik des Handlers ignoriert werden und nach Abarbeitung des Handlers zum point of raise (die Semantik des resume in der Quelle) gesprungen werden. Der Handler wird aber keine Korrekturmaßnahmen zur Fortführung der Operation an dieser Stelle vorgenommen haben und so wäre diese Variante unsicher.

Der zweite Fall besteht darin die terminate Semantik des Handlers durchzusetzen und die Blöcke zwischen point of raise und dem Handler zu überspringen. Dazu müsste nach unwind-Befehl oder direkt mittels unwind-Befehl während der Ausführung des Handlers ein Stack Unwinding stattfinden. Dies kann aber möglicherweise zu rekursivem Fortsetzen führen. Die geeigneteste Lösung ist die, zuerst das Stack Unwinding durchzuführen und danach den Handler auszuführen. So wird eine Exception mit resume Semantik wie eine Exception mit throw Semantik behandelt. Es kann kein rekursives Fortsetzen stattfinden, ein expliziter unwind-Befehl wird nicht benötigt und erleichtert die Benutzung. Dazu treten keine Seiteneffekte von beendeten Blöcken und keine Probleme durch unterschiedliche lexikalische Bereiche auf. Der zweite Fall besteht darin die terminate Semantik des Handlers durchzusetzen und die Blöcke zwischen point of raise und dem

Handler zu überspringen. Dazu müsste nach unwind-Befehl oder direkt mittels unwind-Befehl während der Ausführung des Handlers ein Stack Unwinding stattfinden. Dies kann aber möglicherweise zu rekursivem Fortsetzen führen. Die geeigneteste Lösung ist die, zuerst das Stack Unwinding durchzuführen und danach den Handler auszuführen. So wird eine Exception mit resume Semantik wie eine Exception mit throw Semantik behandelt. Es kann kein rekursives Fortsetzen stattfinden, ein expliziter unwound-Befehl wird nicht benötigt und erleichtert die Benutzung. Dazu treten keine Seiteneffekte von beendeten Blöcken und keine Probleme durch unterschiedliche lexikalische Bereiche auf.

5.6 Exceptionrepräsentation

Eine Exception wird nicht durch Fehlercodes dargestellt, sondern in der objektorientiereten Welt durch eine Instanz eines Klassenobjektes, welcher aber nicht darauf beschränkt ist, wie die folgende Liste zeigt. Eine Exception kann auf folgende Arten dargestellt werden:

- als statisches Klassenobjekt
- als neuerzeugte Klasseninstanz
- als neuerzeugte Klasseninstanz einschließlich Parameter

Dies ist orthogonal zu

- Bound Exception
- unbound Exception

5.6.1 bound Exception

Eine unbound Exception wird von einem Exception-Handler des gleichen Klassentyps behandelt. Eine bound Exception wird nur von einem Exception-Handler des gleichen Klassentyps und des gleichen zugeordneten Klassenobjektes behandelt indem die Exception geworfen wurde. In Ada findet sich dieses Konzept wieder.

Ein Nachbilden von bound Exceptions erfordert die Verwendung von Verzweigungsstrukturen. Wird dies für abgeleite Exceptions durchgeführt, so nehmen die Verzweigunsstrukturen rasch zu, die verwendet werden müssen und es wird erforderlich eine Programmierrichtlinie einzuführen. Eine Nachbildung ist deshalb nur für triviale Situationen geeignet.

5.6.2 Neuerzeugte Klasseninstanz einschließlich Parameter

In der Variante, in der eine Exception durch eine neuerzeugte Klasseninstanz dargestellt wird, die Parameter aufnimmt, können weitere Informationen zu der exceptional condition gesammelt werden. Die Verwendung von Parametern, anstelle von global geteilten Daten, verhindert das Auftreten von Seiteneffekten im Falle von Gleichzeitigkeit, bzw. das Zugreifen auf nichtzusammengehöriger bzw. überschriebenen Informationen im Falle des "explicit exception polling". Nach Yemini et. al [YB85b] sind Parameter eine Notwendigkeit. Ansonsten müsste auf globale Variablen zurückgegriffen werden, um die Umstände zu beschreiben unter denen der Exceptionhandler aufgerufen worden ist. Dies würde aber die Kopplung zwischen dem Exceptionmelder und dem Aufrufer erhöhen. Die Alternative zu globale Variablen wäre einen seperaten Exceptionhandler für jeden **point of raise** zu erstellen. Dies reduziert die Mächtigkeit des Exceptionhandlermechanismus, da mehr Handler mit ähnlicher Funktion bereitgestellt werden müssen.

Als Parameter können auch Routinen verwendet werden, um beispielsweise einen Default-Exeption-Handler anzugeben, der ausgeführt wird, falls kein passenderer gefunden wird. Wenn Zeiger als Parameter übergeben werden ist Synchronisation erforderlich, da sowohl die source execution wie auch die faulting execution auf die gemeinsame Ressource zugreifen kann. Gelöst wird diese Problematik durch gegenseitigen Ausschluss. Oft ist diese Vorgehensweise aber nicht nötig und kostet Laufzeit. Aus diesem Grund muss der Programmierer explizit diese Technik anwenden.

5.6.3 abgeleitete Exception

Im Falle von abgeleiteten Exceptions kann für eine geworfene abgeleitete Exception D entweder ein Handler für genau diesen Klassentyp oder aber für eine Oberklasse aufgerufen werden. Im zweiten Fall gibt es folgende Besonderheiten zu beachten.

Wird ein Exception-Handler für die Oberklasse ausgeführt, so hat dieser nur Zugriff auf die Attribute dieser Oberklasse. Wird nach dem Verlassen des Handlers auf die Exceptioninstanz zugegriffen so können die nicht in der Oberklasse definierten Attribute nicht gesetzt oder nicht kongruent zu den Werten der Attributen in der Oberklasse sein.

Im Falle des termination Modell ist dies laut [BM00] nicht möglich, wohingegen meiner Meinung nach dies aber bei einem erneuten throw der gleichen Exceptioninstanz in einem Handler für eine Oberklasse dennoch auftreten kann, wenn danach ein Handler für einen Unterklassentyp diese Exceptioninstanz fängt.

Im resumption Modell existiert dieses Problem aufgrund des Downcasts. Es handelt sich dabei aber um kein Exceptionspezifisches Problem, sondern um ein Subtyping Problem.

Es ist denkbar mehrfache Ableitungen für Exceptions zuzulassen, dies ist aber zu kompliziert, da es zu viel Spielraum für Interpretationen gibt und der Programmierer sich an strikte Programmierrichtlinien halten müsste, die nicht automatisch kontrollierbar sind. Dies soll im folgenden kurz skizziert werden.

Es existieren die Exceptions E1 und E2 als nicht voneinander in Beziehung stehende Oberklassen und eine Exception E3 die von E1 und E2 abgeleitet worden ist. Es gibt nun drei Fälle die für den Fall dass Exception E3 geworfen wird, eintreten können.

Es wird ein Exception-Handler für E1 und E2 ausgeführt und die exceptional condition ist behoben. Der Exception-Handler für E1 oder E2 wird ausgeführt und die exceptional condition ist behoben. Je nach Ableitung und Semantik kann nur der Exception-Handler E1 (bzw. E2) die exceptional condition beheben.

Diese Flexibilität verhindert eine einheitliche Richtlinie und wird deshalb als ungeeignet angesehen.

5.6.4 Exceptionliste

Die Exceptionsliste dient der statischen Verifikation, ob jede Exception lokal oder von einer aufgerufenen Prozedur behandelt wird. Findet die Prüfung während der Laufzeit statt, so kann die Exception in eine besondere **failure exception** umgewandelt oder das Programm beendet werden. Die Exceptionsliste ist Teil der Signatur einer Routine und spezifiziert welche Exceptiontypen zu dem Aufrufer weitergereicht werden können.

Die Exceptionliste wird als zu restriktiv angesehen. So ist sie auf C++ Templates nicht anwendbar, da nicht abzusehen ist welche Exceptiontypen dadurch signalisiert werden können. Denkbar wäre die Exceptionliste bei der Instanziierung anzugeben, um so mehrere Versionen mit unterschiedlichen Signaturen anzulegen. Handelt es sich aber um ein vorkompiliertes Template so existiert nur eine Signatur für alle Aufrufe.

Wenn eine Funktion z in ihrer Exceptionliste den Exceptiontyp E1 stehen hat, so kann diese Funktion eine Exception vom Typ E1 an den Aufrufer weiterreichen. Wird z von einer Funktion y aufgerufen ohne E1 zu behandeln noch in ihrer Exceptionliste zu führen, da eine Funktion x die Funktion y aufruft und dort E1 behandelt wird, so wird automatisch zur Kompilierzeit die Signatur erweitert. Ist die Funktionen y vorkompiliert so funktioniert dies nicht. Hierbei handelt es sich um das **Sichtbarkeitsproblem** 5.4.2.1. Die Exceptionliste automatisch mit weiteren Exceptiontypen zu erweitern ist nicht zu

befürworten, da so die Funktionen schnell aufgebläht werden können und damit an Robustheit verlieren.

Eine Exceptionsliste für eine routine zu bestimmen wird schwierig, wenn nicht sogar unmöglich, wenn **asynchrone Exceptions** verwendet werden, da diese zu jeder Zeit weitergereicht werden können.

6 Welchen Leistungsumfang an Ausnahmebehandlung findet man in Mainstream-Programmiersprachen?

Im folgenden soll eine Liste aufgestellt werden, die alle wichtigen Features aufzählt, die Einfluss auf die Programmierung haben, sowie jene, die eine Programmiersprache unterstützen sollte, um effektiv Gebrauch von Exceptions zu nehmen.

6.1 Leistungen

Die Leistungsbeschreibung kann in zwei Kategorien unterschieden werden. Zum einen ist das die Semantik und die Realisierung der Exception in der Programmiersprache, also die inhärenten Eigenschaften der Programmiersprache bezüglich Exceptions. Die andere Kategorie setzt sich aus allen Konstrukten und Pattern zusammen, die durch Befehlsworte in der Programmiersprache (meist) explizit vom Benutzer verwendet werden können.

6.1.1 Exception

Die Architektur einer Programmiersprache und die Implementierung von Exceptions legt Prinzipien des Exceptionhandling in dieser Programmiersprache fest, die sich durch folgende Eigenschaften beschreiben lassen.

6.1.1.1 Exceptiondefinition in Programmiersprache

Die Definition einer Exception, bestimmt den Codeentwurf von Anwendungsfällen und die Verwendungsart von Exceptions in der Programmiersprache.

6.1.1.2 Repräsentation

Die Repräsentation der Exception kann ein beliebiger Datentyp, ein String oder ein eigens für Exceptions definierter Datentyp sein.

6.1.1.3 Exception Ableitungshierachie

Existiert ein Datentyp von dem alle weiteren Exceptiontypen abgeleitet sind, oder kann Exception durch jeden beliebigen Datentyp dargestellt werden.

6.1.1.4 Exception-Typsicherheit

Hier wird betrachtet, ob Exceptions auf Typsicherheit geprüft werden.

6.1.1.5 Exception-driven design

Ist die Programmmiersprache so entworfen, dass jede **exceptional condition** durch Exceptions signalisiert werden oder ist es notwendig if-Abfragen zu verwenden, um die Korrektheit des Programms zu garantieren?

6.1.1.6 Rückgabewerte als Fehlerindikator

Sollen Fehler per Rückgabewert behandelt werden oder wird die Fehlerbehandlung auch durch Exceptions modelliert?

6.1.1.7 Un-/checked Exceptions

Findet eine Überprüfung statt, ob Exceptionhandler für jede potentiell signalisierte Exception angelegt worden sind?

6.1.2 Konstrukte / Pattern

6.1.2.1 Exception-chaining

Nach dem Erkennen einer **exceptional condition** wird eine Exception signalisiert. Dies ist die ursprüngliche Exception. Während diese Exception behandelt wird, können weitere Exception signalisiert werden. Diese Exceptions können dazu führen, dass die ursprüngliche Exception verloren geht. Um dies zu verhindern, kann einer zusätzlich signalisierten Exceptions ein Verweis auf die Exception hinzugefügt werden, die zur Zeit der Signalisierung der neuen Exception aktiv war. Diese Eigenschaft bezeichnet, ob es

möglich ist zu ermitteln, ob die aktive Exception andere Exceptions unterdrückt hat und welche Exceptions dies waren.

6.1.2.2 Guarantee-execution-Block

Dieses Konstrukt ermöglicht es dem catch-Block einen Codeblock zuzuordnen, der immer nach dem catch-Block ausgeführt wird. Er wird also ausgeführt, wenn Exceptions und auch wenn keine Exceptions signalisiert worden sind.

6.1.2.3 Try-succeeded-Block

Dieses Konstrukt erlaubt es dem catch-Block einen Codeblock zuzuordnen, der nur ausgeführt wird, wenn im try-Block keine Exception signalisiert worden ist.

6.1.2.4 Try-with-resources-Block

Dieses Konstrukt sorgt dafür, dass beim Signalisieren einer Exception in einem try-Block, alle verwendeten Resourcen freigegeben werden. Während der Freigabe einer Resource kann eine zusätzliche Exception auftreten. Dieses Konstrukt kümmert sich darum, dass auch im Fall zusätzlich signalisierter Exceptions alle Resourcen freigegeben werden.

6.1.2.5 Nested-exceptions

Dieses Konstrukt beschreibt, ob es möglich ist mehrere try-catch Blöcke ineinander zu verschachteln.

6.1.2.6 Multiple-exception-catching

Dieses Konstrukt erlaubt es unterschiedliche Exceptions, die nicht von einer gemeinsamen Oberklasse abgeleitet sind, gemeinsam durch einen Exceptionhandler behandeln zu können.

6.1.2.7 Exception-An-Aufrufer Konstrukt

Dieses Konstrukt erlaubt es, dass Exceptions, die in einer Methode signalisiert werden, automatisch an den Aufrufer weitergereicht werden. Die Verantwortung Exceptionshandler bereitzustellen wurde damit dem Aufrufer übertragen.

6.1.2.8 Default top-level Exceptionhandler

Diese Eigenschaft beschreibt, ob ein Exceptionhandle existiert, der aufgerufen wird, bevor das Programm abstürzt und dem Betriebssystem eine Fehlermeldung zurückgibt.

6.1.2.9 Asynchrone Exceptions

Dieses Kriterium beschreibt, ob in der Sprache eine Möglichkeit besteht, eine Exception in einem anderen Thread zu signalisieren.

6.2 C++

6.2.1 Exception

6.2.1.1 Exceptiondefinition in Programmiersprache

Die Sprachdefinition [Str] von C++ sagt aus, dass Exceptionhandling verwendet werden soll, um Fehler zu behandeln.

6.2.1.2 Repräsentation

Eine Exception kann von einem beliebigen Typ sein, z.B. auch vom Typ int. [cpla] Es ist zu beachten, dass Exceptionhandler nur zu Exceptions passen dessen Typ sie als Argument führen. Exceptionhandler behandeln auch Referenzen des Exceptiontyps für den sie erstellt worden sind. [MSDb]

6.2.1.3 Exception Ableitungshierachie

In der Headerdatei <exception> der C++ Standardbibliothek ist eine Basisklasse vorhanden, die speziell für das Deklarieren von Objekten entworfen wurde, die als Exceptions signalisiert werden können. Diese Klasse enthält einen Standardkonstruktor, einen Kopierkonstruktor, Operatoren und Destruktoren, sowie eine zusätzliche virtuelle Instanzfunktion namens what, die eine nullterminierte Zeichenkette vom Typ char* zurückgeben kann. Diese Funktion what kann in abgeleiteten Klassen überschrieben werden, um eine Beschreibung der Exception ausgeben zu können. [cpla]

6.2.1.4 Exception-Typsicherheit

C++ unterstützt typsichere Exceptionhandler.

6.2.1.5 Exception-driven design

Exception-Driven Design existiert in C++ nicht. [cpp] Dies wird dadurch verdeutlicht, dass der Visual Studio 2010 Compiler eine Option hat, um das Excceptionmodell zu bestimmen [MSDa] oder es vollständig zu deaktivieren. [MSDb]

6.2.1.6 Rückgabewerte als Fehlerindikator

Exceptions sollten nicht verwendet werden, wenn potentielle Logik oder Eingabefehler vorliegen. Dazu zählt auch das Überschreiten einer Arraygrenze. In diesen Fällen sollte auf den Mechanismus von Rückgabewerte zurückgegriffen werden. [MSDb]

6.2.1.7 Un-/checked Exceptions

Soll eine Funktion in der Lage sein nach außen Exceptions zu signalisieren, so muss dies durch eine throw-clause in der Funktionsdeklarations angegeben werden.

6.2.2 Konstrukte / Pattern

6.2.2.1 exception-chaining

Wird eine zusätzliche Exception in C++ signalisiert, so geht die aktive Exception verloren. Wird die Funktion std::throw_with_nested anstelle von throw verwendet (explizites exception-chaining), um eine zusätzliche Exception zu werfen, so wird die aktive und die durch die Anweisung signalisierte Exception in ein Objekt vom Typ nested_exception gepackt. Mit Hilfe von dynamic_cast kann auf die erste Exception zugegriffen werden. [GSK11]

C++ unterstützt nicht von Haus aus implizites exception-chaining. Es kann jedoch leicht durch das Entwerfen einer eigenen Basisklasse für Exceptions realisiert werden. [Ins] [Kra]

6.2.2.2 Guarantee-execution-Block

Ein solches Konstrukt existiert in C++ nicht. Es kann aber mittels Konstruktoren durch die ”resource acquisition is initialization“ Technik nachgebildet werden. [Str]

6.2.2.3 Try-succeeded-Block

In C++ [cpp] existiert ein solches Sprachkonstrukt nicht.

Es kann leicht nachempfunden werden, indem beispielsweise eine boolsche Variable "**bool trysucceeded = true;** " vor den try-Block gesetzt wird. In allen catch-Handlern, die dem try-Block zugeordnet sind, muss die Variable auf false gesetzt werden. Hinter den try-catch Code kann semantisch ein try-succeeded-Block folgen, der mittels **if(trysucceeded)** eingeleitet wird.

6.2.2.4 Try-with-resources-Block

Durch die "Resource Acquisition Is Initialization" Technik ist ein try-with-resources-Block nicht nötig, da die Objekte in C++ auf dem Stack liegen und beim Signalisieren einer Exception beim stack unwinding automatisch freigegeben werden. [Wikb]

6.2.2.5 Nested-exceptions

Das Verschachteln von try-catch-Blöcken ist möglich. Durch den Ausdruck "**throw;**" (throw ohne Parameter) kann eine Exception an den umschließen try-catch-Block weitergereicht werden. [cpla]

6.2.2.6 mMltiple-exception-catching

Durch eine Ellipsis im catch-Konstrukt werden Exceptions von einem beliebigen Typ behandelt. [cpla]

6.2.2.7 Exception-An-Aufrufer Konstrukt

Wird in einer Funktion keine Exception durch einen passenden Exceptionhandler mittels try-catch Konstrukt behandelt und steht diese Exception aber in der throw-clause der Funktion, so impliziert dies die Semantik des Exception-An-Aufrufer Konstrukt. [cpla]

6.2.2.8 Default top-level Exceptionhandler

Mittels der Methode set_terminate() kann eine Funktion bestimmt werden, die aufgerufen werden soll, wenn für eine signalisierte Exception kein passender Exceptionhandler gefunden wird oder eine andere außergewöhnliche Situation sich ereignet, die das Fortsetzen des Exceptionhandling unmöglich macht. [cplb] [MSDb]

6.2.2.9 Asynchrone Exceptions

Visual C++ [MSDe] unterstützt asynchrone Exceptions. Dazu steht der Datentyp exception_ptr und die Funktionen current_exception, rethrow_exception und copy_exception zur Verfügung. Um die Funktionen verwenden zu können, muss die Exception vom Typ exception_ptr sein. Mit current_exception oder und copy_exception kann der einer Variablen vom Typ exception_ptr eine signalisierte Exception zugewiesen werden. Die Funktion rethrow_exception extrahiert die Exception aus dem exception_ptr Objekt und signalisiert die Exception in dem Kontext des primären Threads. Die dadurch signalisierte Exception kann wie jede andere Exception in C++ behandelt werden.

6.3 Python

Python wurde am 20. Februar 1991 erstmals veröffentlicht.

6.3.1 Exception

Nach [Docg] sollten if-Abfragen in Python verwendet werden um zu verhindern, dass Exceptions signalisiert werden. Nur wenn davon ausgegangen wird, dass die Bedingung in den meisten Fällen erfüllt ist, ist es aus Optimierungsgründen ratsam auf die if-Abfrage zu verzichten. Ein try/except Block führt nur zu geringen Leistungseinbußen wenn keine Exceptions signalisiert werden. Wird tatsächlich eine Exceptions signalisiert so ist dies teuer.

Nach van Rossum [vR91] und [GvR11] entspricht der Python Exceptionmechanismus dem von Modula-3. Es handelt sich dabei um dynamische Weiterreichung. Python stützt sich vor allem auf runtime checking.

6.3.1.1 Exceptiondefinition in Programmiersprache

Syntaxfehler, die während dem Kompilieren entdeckt werden, lösen ab Python 2.1 Exceptions aus, die neben dem Dateinamen auch die Zeilennummer des Fehler enthalten. [Kuca]

6.3.1.2 Repräsentation

Bis Python 2.6 kann eine Exception von einem beliebigen Typ sein. Ab Python 3.0 muss jede Exception sich von BaseException ableiten. [vRb]

Ab Python 1.5 alpha 4 sind Exceptions standardmäßig als Klassen und nicht mehr als Strings modelliert. [Docj] Erst mit der Python 2.0 wurde auch die -X Befehlskommandooption entfernt, die alle Standardklassen in Strings verwandelte.[Zad]

Ab Python 2.5 sind Exceptions keine classic classes sondern new-style classes. (die bereits in Python 2.2 eingeführt worden sind.) [Kucb] Dies erhöht die Geschwindigkeit des Exception Handling gegenüber der der Vorgängerversion 2.4 um ca. 30 Prozent, aufgrund dramatisch kürzerer Instantiierungszeit.[Kucc]

In [vR05] ist festgehalten, dass mit den Exceptions als new-style classes, bei der BaseExceptionType Klasse, die nun als Interface für alle Exceptions dient, das Attribut message eingeführt wurde, das wiederum das alte Attribut args ablösen sollte. In args wurden verschiedene Informationen hinein gepackt, die per Index addressierbar waren. Die Idee war dies abzulösen dass in den abgeleiteten Klassen stattdessen Attribute benutzt werden sollen, und in dem abgeleiteten Attribut der BaseException Klasse die Hauptinformation gespeichert werden würde.

Laut Brett Cannon ([Can07]) stellte es sich jedoch heraus, dass der Übergang für bestehenden Programmcode zu hart sei, und viele Klassen und Methoden sich automatisch Werte von args holen. Vor allem scheiterte die Umsetzung aber aufgrund der Anbindung an die C API für die der Übergang nur mit zahlreichen Änderungen und Work-Arounds durchführbar wäre.

Deshalb wurde in Python 2.7 das Attribut message von der BaseException Klasse entfernt.

6.3.1.3 Exception Ableitungshierachie

Weiter aus [Docj] ist zu entnehmen, dass die unterschiedlichen Exception-Klassen von Python in dem „standard library module“ exception.py definiert sind. In 6.1 ist zu sehen dass gleich mit Vererbung und Ableitung eine Exception-Hierachie eingeführt worden ist. Eine Exception wird als Klassenobjekt Exception dargestellt. In Version 1.5 gibt es nur zwei abgeleitete Klassen. SystemExit für schwerwiegende Ausnahmen die zum Programmabbruch führen und StandardError als Oberklasse für alle auftretenen Fehler. Benutzerdefinierte Exceptionklassen können durch Ableiten von der Klasse Exception erzeugt werden.

In Python 3.2.2 (4. September 2011) ist aus [Docf] zu erkennen, das die allgemeinste Exception durch eine Klasse namens BaseException dargestellt wird, wie in 6.2 zu sehen ist. Die abgeleiteten Klassen neben der SystemExit- und der in Exception- umbenannten StandardError-Klasse noch um KeyboardInterrupt - für Tastatureingaben

die das Programm unterbrechen (dies geschieht durch das Kürzel Control+C, oder je nach Belegung in Abhängigkeit von dem Betriebssystem [Doci, 8.]) - und GeneratorExit - für Ausnahmen bei Gebrauch von Generatoren - als direkt abgeleiteten Klassen von Exception ergänzt worden sind. Die Umstrukturierung ([Kucc]) mit BaseException als Kopf der Exception Hierachie fand statt, da ein „except:“ Befehl ohne Angabe eines Exceptiontyps alle Exceptiontypen abgefangen hat. Darunter auch SystemExit und KeyboardInterrupt. Diese stellen aber primär keine Programmfehler dar und werden nur in den seltensten Fällen gewollt aufgefangen. Es bietet sich nun die Möglichkeit durch ein „except Exception:“ alle nicht als Spezialfall ausgesonderten Exceptiontypen abzufangen.

6.3.1.4 Exception-Typsicherheit

6.3.1.5 Exception-driven design

In Python sind alle Funktionen so entwurfen, dass wenn sie ihre Funktion nicht nach der Spezifikation ausführen können, eine Exception signalisiert wird. [GvR11]

6.3.1.6 Rückgabewerte als Fehlerindikator

In Python gilt ein Rückgabewert von 0 als fehlerfreies Ausführen des Programms, ein Wert ungleich 0 weist auf einen Fehler hin. Dabei zählen Exceptions, die erfolgreich durch einen Handler aufgefangen wurden, nicht als Fehler. In den meisten Fällen werden Fehler durch Exceptions modelliert. Nur in wenigen Fällen, beispielsweise beim Beenden eines Pythonprogramms wird ein Rückgabewert an das Betriebsystem benutzt um anzuzeigen, ob ein Fehler auftrat. [Doci, 2.2]

6.3.1.7 Un-/checked Exceptions

In Python existieren keine checked Exceptions.

6.3.2 Konstrukte / Pattern

6.3.2.1 exception-chaining

Ab Python 3.0 wird exception-chaining unterstützt. Es gibt eine implizite und eine explizite Form. Implizites exception-chaining findet statt, wenn eine weitere Exception in einem catch-Block oder finally-Block signalisiert wird. Mit der Syntax in Abbildung 6.3 kann explizites exception-chaining ausgeführt werden.

6.3.2.2 Guarantee-execution-Block

Der Verwendung von dem try-catch Konstrukt funktioniert in Python genau so wie es im Kapitel „Exception-Grundverständnis“ erläutert wurde, lediglich mit dem Unterschied dass der catch-Block durch das Befehlswort except kenntlich gemacht wird. Der finally-Block ist auch hier optional. [Doci, 8.]

6.3.2.3 Try-succeeded-Block

Zusätzlich existiert in Python ein try-succeeded-Block [Doci, 8.] der durch das Befehlswort **else** im Anschluss an den letzten catch-Block eingeleitet wird. Dieser wird nur ausgeführt wenn im try-Block keine Exception geworfen wurde. Somit verhindert man das Programmcode, der nur ausgeführt werden soll wenn im vorherigen Codeblock keine Exception geworfen wurde, im selben try-Block wie der vorherige Codeblock stehen muss. Zusätzlich wird mit dem try-succeeded-Block verhindert, dass Exceptions die der Code aus dem try-succeeded-Block werfen kann nicht von den gleichen catch-Blöcken des try-Blockes „geschluckt“ werden können. Ein try-catch Konstrukt sieht somit in Python aus wie in 6.4.

6.3.2.4 Try-with-resources-Block

Ab Python 2.5 existiert ein ähnliches Konstrukt wie der try-with-resources Block. (vgl. [Pon11]) Es ist in [vRNC05] beschrieben und ist wie in 6.5 definiert:

6.3.2.5 Nested-exceptions

Es können mehrere with-Blöcke in Python ineinander verschachtelt werden. Mit der Methode kann festgelegt werden, ob die Exceptions die im BLOCK geworfen wurde unterdrückt (Rückgabewert true) werden soll oder nicht. Je nach Implementierung der Methode können auch nur bestimmte Exceptiontypen unterdrückt werden bzw. noch andere Aktionen ausgeführt werden.

6.3.2.6 Multiple-exception-catching

Aus [vRa] und [Kuc06] ist zu erfahren, dass bereits in Python 1.4 (Oktober 1996) multiple-exception-catching vorhanden war.

6.3.2.7 Exception-An-Aufrufer Konstrukt

6.3.2.8 Default top-level Exceptionhandler

6.3.2.9 Asynchrone Exceptions

Mit der Funktion PyThreadState_SetAsyncExc() kann in einem beliebigen Thread eine Exception signalisiert werden. [Doch]

Aus [Doce]:Wird eine Exception von keinem passenden Handler gefangen, dann wird in Python bevor das Programm sich beendet, die Methode sys.excepthook aufgerufen, die die Argumente exception class, exception instance und traceback erhält. Wenn sys.excepthook eine andere Methode zugewiesen wird, kann so eine benutzerdefinierte Behandlung von top-level Exceptions implementiert werden.

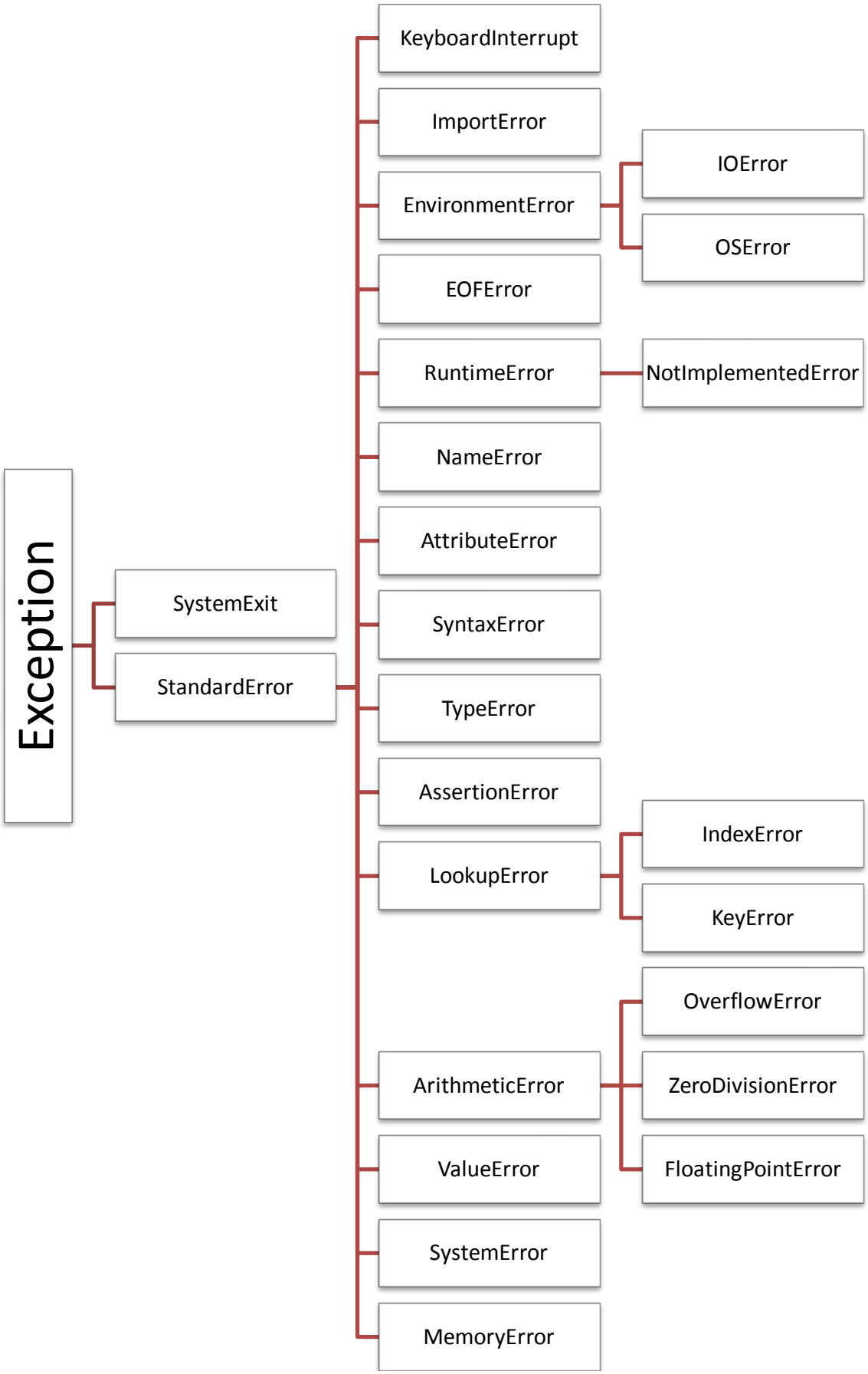


Bild 6.1: Python 1.5: Die Standard Exception Hierarchie. Die mit Asterisk markierten Exceptions sind neu in dieser Version.

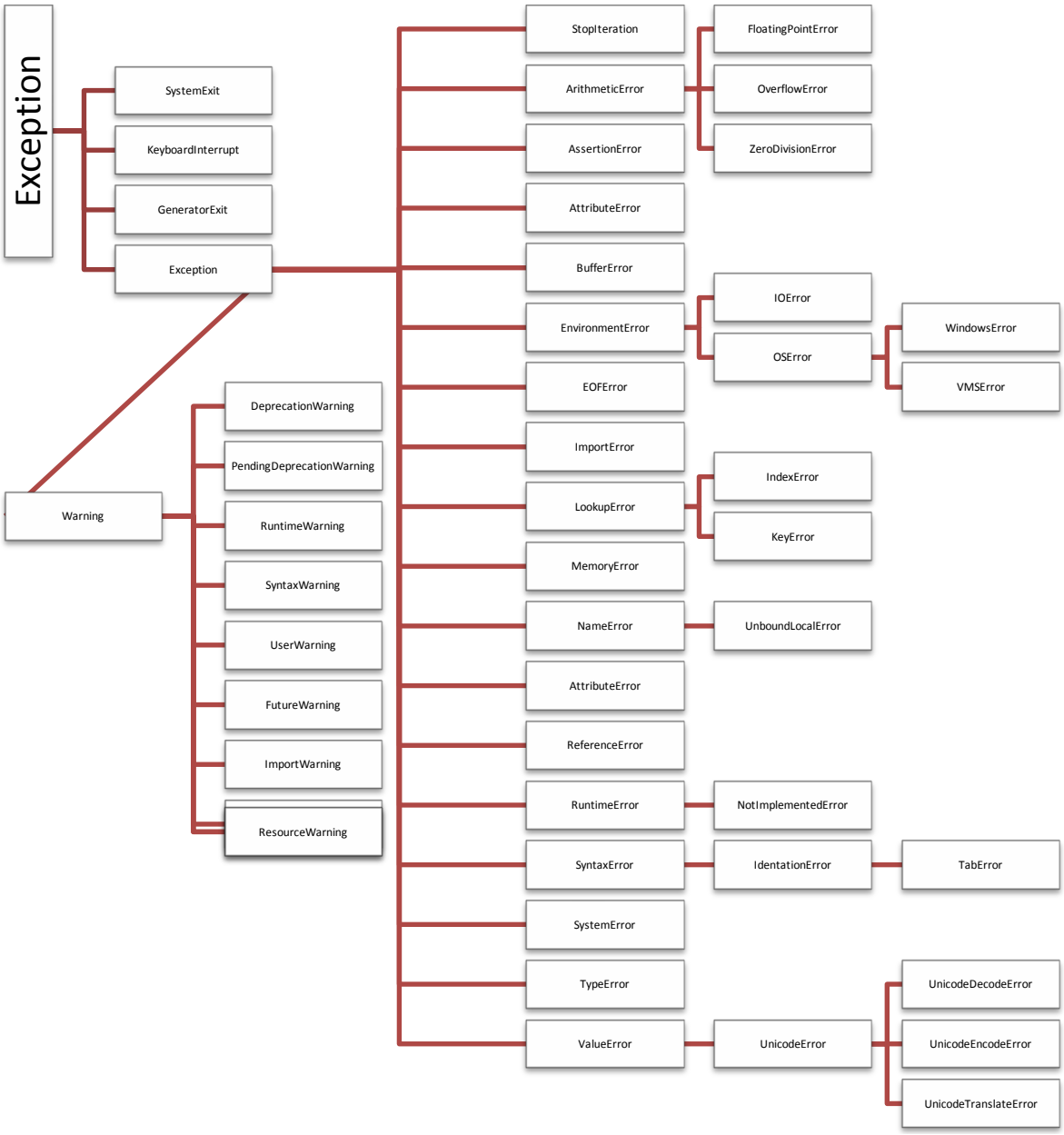


Bild 6.2: Python 3.2.2: Die Standard Exception Hierarchie.

```
raise SecondaryException() from primary_exception
```

Bild 6.3: Python 3.0: explizites exception-chaining.

```
    try:
         // Programmcode, der eine Exception auslösen kann
     except ExceptionBeispielTyp:
         // Programmcode zum Behandeln der Exception
     else:
        // Programmcode der nur ausgeführt wird, wenn keine Exception
           geworfen
     finally:
```

Bild 6.4: Ein try-catch Block in Python

```
Anwendung:
   with EXPR as VAR:
            BLOCK
Definition:
        mgr = (EXPR)
        exit = type(mgr).__exit__  # Not calling it yet
        value = type(mgr).__enter__(mgr)
        exc = True
        try:
            try:
                VAR = value  # Only if äs VAR" is present
                BLOCK
            except:
                # The exceptional case is handled here
                exc = False
                if not exit(mgr, *sys.exc_info()):
                    raise
                # The exception is swallowed if exit() returns true
        finally:
            # The normal and non-local-goto cases are handled here
            if exc:
                exit(mgr, None, None, None)
```

Bild 6.5: Try-with-resources Block Anwendung und Definition in Python

6.4 Java

6.4.1 Exception

6.4.1.1 Exceptiondefinition in Programmiersprache

In Java wird eine Exception wie folgt definiert: ”An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program’s instructions.” [Docc, What Is An Exception?]

6.4.1.2 Repräsentation

In einem laufenden Javaprogramm wird eine Exception als Exceptioninstanz repräsentiert. Dabei handelt es sich um ein Object der Klasse oder einer abgeleiteten Klasse von java.lang.Throwable, die von der Methode erstellt wird, in der ein Exceptionevent aufgetreten ist. Anschließend wird dieses an das runtime system übergeben. Das Erstellen und übergeben der Exceptioninstanz an das runtime system wird als Werfen einer Exception bezeichnet. [Docc, What Is An Exception?]

6.4.1.3 Exception Ableitungshierachie

Folgende Informationen stammen aus [Docc, How To Throw Exceptions?], [Docc, Unchecked Exceptions — The Controversy], [pro]. Die Standard Exception Hierachie in Java ergibt sich daraus, dass alle Exceptiontypen sich von der Klasse java.lang.Throwable ableiten. Diese selber ist ein System.Object. Von dieser allgemeinen Exceptionklasse leiten sich direkt java.lang.Error und java.lang.Exception ab, wie in der Abbildung 6.6 zu sehen ist.

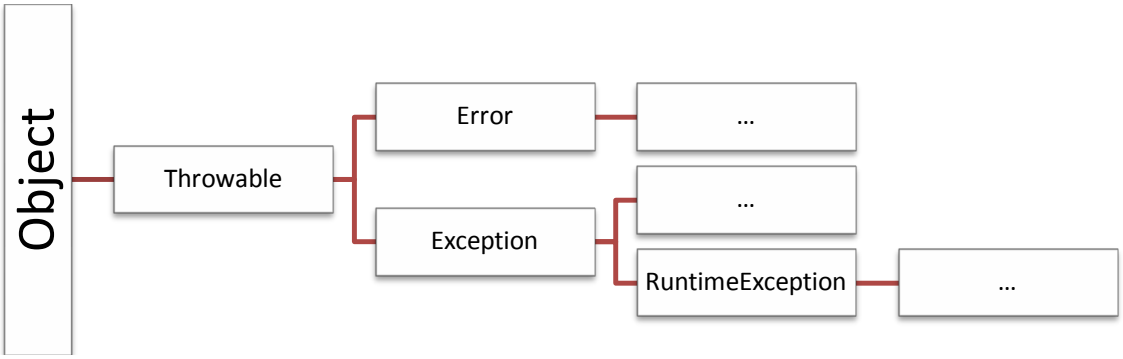


Bild 6.6: Java: Die Standard Exception Hierachie

6.4.1.4 Exception-Typsicherheit

In Java SE 7 wurde Exception-Typsicherheit im throw-Befehl hinzugefügt. Dabei ist das Exception-Object bei Verwendung von multiple-exceptions implizit final und kann nicht verändert werden. Die Exception-Typsicherheit ist nur garantiert, wenn dem Exception-Object kein anderer Wert im catch-Block zugewiesen wird. [Wika] [Doca]

6.4.1.5 Exception-driven design

[Docb, mkdir()]: Leider besteht selbst in Java noch kein durchgängiges Exception-Konzept. So wird immer noch in Java SE 7, wenn wir nur beispielhaft die Methode mkdir() betrachten bei einem Fehlschlagen der Funktion keine Exception geworfen, sondern false als Rückgabewert geliefert. Diese Methode kann aber sehr wohl eine Exception werfen, allerdings nur von dem Typ SecurityException.

Nach Christian Ullenboom [Ull] verfügt Java so über die erforderlichen sprachlichen Mittel, dennoch sind Exceptions immer noch kein zentrales durchgängiges Entwurfselement in jedem Java-Objekt .

6.4.1.6 Rückgabewerte als Fehlerindikator

Error repräsentiert außergewöhnliche Bedingungen, die außerhalb der Anwendung liegen. Die Anwendung kann dies weder hervorsehen noch mit der Programmausführung fortfahren. Beispielsweise wenn ein ”dynamic linking failure” oder ein anderer ”hard failure” in der JVM auftritt, wirft die JVM eine Exceptioninstanz vom Typ java.lang.Error. Einfache Programme ”werfen” noch ”fangen” Exceptions vom Typ java.lang.Error.
java.lang.Exception enthält eine ausgesonderte direkt abgeleitete Klasse namens java.lang.RuntimeException. Sie repräsentiert außergewöhnliche Bedingungen, die innerhalb der internen Strukturen der Anwendung liegen. Die Anwendung kann diese weder hervorsehen noch mit der Programmausführung fortfahren, da es sich um Programmfehler handelt (Logikfehler, falsche API Verwendung). Dazu zählen arithemtische Fehler (Division durch 0), Zeigerfehler (nullpointer), Fehler bei der Indizierung von Arrays durch einen zu kleinen oder zu großen Index. Die Anwendung sollte beendet und der Fehler korrigiert werden.

[Docb, mkdir()]: Leider besteht selbst in Java noch kein durchgängiges Exception-Konzept. So wird immer noch in Java SE 7, wenn wir nur beispielhaft die Methode mkdir() betrachten, bei einem Fehlschlagen der Funktion keine Exception geworfen, sondern false

als Rückgabewert geliefert. Diese Methode kann aber sehr wohl eine Exception werfen, allerdings nur von dem Typ SecurityException.

6.4.1.7 Un-/checked Exceptions

In Java existiert das Konzept der checked exceptions (geprüfte Ausnahmen) und der unchecked exceptions (ungeprüfte Ausnahmen).

Im Fall von checked exceptions muss jede Codezeile, die eine Exceptions werfen kann in einem try-Block mit einem passenden catch-Block stehen oder muss durch eine throw-Klausel weitergeleitet werden. Im zweiten Fall muss in der Aufrufhierachie ein catch-Block den Exceptiontyp in der throw-Klausel behandeln.

Im Fall von unchecked exceptions handelt es sich fast ausschließlich um abgeleitete Exceptiontypen von der Klasse java.lang.RunTimeException. Dieser Exceptiontyp beschreibt eine Exception die aufgrund eines Programmierfehlers geworfen wurde. Eine Behandlung ist demnach nicht möglich und muss demzufolge auch nicht gefangen werden. Dies ist aber möglich und wird meistens auch gemacht. Viele Entwickler schreiben in ihre Dokumentation welche RunTimeExceptions ihre Methoden werfen können. Eine Unterstützung aus Eclipse heraus wie bei den checked exceptions findet leider nicht statt. [Docd]

6.4.1.8 Exception-surpressing

Das Phänomen der Suppressed Exception (unterdrückte Ausnahme) kann auftreten wenn im try-Block eine Exception geworfen wird und im catch-Block oder finally-Block(bei Verwendung des Exception-Weiterleitung-An-Aufrufer Konstrukts) auch eine Exception geworfen wird. Dadurch wird die erste Exception unterdrückt. [Ull]

6.4.2 Konstrukte / Pattern

6.4.2.1 exception-chaining

Es können auch sogenannte "chained exceptions"(kaskadierte Exceptions) erzeugt werden. Mittels getCause() und initCause() können Exceptions kaskadiert werden und die ursprüngliche Exception, die weitere Exceptions ausgelöst hat, mitgeführt werden. Mit getStackTrace() erhält man durch die Exceptioninstanz auch den stacktrace. [Ull, 6. Exceptions]

6.4.2.2 Guarantee-execution-Block

In Java wird diese Semantik durch das finally-Konstrukt bereitgestellt. Jedoch wird der finally-Block nicht immer ausgeführt: wenn die JVM beendet wird; während ein try/catch Block ausgeführt wird oder ein Thread beendet oder unterbrochen wird, der einen try/catch Block ausführt. [Docc, The finally Block?]

6.4.2.3 Try-succeeded-Block

Dieses Konstrukt existiert in Java [Ora] nicht. Es kann aber wie in [6.2.2.3] beschrieben, nachgebildet werden.

6.4.2.4 Try-with-resources-Block

In Java SE 7 wurde dafür das Sprachmittel try-with-resources [Docc, The try-with-resources Statement] eingeführt. Es wird wie in der Abbildung 6.7 gezeigt angewandt. Dabei handelt es sich um einen Zusatz zum try-Befehl, der in runden Klammern nach dem try-Befehl steht.

```
1  Definition:
2  try ( /* try-with-resources-block : Öffnen von Ressourcen */)
3  {
4    // Programmcode, der eine Exception vom Typ ExceptionABC auslösen
       kann
5  }
6  catch ( ... )
7  {
8    // Programmcode zum Behandeln der Exception
9  }
10 // optional noch finally
11 finally
12 {
13  // Programmcode der bedingungslos ausgeführt wird
14 }
15
16 Beispiel:
17 void Methode(String path)
18     throws IOException {
19   try (BufferedReader br = new BufferedReader(new FileReader(path))) {
20     return br.readLine();
21   }
22 }
```

Bild 6.7: Try-with-resources Definition und Beispiel

In diesem try-with-resources-Block können beliebig viele Ressourcen – getrennt durch Semicolon - angelegt werden, die alle das Interface java.lang.AutoCloseable implementieren müssen. Die Reihenfolge, falls keine Exception geworfen wird, besteht aus dem Aufruf des try-with-resources, dem try und dem finally-Block. Wird eine Exception geworfen, so werden erst per java.lang.AutoCloseable Interface die angegebenen Ressourcen in umgekehrter Reihenfolge wie sie im try-with-resources-Block angegeben wurden, geschlossen, um anschließend den finally-Block auszuführen. Danach wird die Exception geworfen, die mit einem passenden catch Befehl aufgefangen werden kann. Treten eine oder mehrere Exceptions beim Schließen der Ressourcen auf, so werden diese unterdrückt, wenn auch im try-Block eine Exception geworfen wurde. Mit Throwable.getSurpressed kann Thread-sicher auf ein Array das alle unterdrückten Exceptions enthält zugegriffen werden. Hierbei ist zu beachten, dass beim rückwärtigen Abarbeiten des Stacktraces in jeder Aufrufhierachie weitere unterdrückte Exceptions hinzugefügt werden können.

6.4.2.5 Nested-exceptions

Erst in der JDK 1.4 (6. Februar 2002) wurde exception chaining hinzugefügt. [Wika] [Doca]

6.4.2.6 Multiple-exception-catching

In Java SE 7 wurden multiple-exceptions im catch-Befehl (multiple-exception-catching) und Exception-Typsicherheit im throw-Befehl hinzugefügt. Dabei ist das Exception-Object bei Verwendung von multiple-exceptions implizit final und kann nicht verändert werden. Die Exception-Typsicherheit ist nur garantiert, wenn dem Exception-Object kein anderer Wert im catch-Block zugewiesen wird. [Wika] [Doca]

6.4.2.7 Exception-An-Aufrufer Konstrukt

Exception-Weiterleitung-An-Aufrufer ist ein Vorgehen in dem eine Methode in Java ihre Verantwortlichkeit zur Behandlung von angegebenen Exceptiontypen an den Aufrufer der Methode delegieren kann. Tritt eine Exception in der Methode auf, wird die Ausführung des Kontrollflusses unterbrochen und die Exception weitergereicht. Das Sprachmittel dafür von Java ist throw. Es wird an die Methoden Signatur gefolgt von den betroffenen Exceptiontypen geschrieben. Die Notation hierzu ist in 6.8 zu sehen.

Das try-finally Konstrukt kann mit dem Exception-Weiterleitung-An-Aufrufer Konstrukt verbunden werden. Durch das Weiterleiten der Exception an den Aufrufer ist ein

```
1 void Methode( )
2   throws ExceptionA, ExceptionB
3 {
4 }
```

Bild 6.8: Methode, die Exceptions vom Typ ExceptionA und ExceptionB an den Aufrufer weiterleitet

catch nicht mehr notwendig. (Ein try-Block darf aber niemals alleine stehen!) So kann wenn ein Fehler auftritt noch bedingungslos ausstehender Programmcode abgearbeitet werden, während der Aufrufer die Ausnahmebehandlung durchführt. [Ull]

6.4.2.8 Default top-level Exceptionhandler

Dies kann in Java durch das Festlegen eines **uncaught exception handler**s erfolgen. [Ora]

6.4.2.9 Asynchrone Exceptions

Laut der Javaspezifikation [Ora] gibt es kein Sprachkonstrukt um asynchrone Exceptions zu werfen. Dies kann also nur durch normale Threadkommunikationsmittel erfolgen.

6.5 BPMN

In BPMN [OMG11] [Sil10] gibt es den Ereignistyp Error, das ein außergewöhnliches Signal ist, das innerhalb eines Prozesses generiert wird. Als intermediate event gibt es nur das interrupting boundary event. Ein Error kann nur von einem end event signalisiert werden.

Die Details eines Tasks in BPMN sind nach außen nicht sichtbar. Deshalb kann die Quelle eines Error-Ereignis nicht an der Task Boundary angegeben werden. Eine Kenntlichmachung ist nur durch eine Kovention in der Bennenung des Labels des Error-Ereignisses möglich.

In einem Prozess können mehrere Fehlerereignisse signalisiert werden und damit ein signalisiertes Fehlerereignis von dem richtigen interrupting boundarry event gefangen wird, werden das Error end event und das interrupting boundary event mittels einer Referenz zu einer common Error event definition verlinkt.

Die Weiterreichung ist einschrittig. Das Error-Ereignis kann nur von dem end event zu einem boundary event desselben Prozesses signalisiert werden.

Neben dem Fehler-Ereignis, wurde in BPM 2.0 das Escalations-Ereignis eingeführt, das dasselbe Verhalten wie das Fehler-Ereignis zeigt, aber mit den folgenden entscheidenden Ausnahmen. Das boundary event ist standardmäßig nicht-unterbrechend, es darf aber auch unterbrechend verwendet werden. Das Escalations-Ereignis impliziert keinen Fehler, sondern nur zusätzliche Berechnungen, die durchgeführt werden sollen. Aus diesem Grund können Escalation-Ereignisse von einem end event und von einem intermediate event signalisiert werden. Wie das Fehler-Ereignis, kann das Escalation-Ereignis nur von einem Task zu der Boundary des Task signalisiert werden.

6.5.1 Exception

6.5.1.1 Exceptiondefinition in Programmiersprache

Ein Error-Ereignis angewendet auf einem Service Task weist auf einen fehlerhaften Zustand hin, d.h. der Service konnte nicht normal abgeschlossen werden. Das System, das den Service bereitstellt ist entweder nicht verfügbar, der communication link ist nicht verfügbar oder es wurden nicht valide Daten an die Ausführung des Services übergeben. [Sil10]

6.5.1.2 Repräsentation

Eine Exception kann entweder als Error-Ereignis oder als Escalation-Ereignis dargestellt werden. [Sil10]

6.5.1.3 Exception Ableitungshierachie

In BPMN gibt es keine Ableitungshierachien. Allerdings ist dies in Implementierungen wie in Activiti, das in Java aufbaut, möglich. [Act]

6.5.1.4 Exception-Typsicherheit

6.5.1.5 Exception-driven design

In BPMN existiert kein Exception-driven design. Es ist nur eine Konvention, wird aber nicht gefordert.

6.5.1.6 Rückgabewerte als Fehlerindikator

In BPMN werden oft Fehler modelliert indem einem Prozess ein Gateway folgt, indem geprüft wird, ob ein Fehler aufgetreten ist. [Sil10]

6.5.1.7 Un-/checked Exceptions

Im Diagramm von BPMN müssen keine Zuordnungen von signalisierenden Ereignissen und fangenden Ereignissen bestimmt werden. Dies spricht dafür, dass auch unchecked Exceptions möglich sind. Werden mehrere Exceptions in einem Subprozess signalisiert, so findet eine Zuordnung im Diagramm mittels der Label statt. In einer Implementierung geschieht eine nicht im Diagramm sichtbare Zuordnung mittels der Error event definition. Dies spricht dafür, dass in einer Implementierung checked Exceptions vorausgesetzt werden. Im Business Process Manager von IBM [IBM] gibt ein BPEL Prozess nur checked Exceptions zurück.

6.5.1.8 Exception-surpressing

Existiert nicht. [OMG11]

6.5.2 Konstrukte / Pattern

6.5.2.1 exception-chaining

Existiert nicht. [OMG11]

6.5.2.2 Guarantee-execution-Block

Ist durch Pattern darstellbar. Nicht-unterbechendes Intermediate Escalation Event mit anschließendem Guarantee-Execution-Block. Allerdings ist die Semantik leicht unterschiedlich, da hier nicht der catch-Handler und danach der Guarantee-execution-Block ausgeführt wird, sondern gleichzeitig.

6.5.2.3 Try-succeeded-Block

Existiert nicht. [OMG11]

6.5.2.4 Try-with-resources-Block

Existiert nicht. [OMG11]

6.5.2.5 Nested-exceptions

Existiert nicht. [OMG11]

6.5.2.6 Multiple-exception-catching

Existiert nicht. [OMG11]

6.5.2.7 Exception-An-Aufrufer Konstrukt

Existiert nicht. [OMG11]

6.5.2.8 Default top-level Exceptionhandler

in BPEL wird eine Exception bis zum main process weitergereicht, dem ein Exceptionhandler zugewiesen werden kann.

6.5.2.9 Asynchrone Exceptions

Nicht-unterbrechende Eskalationen sind asynchrone Exceptions.

In seltenen Fällen kann zur Signalisierung von Exceptions auch auf die Ereignis-Typen Message und Signal zurückgegriffen werden. Message kann zwischen Pools signalisiert werden. Dabei ist zu beachten, dass Message aber nicht innerhalb eines Pools verwendet werden kann. Signal folgt dem Publish Subscribe Paradigma, und wird so mehreren Subscribern signalisiert.

7 Exception-Handling in Bytecode

Folgende Informationen stammen aus [Ven97a]. Die Abarbeitung des Programmcodes läuft in vielen Programmiersprachen wie beispielsweise Java oder Python über Bytecode. Im folgenden betrachten wir den Exception Handling Mechanismus auf der Java Bytecode-Ebene anhand einer Funktion „remainder“, die eine Integerdivision durchführt. Bei einer Division wird von der Divisior-methode von Java eine unchecked Exception vom Typ ArithmeticException geworfen. In der Methode „remainder“ wird diese Exception gefangen und in eine Exception vom Typ DivideByZeroException konvertiert und diese geworfen. In der Tabelle 7.1 ist die Methode „remainder“ auf der linken Seite zu sehen. In der Mitte steht der korrespondierende Bytecode und rechts ist eine Erläuterung zu jedem Bytecodefragment zu finden.

In einem Programmabschnitt folgt erst der normale Programmablauf, dahinter folgen die ”catch-Blöck(e)” in Bytecode. In dem Beispiel erstreckt sich der normale Code ohne Exceptionhandling vom PC (engl. Programm counter) Offset 0 bis einschließlich 3. Der try-Block wird durch den PC Offset 4 bis einschließlich 12 abgegrenzt. Für jede Methode, die Exceptions fängt wird neben dem Bytecode eine Exception-Tabelle generiert, die in der .class-Datei enthalten ist. In ihr steht pro Exception in jedem ”try-Block” ein 4-stelliger Eintrag (start, ende, pc_offset, exceptiontyp). Dabei handelt es sich um den Start- und End-PC-Offset des zu überwachenden try-Blocks, dem PC-Offset des catch-Blockes und einem konstanten Indexwert, der den Exceptiontyp repräsentiert. In dem Beispiel sieht die Exceptiontabelle aus wie in 7.2.

Bei Auftreten einer Exception wird die Exception-Tabelle vom ersten Eintrag beginnend abgearbeitet. Zunächst wird der from und to Bereich geprüft, passt dieser, wird der type untersucht, ob dieser dem Exceptiontyp oder einer Oberklasse entspricht. Wenn ja wird in den Exception-Handler an target gesprungen.

Im weiteren beziehe ich mich auf [Ven97b]. Im folgenden soll nun noch das finally-Konstrukt in Bytecode betrachtet werden. Der finally-Block wird bei jedem Austrittspunkt (return, break, continue) vom try-Block ausgeführt. In Bytecode wird ein finally-Block durch eine Miniatur-Subroutine realisiert. Zu dieser Subroutine wird mit dem Opcode jsr oder jsr_w gesprungen (jsr nimm 2-Byte, jsr_w 4-Byte Operanden entgegen). Im

Java	Zeile	Bytecode	Erklärung

`static int remainder(int dividend, int divisor)`
`throws DivideByZeroException`
`try {`			

`return dividend % divisor;`	0	iload_0	Lege Divisior auf Stack
	1	iload_1	Lege Dividend auf Stack
	2	irem	Hole Dividend, Divisor, lege Rest auf den Stack
	3	ireturn	Gibt obersten Wert des Stacks als Integer als Rückgabewert zurück
`}`			
`catch (ArithmeticException e){`	4	pop	Verwerfe obersten Wert vom Stack, da Referenz nicht verwendet wird
`throw new DivideByZeroException();`	5	new #5 <Class DivideByZeroException>	Erzeuge Referenz zu einer neuen Instanz von DivideByZeroException und lege diese auf den Stack
	8	dup	Dupliziere den obersten Wert auf den Stack
	9	invokenonvirtual #9 <Method DivideByZeroException.<init>()V>	Rufe den Konstruktor zur Initialisierung der Instanz auf. Oberster Stackeintrag wird als Argument genommen und vom Stack entfernt.
	12	athrow	Hole die Referenz (zu einem Throwable-Objekt) vom Stack und werfe die Exception
`}` `}`			

Tabelle 7.1: try & catch in Bytecode

from	to	target	type
0	4	4	<Class java.lang.ArithmeticException>

Tabelle 7.2: Exceptiontabelle zu 7.1

weiteren steht jsr synonym für jsr oder jsr_w. Wird die letzte Zeile des finally-Blocks erreicht, wird mit ret die Subroutine beendet und zur Zeile nach dem Aufruf von der Subroutine gesprungen. Wird aber eine Exception geworfen oder der finally-Block durch einen anderen Austrittspunkt (return, break, continue) beendet, so wird der Opcode ret nie ausgeführt. Diese Besonderheit ist zu beachten, da jsr auf den Stack den PC-Offset legt, der auf den jsr Opcode folgt. Da nun der finally-Block vorzeitig unterbrochen werden kann und diese Rücksprungadresse nicht von ret vom Stack geholt werden könnte, wird die Rücksprungadresse zu Beginn des finally-Blocks vom Stack geholt und in eine lokale Variable gespeichert. Wird der finally-Block vollständig abgearbeitet, so wird an ret der Wert der lokalen Variable gegeben und es wird zum PC-Offset nach dem jsr Opcode gesprungen. Zur Verdeutlichung soll das Beispiel dienen dass einen Boolean-Wert in eine 0 oder 1 Repräsentation konvertiert. Das Beispiel ist in 7.3 zu sehen.

Es sind drei Sprungbefehle zu dem finally-Block in dem Bytecode zu sehen. Zum einen an beiden Austrittspunkten im try-Block, zum anderen in dem vom Compiler generierten catch-Block. Selbst wenn eine Exception im try-Block geworfen wird, soll der finally-Block ausgeführt werden. Deshalb wird ein try-Block generiert, der den finally-Block aufruft bevor er die Exception weiter „nach oben“ wirft.

Die zu dem Code gehörige Exceptiontabelle sieht aus wie in 7.4:

Java	Zeile	Bytecode	Erklärung
`static int ConvertBooleanTo01 (boolean bVal){` `try {`			
`if (bVal){`	0 1	iload_0 ifeq 11	Lege Variable 0 auf den Stack Wenn gleich 0, springe zu Zeile 11
`return 1;`	4	iconst_1	Lege 1 auf den Stack
	5	istore_3	Speichere Integer in Variable 3
`\\impliziter Aufruf von finally`	6	jsr 24	Springe in Subroutine in Zeile 24
`führe ''return''aus`	9	iload_3	Lege Variable 3 („1“) auf den Stack
`}`	10	ireturn	Gebe den Wert auf dem Stack zurück
`return 0;`	11	iconst_0	Lege 0 auf den Stack
	12	istore_3	Speichere Integer in Variable 3
`\\impliziter Aufruf von finally`	13	jsr 24	Springe in Subroutine in Zeile 24
`führe ''return''aus`	16	iload_3	Lege Variable 3 („0“) auf den Stack
`}`	17	ireturn	Gebe den Wert auf dem Stack zurück
`// implizites catch(Throwable){}`	18	astore_1	Hole Referenz der geworfenen Exception von dem Stack und speichere sie in Variable 1
	19	jsr 24	Springe in Subroutine in Zeile 24
	22	aload_1	Lege Referenz der Exception aus Variable 1 auf den Stack
	23	athrow	Werfe Exception erneut

`Finally`
`{`
`System.out.println("Boolean␣Conversion␣Exception");`
	24	astore_2	Speichere Rücksprungadresse in Variable 2

	25	getstatic #8	Hole eine Referenz zu java.lang.System.out
	28	ldc #1	Lege den String ”Boolean Conversion Exception” vom konstanten Pool auf den Stack
	30	invokevirtual #7	Rufe System.out.println() auf
	33	ret 2	Springe zur Adresse aus Variable 2
`}` `}`			

Tabelle 7.3: finally in Bytecode

from	to	target	type
0	18	18	any

Tabelle 7.4: Exceptiontabelle zu 7.3

8 Leistungsanalyse von Exceptions in lokalen Systemen

8.1 Leseanleitung

Als Hauptquelle dienen die Aussagen von Christophe de Dinechin [dD00]. Nebenquellen werden an der Stelle angeben, an denen sich auf sie bezogen wird.

8.2 Einführung

Ein wichtiges Kriterium dafür, inwieweit Ausnahmebehandlung in einer Programmiersprache verwendet wird, ist der Einfluss auf die Leistung des Computerprogramms. Allein die Verwendung von Ausnahmebehandlung in einem Codeblock kann zu Leistungseinbußen führen, auch wenn während der gesamten Programmlaufzeit keine einzige Exception geworfen wird. Andere Implementierungsansätze für Ausnahmebehandlung können solche dramatischen Leistungseinbußen durch Einführung zusätzlicher Hilfsmittel verringern. Die Frage stellt sich also wie teuer ist es in einem Codeblock Exceptions zu verwenden? Dabei können zwei Dimensionen unterschieden werden: einerseits wie teuer ist es Exception Handling Code zu verwenden? Und zweitens: wie teuer ist es wenn eine Exception tatsächlich geworfen wird? Dies soll nun genauer in der Programmiersprache C++ untersucht werden, da C++ schon seit jeher als eine auf Leistung ausgelegte Programmiersprache gilt.

8.3 Vorbemerkung

Natürlich hat auch die Rechnerarchitektur Einfluss auf die Geschwindigkeit des Ausnahmebehandlungsmechanismus. Auf diese Dimension kann im Rahmen dieser Arbeit nicht eingegangen werden. In dem nächsten Abschnitt beziehen sich Vergleiche zwischen den Implementierungsansätzen auf die Rechnerarchitektur HP/Intel IA-64. Die Basis

für die Vergleiche ist der Programmcode ohne Ausnahmebehandlung, denn allein die Schachtelung eines Codeblocks in ein try-Block kann dessen Ausführung verlangsamen.

8.4 Portable Exception Handling / dynamic registration

8.4.1 Beschreibung

Der erste Ansatz Ausnahmebehandlung in C++ einzuführen war ein Ansatz basierend auf der Verwendung der zwei C Funktionen setjmp und longjmp. Die Funktion setjmp speichert einen Ausführungskontext in einer *jmp_buf* Struktur. Die Funktion longjmp kann später einen nicht-lokalen Sprung zurück zu der Funktion vornehmen, in der setjmp aufgerufen wurde, sofern diese Funktion nicht per return beendet worden ist. In dem "Portable Exception Handling" Ansatz wird eine try-Anweisung mit einem setjmp, und das Werfen einer Exception mit einem longjmp Befehl modelliert. Eine verkette Liste von *jmp_buf* Strukturen stellt eine Verschachtelung von try-Blöcken dar. Da lokale Objekte auf dem Stack angelegt werden, müssen diese zu einer Liste zu zerstörender Objekte hinzugefügt werden, damit diese bei Beenden des try-Blocks zerstört werden.

8.4.2 Leistungsanalyse

Bei dem Eintritt in den try-Block muss jedesmal die Funktion setjmp aufgerufen werden und die Listen für die *jmp_buf* Strukturen und die der zu zerstörenden Objekte muss verwaltet werden und untereinander konsistent sein. Außerdem müssen alle Variablen die in Registern und außerhalb eines try-Blocks stehen zu ihrem ursprünglichen Wert zurückgesetzt werden, wenn longjmp (entspricht dem Werfen einer Exception) aufgerufen wird. Dies erfordert dass alle Register bei Anruf von setjmp gespeichert werden. Alternativ können auch nur einzelne Variablen gespeichert werden. Beide Varianten sind kostspielig auf Architekturen mit vielen Registern. Nach Jonathan L. Schilling [Sch98] leidet dieser Ansatz unter signifikanten Ausführungszeiten und es ist schwieriger Threadsicherheit zu verwirklichen.

8.5 Table-driven Exception Handling

8.5.1 Beschreibung

Dieser Ansatz basiert auf der Verwendung von Tabellen, die der Compiler neben dem Maschinencode erzeugt und die von der C++ Laufzeitbibliothek zur Ausnahmebehandlung verwendet werden. Eine Tabelle ordnet den Programmzählern an Stellen, an denen eine Exception geworfen werden kann, eine Aktionstabelle zu. Eine Aktionstabelle beinhaltet alle Schritte die notwendig sind, um mit der Ausnahmebehandlung zu beginnen. Dazu zählt das Aufrufen von Destruktoren und das Anpassen des Stacks. Der letzte Schritt einer Aktionstabelle besteht darin, einen neuen Programmzähler zu berechnen und den Kontrollfluss in den Handler zu leiten.

8.5.2 Leistungsanalyse

Im Vergleich zum "Portable Exception Handling" Ansatz fallen keine Kosten für den Aufruf der setjmp Funktion bei jedem Betreten eines try-Blocks sowie das Verwalten einer verketteten Liste für verschachtelte try-Blöcke an. Nach [B.93] ist es das Ziel dieses Ansatzes keinen Overhead zu erzeugen bis zu dem Zeitpunkt, bei demeine Exception geworfen wird. Ein weiterer Vorteil im Gegensatz zu dem Portable Exception Handling Ansatz ist, dass die Verwendung hier threadsicher ist.

Nach Jonathan L. Schilling [Sch98] können Exceptions mehr Zeit zur Verarbeitung benötigen und die Exceptionhandling-Tabellen können viel Platz im Arbeitsspeicher einnehmen. Die größten Leistungseinbußen entstehen aber durch die Notwendigkeit die Assoziationen zwischen den Exceptionhandling-Tabellen und den Instruktionsbereichen aufrechtzuerhalten, sodass Optimierungen nur eingeschränkt möglich oder sogar unmöglich sind. Unmögliche Optimierungen resultieren aus folgenden Gründen:

1. Es können keine Codeverschiebungen und **instruction scheduling** durchgeführt werden, da diese dafür sorgen könnten dass die Lebenszeit von Objekten sich mit den Programmzähler-Marken überschneiden würden und somit auch die Programmzählermarken den Exceptiontabellen beschnitten werden würden.
2. Da die Tabellen auf Codestellen verweisen, sind alte Optimierungsverfahren, die allein den Maschinencode verändern, nicht mehr anwendbar. So müssen **back ends** (z.B. code generators und Optimierer) umgeschrieben werden, sodass catch-handler-Blöcke nicht als unereichbarer Code entfernt werden. Dazu brauchen die Optimierer

Pseudo Bedingungszweige nach jedem Aufruf innerhalb eines try-Blocks um den Kontrollfluss in den catch-Handler-Block zu leiten.

3. Desweiteren muss jedes Objekt, das über Destruktoren verfügt, seine Adresse in einer Tabelle speichern und als Konsequenz im Speicher bleiben. Implizit wird durch diese Forderung die Adresse der Obejekte preisgegeben und sie müssen vor jedem Aufruf im Arbeitsspeicher gesichert werden. Leistungseinbußen treten hierbei vor allem bei als inline deklarierten Klassenfunktionen auf, die nun nicht in Registern sondern im Arbeitsspeicher gehalten werden müssen, weil die Inhalte der Exceptionhandling-Tabellen erstellt werden, bevor der Compiler die Entscheidung trifft die Funktionen inline zu erstellen.
4. **Cache misses** aufgrund von selten ausgeführten Catchhandler-Code, der in der Mitte von dem ”happy path” Code sitzt.
5. Meistens ist nur das alte Stacklayout dass im **application binary interface** spezifiziert worden ist, verwendbar, weil die anderen Layouts nicht selbstbeschreibend sind wie von einem **stack frame** zu dem vorherigen gesprungen werden kann.
6. Die Laufzeitumgebung muss alle Variablen außerhalb eines try-Blockes auf einen korrekten Wert setzen, bevor ein catch-Block betreten wird und geht einher mit subtilen Leistungseinbußen 8.6.

Die Größe der Exception-Handling-Tabellen hängt von der Art der Verwendung von Objekten ab. Werden viele Objekte in kurzen Zeitabschnitten erzeugt und zerstört oder werden viele Funktionen mit wenig Codeumfang mit zu zerstörenden Objekten verwendet, so werden größerere Exception-Handling-Tabellen erzeugt.

8.5.3 Optimierungsansatz: Optimizing Away C++ Exception Handling

Dieser Ansatz ist die Arbeit von Jonathan L. Schilling [Sch98] und basiert auf das Wegoptimieren von nicht benötigten Tabellen im **Table-driven Exception Handling** Ansatz. Diese Möglichkeit ergibt sich daraus, dass selbst wenn eine Funktion keine expliziten Exceptionhandling-Konstrukte enthält Exceptionhandling-Tabellen erzeugt werden. Die Leistungszunahme kann Codegröße und Ausführungsgeschwindigkeit betreffend für die einzelne Funktion sehr groß sein, aber im Gesamtblick ist sie klein. Aber es können in diesem Ansatz durch das Spezifizieren von leeren Exceptions an Funktionen die Ausfüh-

rungszeiten signifikant verbessert werden, falls zusätzliche Optimierungen des Compilers eingeschaltet sind.

8.6 Leistungseinbußen durch Verwendung eines try-Blocks

8.6.1 Problembeschreibung

Die Verwendung eines try-Blocks schließt einige Optimierungsverfahren aus, da zwischen einem Funktionsaufruf oder throw-Befehl in einem try-Block und jedem assoziierten catch-Block implizite Sprünge existieren. Dies kann zur Ausdehnung der Lebenszeit einer Variablen führen und den Druck auf den Registerzuweiser erhöhen.

Dient eine Variable als Ausgabe der einen und Eingabe einer weiteren Funktion und wird diese Fließbandverarbeitung über eine Reihe von Funktionen hinfort geführt, so wird die Eingabe der vorherigen Funktion nicht mehr benötigt und kann verworfen werden. Meistens wird für den neuberechnenden Wert zur Leistungssteigerung mit dem Umgang von Registern ein anderes Register verwendet, und das alte wieder frei. In einem try-Block kann der Kontrollfluss aber jederzeit im try-Block unterbrochen werden und ein Zugriff auf die Teilergebnisse erfolgen. Dies hat zur Folge dass alle Werte in dasselbe Register gespeichert werden müssen.

In for-Schleifen können von Compiler zahlreiche Optimierungen vorgenommen werden. So können Additionen und Multiplikationen aufgeschoben werden bis die for-Schleife verlassen wird. Bei Verwendung eines try-Blocks kann die for-Schleife aber zu jederzeit unterbrochen werden und das bis dahin berechnete Teilergebnis wäre nicht korrekt. Deshalb muss in diesem Fall auf die Optimierungen verzichtet werden.

Bei Verwendung von for-Schleifen werden häufige Zugriffe auf Variablen optimiert in dem sie in Registern gespeichert werden. Wenn es sich dabei um Variablen eines Objektes handelt, so kann jederzeit der Destruktor des Objekts aufgerufen werden, der Zugriff auf die tatsächlichen Werte haben muss. Dies hat zur Folge, dass die Variablen bei jedem Schritt der for-Schleife in den Arbeitsspeicher zurückgeschrieben werden müssen.

8.6.2 Lösungsansatz mittels Rechnerarchitektur

Um die Kosten des Speicherns und Ladens von Registern abzufedern gibt es Lösungen wie die **Register Stack Engine** oder kurz RSE. Die RSE bietet einen allgemeingültigen

Weg an Register zu speichern und zu laden, wenn eine Funktion betreten oder verlassen wird. Das Verfahren basiert darauf, dass Register nicht notwendigerweise beim Betreten einer Funktion im Arbeitsspeicher gesichert werden, sondern so umbenannt werden, dass das erste Register für die aktuelle Prozedur immer r32 lautet. Nicht-gesicherte Register werden auf einen Registerstack gelegt und vorher gespeicherte Register werden vom Registerstack genommen und wiederhergestellt, wenn ein Funktionsaufruf zurückkehrt. Der Registerstack wird automatisch auf einem speziellen Stack gesichert, wenn freie Zyklen in der **load-store unit** auftreten. D.h. sobald ein Funktionsaufruf stattfindet, sind die Register des Aufrufers nicht mehr sichtbar und werden im Hintergrund gespeichert. So lange genügend freie Register verfügbar sind, werden Register des Aufrufers geladen, so dass sie schon zur Verfügung stehen, wenn die Funktion verlassen wird. Natürlich gibt es aber auch Fälle, in denen die load-store unit so lange Warten muss bis alle Register gesichert oder wiederhergestellt worden sind.

8.6.3 Lösungsansatz durch eine weitere Exceptiontabelle

Lee et. al [LYK+00] haben einen Ansatz entwickelt, der das Problem der unmöglichen Optimierungen, die im Table-Driven Ansatz 8.5.2 zurückbleiben und die allgemeinen Leistungseinbußen, die durch Verwendung eines try-Blocks entstehen 8.6, beseitigt.

Die Auswertung der SPEC Benchmarks, die einen repräsentativen Ausschnitt für Javaprogramme bieten, stellte sich heraus, dass nur in 3 von den insgesamt 8 Benchmarks, Exceptions signalisiert wurden. In den verbleibenden Benchmarks fiel auf, dass der Exceptiontyp eindeutig für die **point of raise** ist. Hauptsächlich liegt dies daran, dass Exceptions für den Kontrollfluss, und nicht für die Fehlerbehandlung genutzt werden.

8.6.3.1 Exception-Informations-Tabelle

Aus diesen Ergebnisse, schlagen Lee et. al vor, dass wegen dem seltenen Gebrauch von Exceptions, die Exceptionhandler nicht mit den anderen Teilen einer Methode in Bytecode übersetzt werden sollte. Die Vorteile der verzögerten Übersetzung der Exceptionhandler erst auf Nachfrage, ist dass JIT (just in time) Kompilierungszeit reduziert wird und der Kontrollfluss einer Methode vereinfacht wird. Die JIT Kompilierungszeit ist von der Größe des Bytecodes abhängig und dieser wird durch das Auslassen der Exceptionhandler reduziert.

Exceptionhandler werden mit Hilfe der Exceptiontabelle gefunden, die einer class Datei zugeordnet wird. Sie besteht aus Einträgen, die sich aus den Feldern start_pc.

end_pc, handler_pc und catch_type zusammensetzen. (siehe 7.2) Alle PCs (programm counter) sind Bytecode-Adressen. Nachdem eine Methode in nativen Code durch einen JIT Kompilierer übersetzt wurde, existieren nur noch native Adressen. Exceptionhandler müssen mittels nativen Adressen gefunden werden, wenn eine Exception auftritt. Werden die Adressbereiche einfach von Bytecodeadressen auf native Adressen abgebildert, indem die Codeblockstruktur übernommen wird, so können einige Codeoptimierungen nicht angewendet werden.

Durch Einführung einer neuen Tabelle, der **Exception-Informations-Tabelle**, die eine Hashtabelle ist, die Paare aus nativer Adresse und Bytecode Program Counter ält und alle potentielle Instruktionen, die Exceptions werfen können enthält, kann die ursprüngliche Struktur des Bytecodes verändert werden und so können agressive Optimierungen, die Code verschieben, angewendet werden.

8.6.3.2 Exceptionhandler-Voraussage

Häufig verwendete Exceptions werden meistens mittels eines throw-Befehls signalisiert. Dieser Befehl benötigt einen Exceptiontyp um einen geeigneten Exceptionhandler zu finden. An dem **point of raise** wird meistens eine neue Exceptioninstanz erstellt und somit ist der Exceptiontyp an dieser Stelle eindeutig. Wird eine Exception vom selben Typ an diesem **point of raise** signalisiert, so wird derselbe Exceptionhandler verwendet werden.

Somit kann anstelle der Suche, ein direkter Sprung zu dem Exceptionhandler unternommen werden. Mittels dieses Verfahrens kann die Suche bei einem zweiten Auftreten übersprungen werden. Es muss aber eine Überprüfung stattfinden, ob die Exception vom selben Typ ist und lokale Variablen, die im normalen Kontrollfluss vorhanden sind, können im Exceptionhandler benutzt werden.

8.6.3.3 Leistung

Durch diesen Ansatz können die SPEC Benchmarks um 4% und die Ausführungszeit durch Exceptionhandler-Voraussage um 11% gesteigert werden.

9 Exception Pattern

9.1 Exceptiontypen definieren

9.1.1 Namensgebung

Nach William Grosso [Groa] soll Name einer Exception das Problem schreiben, dass sie darstellt. Der Name dern Komponente, die die Exception wirft, sollte nicht im Namen vorkommen. Dies bringt zwei Vorteile. Zum einen ist der Name (Kodierung des Problems) und der unmittelbare stack frame ausreichend um das Problem zu lösen. Zweitens ist die Einordnung der Exceptions in eine Exceptionhierchie einfacher, da Ähnlichkeiten so betont werden.

9.1.2 Strukturierung

9.1.2.1 Exceptionhierachie

Nach William Grosso et al. [Grob] soll in einer exceptional condition bei der Entscheidung welcher Exceptiontyp signalisiert werden soll, die Auswahl danach getroffen werden, dass der Typ so spezifisch wie möglich sein soll mit der Einschränkung dass der Aufrufer noch den Sinn erkennen können soll.

9.1.2.1.1 Exceptiontypen für jeden Kontext

Nach William Grosso [Groa] hat in einem System, das aus Schichten aufgebaut ist, jede Schicht ihren individuellen Kontext und individuelle Domäne. Aus diesem Grund sollte jede Schicht Exceptions konvertieren.

9.1.2.1.2 Verfeinern der Typen

Laut Martin Pool [c2.c] sollte für jedes logische Untersystem ein eigener Exceptiontyp von der **BaseTypeException** abgeleitet werden. Während der Entwicklung der Systems wird mehr Information angesammelt, welche **unterschiedliche** Exceptiontypen benötigt

werden. Diese werden von der Untersystem-Basis-Exception des logischen Untersystems abgeleitet und verfeinern die Exceptionhierachie.

9.1.2.1.3 Homogenisierung von Exceptions

Nach Phil Goodwin [c2.b] ist dieses Pattern sinnvoll, wenn Methoden eine Liste führen müssen, welche Exceptions sie werfen können und sich diese Liste ändert. Findet eine Evolution dieser Exception statt, d.h. wird der Liste eine weitere Exception hinzugefügt, so muss jeder Aufrufer, diese Exception auch seiner Liste hinzufügen. Die Kette kann sich bis zum ersten Aufrufer des Programms fortsetzen. Um dieses angesprochene Problem zu umgehen, soll das Pattern **Verfeinern der Typen** verwendet werden. Außerhalb des Untersystems dürfen nur Exceptions signalisiert werden, die vom (abgeleiteten) Typ der Untersystem-Basis-Exception sind. Dadurch müssen wenn eine Methode den Exceptionbereich ihrer Signatur ändert, andere Methoden dies nicht.

9.2 Signalisieren von Exceptions

Nach Bill Venners [c2.a] sollte man sich an folgende Regeln halten. Wenn eine Methode sich in einem nicht-normalen Zustand befindet, und das Problem nicht lösen kann, so sollte eine Exception signalisiert werden.

Exceptions sollten nicht verwendet werden um auf Zustände hinzuweisen, die vernünftigerweise als Teil der normalen Funktionsausführung der Methode erwartet werden kann.

Wenn die Methode ihrer vertraglich festgelegte Aufgabe nicht erfüllen kann, muss eine Exception signalisiert werden.

Wenn eine Methode feststellt, dass die aufgerufene Methode ihre vertraglich festgelegte Verpflichtungen verletzt (bsp. inkorrekte Eingabedaten zurückgeben), muss eine Exception signalisiert werden.

Wenn eine Methode sich in einer exceptional condition befindet und der Aufrufer bewusst entscheiden soll wie diese behandelt werden muss, so sollte eine checked Exception signalisiert werden.

Für jede exceptional condition sollte eine Exception geworfen werden.

Nach William Grosso et al. [Grob] sollen keine ”stillen” (d.h. unchecked) Exceptions verwendet werden, um auf die throw clauses verzichten zu können. Exception müssen gut dokumentiert und nicht ignorierbar sein.

9.2.1 Aufräumarbeiten vor dem Signalisieren

Vor dem Signalisieren einer Exception müssen exklusiv genutzte Ressourcen freigegeben werden. Darüberhinaus sollte ein konsistenter Zustand wiederhergestellt werden. Dies ist vor allem für Objekte wichtig, die Methoden bereitstellen. Ein konsistenter Zustand des Objekts, der die Klasseninvarianten erfüllt, muss hergestellt werden, damit ein erneuter Methodenaufruf erfolgen kann. Vor dem Signalisieren von Exceptions sollte ein Zustand angestrebt werden, der die Invarianten, sowie Vor - und Nachbedingungen der Methoden erfüllt. [c2.d]

9.2.2 asynchrone Exceptions

Das Pattern **Exception Reporter** [Unb] Wenn eine Exception in einem Thread signalisiert und in einem anderen Thread - der den Kontext hat um eine Entscheidung beim Exceptionhandling treffen zu können - behandelt werden soll, so kann dies nicht durch das throw-Sprachkonstrukt realisiert werden. Eine Lösung ist es, Public-Subcribe zu verwenden. Statt dem throw-Befehl wird für jeden Subscriber eine spezielle Interface-Methode aufgerufen, die als Argument die Exception entgegennimmt. Die Subscriber können in ihrem Thread angemessen auf die Exception reagieren.

9.3 Exception Weiterreichung

9.3.1 Weterreichungsrichtlinien

Nach Phil Goodwin [Goob] besteht ein System aus mehreren Abstraktionsebenen. Die unteren Ebenen enthalten meistens keinen Code der fundamentale Entscheidungen fällt. Diese Eigenschaft liegt in den höheren Ebenen. Nur diese haben genügend Kontext um eine Entscheidung treffen zu können, wenn eine Exception auftritt. Aus diesem Grund sollten Exceptions nur auf den höheren Ebenen behandelt werden.

9.3.2 Einsatz von catch

Nach Phil Goodwin [Gooa] sollten catch Klauseln nur an Stellen verwendet werden, wo die exceptional condition behandelt werden kann. Ist dies nicht möglich so sollen Ressourcen wieder freigegeben und die Exception erneut signalisiert werden.

9.3.3 Nicht-behandelte Exception

Martin Pool [Poo] schlägt vor, in Java den Aufwand für sich noch ändernde throw Klauseln zu begrenzen, indem ein spezieller Exceptiontyp ”UnhandledException” eingeführt wird. Wird eine Exception von diesem Typ signalisiert, so wird in dem Konstruktor Code ausgeführt, der alle Werte der Exception auf der Konsole ausgibt. Diese Vorgehensweise hat den Vorteil, dass anfangs Zeit gespart wird, die Exceptiontyp RunTimeException von Java nicht missbraucht wird, und der Änderungsaufwand und die Abhängigkeiten unter den Methoden bei Änderung der throw Klausel nicht existieren. Die Spezifikation der throw Klausel kann herausgeschoben werden, bis genügend Information über das System und die Interaktion der Methoden gesammelt worden ist um einmalig die Spezifikation durchzuführen.

9.4 Exception-Safety Strong Guarantee

Nach David Abrahams [Abr00] wird die Exception-Safety Strong Guarantee erfüllt, wenn eine Operation entweder vollständig erfolgreich ist, oder wenn eine Exception geworfen wird, der Programmzustand nach Beendigung der Operation genau derselbe ist wie vor der Ausführung der Operation. Wenn Exceptions weitergereicht werden können subtile Änderungen des Programmzustandes Auswirkungen auf zukünftige Operationsausführungen haben. Wird eine Operation durch eine Exception abgebrochen, so sollte sich der Zustand von Objekten nicht ändern, die für den Aufrufer zugreifbar sind. Durch Transaktionen kann dies sichergestellt werden, ist aber inpraktikabel für general-purpose Programmiersprachen. Zudem können viele I/O Operationen nicht durch roll-back Strategie kompensiert werden.

Nach Lagorio und Servetto [LS10] kann die Exception-Safety Strong Guarantee durch ein Pattern in Java und jeder anderen Programmiersprache realisiert werden. Es basiert auf der copy-and-swap [Wik11] Technik von C++. Berechnungen werden in temporären Variablen durchgeführt. Erst am Ende der Operation wird die temporäre Berechnung mit dem tatsächlichen Ergebnis getauscht.

Lagorio und Servetto [LS10] haben dies dazu erweitert, dass eine Operation alles machen darf bis Daten geändert werden, die von dem Heap des Aufrufers erreichbar sind. Dazu werden **ro** (read-only) und **rw** (read-write) Zugriffsmodifizierer eingeführt. Diese sind auf Methoden und Referenzen anwendbar. ro erzwingt dass die mit ihnen behafteten Objekte nur lesende Aufrufe durchführen können. Dadurch kann eine Transitivität über

die Konstanz (Unveränderlichkeit) von Objekten aufgebaut werden. Durch statische Analyse kann festgestellt ob eine Anweisung eine Exception signalisieren darf oder nicht. Können durch statische Analyse die vorherigen Anweisungen kein Objekt verändert haben, so darf eine Anweisung folgen, die eine Exception wirft. Sobald in einer Codezeile ein Objekt geändert wurde, dass vom Aufrufer aus zugreifbar ist, verhindert das Typsystem von Lagorio und Servetto dass eine Anweisung folgen darf, die eine Exception werfen kann.

10 Welche Konzepte, Notationsmodelle, Referenzarchitekturen gibt es für verteilte Systeme

In verteilten Systemen herrschen meist komplexe asynchrone und interagierende Aktivitäten. Unter diesen Bedingungen wird Exceptionhandling sehr schwierig und ein Weg die Komplexität zu minimieren und das Exceptionhandling einfacher zu gestalten ist es die Interaktion und Kommunikation durch **Atomic-Action**s einzuschränken. Die meisten Exceptionhandlingansätze in verteilten Systemen beruhen auf atomic actions, aber es gibt keine Einstimmigkeit darüber wie Exceptions behandelt werden wenn asynchrone Aktivitäten auftreten.

Zunächst soll eine Klassifikation aufgestellt werden, welche die Dimensionen und Kriterien mit denen Exceptionhandling in verteilten Systemen realisierbar ist, aufzeigt. Nach der Erkenntnis welche Parameter einen Exceptionhandlingsystem in verteilten Systemen bestimmen, soll darauf eingegangen werden, wie diese Parameter eingestellt werden müssen um es in lose-verteilten Systemen nutzen zu können.

10.1 Vorwissen

10.1.1 Exceptionhandling in Webservices

Als Einstieg für Exceptionhandling in verteilten System, soll zunächst betrachtet werden, wie Exception in einer Server / Client Architektur behandelt werden.

10.1.1.0.4 Java Enterprise Java Beans

Nach den Ausführungen von Srikanth Shenoy [She] werden in Java Enterprise Java Beans (EJBs) Exceptions in drei Kategorien eingeteilt: JVM Exceptions, Application

Exceptions und System Exceptions. JVM Exceptions werden von der JVM signalisiert und zeigen einen schwerwiegenden, nicht behandelbaren Zustand an. Als Reaktion muss der Application Server gestoppt und das System neugestartet werden. Application Exceptions sind spezifische Exceptions, die von der Anwendung oder einer Drittanbieterbibliothek definiert worden sind. Es handelt sich meist um checked Exceptions und meistens an, dass eine Bedingung für die Geschäftslogik verletzt wurde. Der Aufrufer der EJB Methode kann die Exception behandeln. System Exceptions sind abgeleitete Exceptions von RuntimeException und somit unchecked Exceptions. Es handelt sich meist um Programmierfehler.

Nach Simons et al [SS04] handelt es sich bei EJB um ein Caller-Based Exceptionhandling. Ein Methodenaufruf auf einem Stub im Client führt zur Ausführung einer Methode auf dem Server. Wird innerhalb der Methode eine Exception signalisiert und kann diese nicht vom Server behandelt werden, so wird die Exception an den Aufrufer weitergeleitet. Diese verhält sich hier wie eine normale lokale Exception und wird mit den lokalen Exceptionhandlingmechanismen von Java behandelt.

Die EJB Spezifikation [Sak09] behandelt nur Application Exceptions und System Exceptions. Dabei finden alle Methodenaufrufe durch den EJB Container statt. Application Exceptions sind in der Methodensignatur des Remote-Interfaces deklariert und werden unverarbeitet an den Client übermittelt. Der Client hat die Möglichkeit eine Behebungsfunktion auszuwählen und zu entscheiden, ob ein Roll-Back für eine Transaktion durchgeführt werden soll. Application Exceptions sollen nur vom Client und von keiner anderen Instanz behandelt werden, da sie Geschäftslogik repräsentieren. Deswegen sollten dem Client auch keine Probleme auf der Systemebene mitgeteilt werden, da dies nicht zu den Application Exceptions gehört.

System Exceptions sind entweder checked oder unchecked Exceptions. Im Fall einer unchecked Exception wird automatisch ein Roll-back einer Transaktion durchgeführt. Der Container verpackt die unchecked Exception in eine RemoteException und signalisiert diese dem Client. Im Fall von checked Exceptions sollten alle Exceptions, die an den Client signalisiert werden sollen, in eine (abgeleitete) java.ejb.EJBException verpackt werden. Diese wird von dem EJB Container automatisch in eine RemoteException verpackt und dem Client signalisiert. Beim Auftreten einer System Exception nimmt der Container an, dass die Bean Instanz fehlerhaft ist. Dies hat zur Folge, dass diese dereferenziert und durch die Garbage Collection aufgeräumt wird. Bei Stateless Session Beans, die keinen Zustand über die Kommunikation zwischen Server und Client speichern, hat dies keine Auswirkungen. Für stateful session beans, die Informationen speichern, die sich während

der Kommunikation zwischen Server und Client ansammeln, hat dies die Folge, dass diese Informationen verloren gehen.

10.1.1.0.5 Exceptionhandling in SOAP Web Services mittels JAX-RPC

Server in SOAP Web Services [WB] übermitteln dem Client einen fehlerhaften Zustand durch einen SOAP fault. Ein SOAP fault besteht aus einem Fehlercode, einer für Menschen lesbare Zeichenkette und optional noch weiteren Details. In der JAX-RPC Spezifikation sind eine Vielzahl von Regeln definiert, wie eine Java Exception serverseitig auf einen SOAP fault und im Clientcode von einem SOAP fault zurück auf eine Java Exception gemappt werden soll.

JAX-RPC erfordert, dass alle remote methods in einem service endopoint interface Exceptions vom Typ java.rmi.RemoteException werfen. Dies ermöglicht es, dass Exceptions zurück zu dem Aufrufer weitergereicht werden können. Die Anwendung sollte keine RemoteException Objekte an den Client signalisieren, da unterschiedliche JAX-RPC Laufzeitumgebungen auf der Clientseite unterschiedliche Interpretationen für einen bestimmten SOAP fault haben kann und verschiedene RemoteExceptions generieren würden.

In JAX-RPC sollten explizit die Exceptions in den Schnittstellen angegeben werden, die signalisiert werden können. Dazu müssen zu einer **wsdl:operation** (die einer Java Methode entspricht) mehrere **wsdl:fault**s (die Java Exceptions entsprechen) zugeordnet werden, die zuerst definiert werden müssen. Jeder wsdl:fault wird durch mapping eine user exception als Teil des service endpoint interface zugeordnet.

Tritt eine Exception auf der Servserseite auf, so wird lokales Exceptionhandling auf dem Server betrieben. Ist dies nicht möglich, so kann falls der Client eine Entscheidung zum weiteren Vorgehen treffen muss, die Exception als SOAP fault zurückgegeben werden. Ist die Exception nicht vom Client behandelbar, so soll dem Client auch nur ein Fehlercode per SOAP fault übergeben werden.

10.1.1.0.6 Exceptionhandling in ASP.NET Web Services

Exceptions in ASP.NET Web Services werden vom Server an den Client signalisiert [MSDd], indem ein Objekt vom Typ SoapException angelegt und mit einer Fehlerbeschreibung, einem Fehlercode und einer Url als Kontext gefüllt wird. Durch den throw-Befehl wird diese Exception an den Client signalisiert.

Im Client [MSDc] muss der Aufruf einer remote methode in einem try-catch Block stehen. Signalisiert die Methode auf der Serverseite eine Exception, so wird diese lokal im Client signalisiert und der Kontrollfluss wird in den Client geleitet.

10.1.1.0.7 Beurteilung von Exceptionhandling in Web Services

Abschließend kann Exceptionhandling in den unterschiedlichen Web Services wie folgend betrachtet werden. Der Client ruft eine entfernte Methode auf dem Server auf. Der Methodenaufruf sieht aber im Clientcode wie ein lokaler Methodenaufruf aus. Tritt im Server eine exceptional condition auf so wird diese lokal behandelt. Ist dies nicht möglich wird die Exception dem Client signalisiert. Da der entfernte Methodenaufruf wie ein lokaler Aufruf implementiert ist, findet nun im Client auch lokales Exceptionhandling auf. Die neue Dimension des verteilten Exceptionhandling besteht darin, die Exception an ein anderes System zu übertragen. Dies kann entweder durch Mapping stattfinden, indem ein Exceptionobjekt auf einen Fehlercode im Server und im Client vom Fehlercode wieder auf ein Exceptionobjekt gemappt wird. Andererseits kann das Exceptionobjekt auch serialisiert übertragen werden. Dadurch besteht zwischen Server und Klient hohe Kopplung. Im weiteren sollen Exceptionmodelle betrachtet werden, die hohe Kopplung unterstützen.

10.1.2 Agenten

Zum Verständnis des nachfolgenden Abschnittes ist der Begriff des Agents wichtig. In der Literatur trifft man diesen Begriff ständig an, wenn Exceptionhandling und geringe Kopplung in einem Atemzug genannt werden. Deswegen soll kurz erläutet werden, was unter einem Agenten zu verstehen ist. Nach Danny B. Lange [Lan98] reduzieren mobile Agenten den Netzwerkverkehr und Netzwerklatenzen. Ein Agent ist aus Sicht eines Endbenutzers, ein Programm, das Leute assistiert und in ihrem Auftrag handelt. Aus Sicht eines Systems, ist ein Agent ein Softwareobjekt, das:

- sich in einer Laufzeitumgebung befindet
- reaktiv ist, d.h. Änderungen in der Umgebung wahrnimmt und adäquat darauf reagiert
- autonom ist, d.h. Kontrolle über seine eigenen Aktionen hat
- ziel-orientiert ist, d.h. proaktiv ist
- zeitkontinuierlich ist, d.h. es wird kontinuierlich ausgeführt

Der Zustand eines Agenten umfasst alle Attributwerte, die benötigt werden um die Ausführung des Agenten zu stoppen und in einer anderen Laufzeitumgebung fortfahren lassen zu können.

10.2 Klassifikation

Zunächst soll wieder eine Klassifikation aufgestellt werden, um die Exceptionhandling-modelle in verteilten System systematisieren zu können. Die Klassifikation beinhaltet als erstes eine Betrachtung der Bedeutung von Exceptions in verteilten Systemen und inwiefern sich diese von den in lokalen Systemen unterscheidet. Im weiteren werden zusätzliche Dimensionen eingeführt, die durch die horizontale Erweiterung des lokalen Systems zu verteilten Systemen, Einfluss auf das Exceptionhandling haben und im weiteren berücksichtigt werden sollen. Anschließend werden die Phasen Signalisieren, Weiterreichen und Behandeln von Exceptions betrachtet.

10.2.1 Exceptiondefinition in verteilten Systemen

Exceptions erhalten, wenn sie von dem lokalen Anwendungsbereich herausgelöst werden und in verteilte Systeme gebracht werden, neue Bedeutungen und Eigenschaften, aber auch Umdeutungen.

Beim lokalen Exceptionhandling ist eine Exception vollständig bestimmt wenn Bedingungen im Aufrufkontext einer Operation verletzt werden. In einem Agentensystem obliegt es aber nach Eric Platon [PSH07] jedem Agenten selbstständig ob er ein Ergebnis eines Operationaufrufs als normal oder als Exception wertet. Je nach dem Kontext in dem sich ein Agent befindet wird ein Ergebnis zu einem anderen Exceptiontyp aufgelöst. Nach Dellarocas [KD99] ist es fraglich, ob Sprachkonstrukte in und für Agentensysteme, ausreichend sind. Oft schließt die Bedeutung von Exceptions in Agentensystemen nicht Autonomität und soziale Beziehungen mit ein, die die Ursache von Exceptions sein können. Nach Tripathi et al. [TM01] können **exceptional conditions** aus denselben Gründen wie beim lokalen Exceptionhandling auftreten. Hinzu kommen Gründe die sich aus den Eigenschaften der verteilten Systemen ergeben. Dazu zählen vor allem die Bereiche Sicherheit und Kommunikation. Das Signalisieren von Exceptions zwingt, nach Eric Platon, einem Event die Bedeutung einer exceptional condition auf. Damit zwingt der Dienstanbieter, den Aufrufer dazu dass Ergebnis als Exception zu handhaben. Jeder Agent sollte aber selber entscheiden können, ob dies für ihn eine exceptional condition ist

oder nicht. Nach Goodenough [Goo75] können Exceptions als **Opportunities** verstanden werden. Eric Platon [PHS06] [Pla06] zeigt dass in MAS Exceptions zu einem großen Teil als Opportunities benutzt werden. Dadurch bietet das Umgebungssystem den Agenten erweiterte Funktionalitäten an, die zur Leistungssteigerung und Flexibilität des Systems beitragen. Nach Klein et al. [KRAD03] muss eine Exception dahingehend erweitert werden, dass jegliche Abweichung von dem "idealen" Multi-Agent-System verhalten, eine Exception darstellt. Damit erlangt die Exception eine neue Dimension und ist losgelöst von der Vorstellung, dass sie nur auf einen Thread beschränkt ist.

Eine **agent exception** ist die Auswertung von einem Agenten, dass ein erhaltenes Event außergewöhnlich bzw. unerwartet ist. Damit ist die Quelle der Exception nicht wie im klassischen Sinn die Exceptioninstanz / das Event, sondern der Agent selbst. Dadurch wird die Autonomität des Agenten bewahrt.

Eine Exception macht nur im Kontext des Agenten Sinn, obwohl die Semantik des Events in der Quelle enkodiert wird, hängt der außergewöhnliche Charakter des Events von der Sichtweise des Agenten ab. Durch diese Bedeutung der Exception kann lose Kopplung zwischen Agenten verwirklicht werden, und die Kopplung der Agenten wird durch Exceptions nicht erhöht.

Desweiteren sind nicht alle Exceptions von grundauf mit Fehlern gleichzusetzen, sondern sind auch dazu da um neue Lösungsmöglichkeiten oder Alternativen zu beschreiben, worauf schon Goodenough hinwies.

10.2.2 Exceptionumfeld

Das Exceptionumfeld beschreibt Umgebung, Bedingungen, Einschränkungen, Interaktionspartner, Kommunikationsmittel, Techniken und Strategien, die Einfluss auf den Entwurf der Exceptionsignalisierungsmechanismus haben, und den Raum der Möglichkeiten einschränken wie Exceptions durch Exceptionhandler behandelt werden können. So nehmen gemeinsam verwendete Resourcen oder Aufteilung der Arbeitslast eines Algorithmus auf mehrere Teilnehmer, Einfluss auf den Spielraum signalisierende Exceptions behandeln zu können. Dazu zählt auch, dass jeder Teilnehmer alleine die signalisierte Exception unabhängig von der Absprache mit einem anderen Teilnehmer behandeln kann, oder ob ein gemeinsames Exceptionhandling nötig ist.

10.2.2.1 Art der Interaktion

Nach Romanovsky et al. [RK01] kann die Art der Interaktion mehrere Teilnehmer wie folgend unterschieden werden.

disjoint concurrency wenn nebenläufige Aktivitäten keine gemeinsame Ressourcen teilen und auch keine Synchronisation untereinander durchführen.

competitive concurrency wenn verschiedene nebenläufige Aktivitäten gemeinsame Ressourcen verwenden

cooperative concurrency wenn verschiedene nebenläufige Entitäten in Zusammenschluss eine Ausgabe produzieren

Ein verteiltes System kann aus mehreren **Komponenten** bestehen. Wenn eine Exception in einer dieser Komponenten auftritt müssen neben dem Anwender der Komponente auch alle anderen Komponenten informiert werden, die an einer **kooperativen Berechnung** beteiligt sind, um eine **koordinierte recovery activity** einleiten zu können. Dabei ist zu beachten dass unterschiedliche Komponenten Exceptions unterschiedlicher Exceptiontypen werfen können.

10.2.2.2 Integration von Exceptionhandling in verteilten Systemen

Nach Campéas et al. [CDUV05] ist Exceptionhandling ein low-level Konstrukt dass orthogonal bleiben muss, zugleich aber kompatibel zu transaktionalen Systemen sein muss.

10.2.2.3 Unhandled Exceptions

Unbehandelte Exceptions sind ein Phänomen dass nur in den wenigsten Exceptionhandlingsystem ausgeschlossen werden kann. Sicher können sie nur durch Verifikation ausgeschlossen werden. Aber der steile Anstieg der Anzahl der Verzweigungspunkte in einem Programm, verhindert die Durchführung von Verifikation. In verteilten Systemen, ist der Anstieg der Verzweigungspunkte noch steiler, da Exceptions von anderen Systemen zusätzlich zu denen des betrachteten Systems signalisiert werden können. Hinzu kommen maschinenunabhängige Programme wie beispielsweise Javaprogramme, für die keine Garantie gegeben werden kann welche Exceptiontypen von der Laufzeitumgebung signalisiert werden können.

Ein generischer Exceptionhandler behebt das Problem nicht, da die typische Aktion auf eine **unhandled Exception** die Beendigung des Prozesses ist. Dass die Exception

nicht erwartet wurde impliziert, dass das Programm mit keiner Behebungsprozedur ausgestattet wurde um das Problem zu behandeln.

10.2.2.4 Exception Handling Methodology

Viele Forscher teilen die Meinung, dass die Komplexität von Exceptionhandling eine Trennung von Exceptionhandling und dem restlichen Kontrollfluss des Programms erfordert. Es wurde ausreichend Forschung betrieben, wie der Exceptionhandler von dem Programmfluss getrennt werden soll, und diese Thematik wird als vollständig verstanden betrachetet. Das Trennen des Exceptionhandling im Gesamten von dem Rest des Programmes zu trennen ist aber noch nicht gut verstanden.

Bisher werden zwei Methoden verfolgt um die Trennung zu erreichen. Zum einen explizite Exceptionhandler- Koordination, und zum andern **Exception Handling Patterns**. Explizite Exceptionhandler erreichen die Trennung der Exceptionhandler von dem restlichen Kontrollfluss des Programms. Sie trennen das Exceptionhandling aber nicht von dem Programm. Folglich muss jedes Programm sein eigenes Exceptionhandling bereitstellen. Dies ist eine ”brute force Methode’, die fehleranfällig, wiederholend, und nicht leicht erweiterbar ist.

Eine andere Methode, die eine solche angestrebte Trennung ermöglicht, sind Exception Handling Patterns. Handler fragen eine Wissensdatenbestand ab um ein Pattern zu bestimmen, das zu der signalisierten Exception passt, und erhalten als Ergebnis eine geeignete Exceptionhandler-Aktion.

10.2.2.5 Art der Architektur

Open multi-agent systems [SCG07], kurz MAS, sind dezentralisierte und hochverteilte Systeme, die aus einer Vielzahl von loose-gekoppelten autonomen Agenten bestehen. ”Open” bedeutet dass unabhängig voneinander entwickelte autonome Agenten miteinander interagieren um ihre Ziele zu erreichen.

10.2.3 physikalische Verteilung

In einem verteilten System kann jeder **node** einen eigenen Speicher besitzen. **software segments** die auf verschiedenen **nodes** ausgeführt werden liegen in **disjoint address spaces**. In diesem Fall muss die Kommunikation durch **message passing** (den Austausch von Nachrichten) erfolgen. Meistens steht im Verhältnis dazu nur eine geringe Bandbreite zur Verfügung. Die Übertragungszeit und die Verzögerung sind deshalb eine entscheidende

Variable die bei dem Entwurf eines Exceptionhandlingmechanismus Beachtung finden muss.

10.2.4 Erkennen von exceptional conditions

In lokalen Systemen, findet die Erkennung einer exceptional condition, meist bei der Überprüfung von Vorbedingungen bei Funktionseintritt oder vor dem Aufruf einer Funktion oder der Ausführung eines Algorithmus statt. In verteilten Systemen ist der komplette Zustand einer Softwareanwendung einem Teilnehmer alleine meist nicht bekannt. Auch kann auf Überlegungen, ob die exceptional condition Einfluss auf andere Systeme hat, in lokalen System verzichtet werden.

Shah et al. [SCG07] fordern, dass in verteilten Systemen (Open-MAS) eine Unterscheidung auf den drei Ebenenen: Umgebung, Wissen und soziale Ebene möglich sein soll.

Umgebung-Exceptions sind Exceptions, die innerhalb der internen Umgebung eines Agenten und seinen zugeordneten Softwarekomponenten auftreten

wissensbasierte Exceptions sind Exceptions, die aus einer falschen Auswahl von Aktionen enstehen, weil das Wissen des Agenten über die Umgebung veraltet ist.

soziale Exceptions sind Exceptions, die sich aus der Fehlfunktion von einem Kommunikationskanal, Abhängigkeiten von Agenten und organisatorischen Beziehungen ergeben.

Der ursprüngliche Grund muss identifizierbar sein. Die Autonomie der Agenten muss unangetastet bleiben. Folglich muss der Diagnosemechanismus auf nicht-invasive Weise funktionieren, darf aber Zustandsinformationen der Agenten erfragen. Die Arbeitslast um Exceptions zu identifizieren sollte für den Agenten auf ein Minimum beschränkt sein. Die zusätzliche Funktionalität für die Exceptionsdiagnose sollte getrennt von den problemlösenden Funktionalitäten des Agenten zur besseren Wartbarkeit gehalten werden.

Nach Shah et al. [SCG07] gibt es zwei Verfahrensgruppen, wie Runtime-Aktivitäten überwacht und exceptional conditions erkannt werden können. Zum einen durch die **externe Exception-Diagnose**. Hier werden neben den vorhandenen Agenten, die als **problemlösende Agenten** bezeichnet werden, eine weitere Kategorie von Agenten eingeführt. Diese Agenten, auch **sentinel agents** genannt, werden verwendet um **problemlösende Agenten** zu beobachten. Dazu wird vorausgesetzt dass die sentinel agents, selber nicht versagen können und sie mit Wissen ausgestattet wurden um beobachtbare abnormale Situationen zu erkennen. Zum anderen die **interne Exception-Diagnose** oder

Introspektion genannnt, die es dem Agenten ermöglichen sein eigenes Laufzeitverhalten zu beobachten und Versagen zu erkennen.

Es gibt einige Fehlermodelle, die viele Eigenschaften gemeinsam haben. Miller et al. [PE02] haben ein standard fault model herausgearbeitet das repräsentativen Charakter hat. Dieses definiert die folgenden fehlerhaften Zustände.

Crash Ein Teilnehmer hat versagt oder wurde angehalten. Es gibt drei Varianten. **Fail-salient** bedeutet wenn andere Teilnehmer das Versagen nicht bemerken. **Fail-stop** heißt dass andere das Versagen mitbekommen. **Fail-safe** tritt auf wenn ein Teilnehmer willkürliches Verhalten zeigt bevor er angehalten wird, aber keine Auswirkungen auf andere Teilnehmer hat.

Omission Ein Teilnehmer versagt weil er zufällig eine Nachricht nicht empfängt oder nicht sendet. Ein Nachrichtenkanal oder Prozess kann versagt haben und solche Versagen können sporadisch auftreten.

Byzantine Ein Teilnehmer kann beliebig inkorrektes Verhalten zeigen das Auswirkungen auf andere Teilnehmer haben könnte.

Timing Ein Teilnehmer zeigt korrektes Verhalten, aber er hat eine Zeitschranke überschritten. Dies kann aus Nachrichtenübertragungsverzögerungen, Überlastungssituationen oder Einplanungsverzögerungen liegen.

Response Ein für den Teilnehmer lokaler Fehler ist aufgetreten.

Der Zweck eines fault model ist es fehlerhafte Zustände zu identifizieren, dass von einem Exception Handling System erwartet wird, diese behandeln zu können. Das standard fault model kann nicht einfach die aufgelisteten fehlerhaften Zustände identifizieren. So kann nicht zwischen einem Crash, bei dem ein Teilnehmer schließlich anhält unterschieden werden, und wenn ein Teilnehmer nicht anhält (fail-partition). Das standard fault model kann auch nicht unterscheiden ob zwei gleichzeitig auftretende Exceptions Bezug zueinander haben oder nicht.

10.2.5 Signalisieren von Exceptions

Durch das Signalisieren einer Exception wird ein außergewöhnlicher Zustand signalisiert. Dies kann lokal beschränkt oder global erfolgen. Lokale Exceptions sind auf den Host beschränkt, indem sie ausgelöst wurden. Globale Exceptions werden nicht nur einem Teilnehmer, sondern mehreren Teilnehmern signalisiert, die entweder einzeln oder zusammen

die Behandlung der Exception vornehmen. Es gilt jedoch zu beachten, dass Exceptions innerhalb eines Kontextes signalisiert werden. Dieser Kontext ist der Programmzustand in dem die Bedeutung der Exception kodiert ist. Wenn die Exception und ihre Daten auf einen gültigen Programmzustand bzw. auf eine aktuelle Operation verweist, so handelt es sich um eine **contextual** Exception [PE02], ansonsten um eine **non-contextual** Exception. In sequentiellen Programmen ist laut dieser Definition jede Exception am **point of raise** eine **contextual** Exception. Bei nebenläufigem Exception Handling kann eine Exception von einem Teilnehmer an einen anderen Teilnehmer signalisiert werden, also von einer **source execution** an eine **faulting execution** signalisiert werden. Diese Exception wird **non-contextual** sein, außer die Teilnehmer sind synchron in ihrer Nachrichtenkommunikation sowie auch in der Schrittfolge ihrer Prozessverarbeitung.

Nach Tazuneki[TY00] ist ein Exceptionkontext ein Bereich, in dem dieselbe Exception auf dieselbe Weise behandelt wird.

10.2.6 Weiterreichen von Exceptions

Die Art wie Exceptions weitergereicht werden ist in verteilten Systemen multidimensional. Exceptions können wie im lokalen System, vertikal weitergereicht werden und entscheiden, wo der Kontrollfluss fortgeführt werden soll, nachdem die Exception behandelt worden ist. Dies wird durch den Weiterreichungsmechanismus entschieden. Die zusätzliche Dimension ist, dass Exceptions, falls erforderlich auch horizontal weitergereicht werden, und somit auch in anderen Systemen signalisiert werden. Als zweites muss noch bestimmt werden wie der Pfad gebildet wird, entlang dessen die Exception weitergereicht wird, bis eine passende **handler clause** gefunden wird. Diesen gibt das Weiterreichungsmodell vor. Zuletzt folgt die Auswahl des geeignetesten Exceptionhandler aus der gefundenen handler clause.

10.2.6.1 Weiterreichungsmechanismus / Kontrollfluss in der Quelle

Der Weiterreichungsmechanismus bestimmt in der Quelle welchen Weg der Kontrollfluss nach Auftreten einer Exception in der Quelle geht und kann in eine lokale und in eine globale Komponente eingeteilt werden. Auf der lokalen Ebene gibt es die drei Möglichkeiten Werfen, Wiederaufnehmen und beide Verfahren zusammen. Aus Sichtweise der globalen Ebene kann entschieden werden, die Exception nur lokal zu behandeln, oder die Exception an eine oder mehrere Teilnehmer zu signalisieren. In dem Fall dass die

signalisierte Exception an mehrere Teilnehmer signalisiert wurde, so gibt es mehr als eine Quelle und in jedem Teilnehmer läuft ein Weiterreichungsmechanismus ab.

10.2.6.1.1 Werfen / throw

Bei **Werfen** wird nach der Behandlung der Exception nicht zum **point of raise** zurückgesprungen. Auf der Suche nach dem Handler wird der Stack abgearbeitet. Dabei werden die einzelnen stack frames entfernt (**stack unwinding**) bis ein passender Handler gefunden worden ist. Wurde die signalisierte Exception global signalisiert, so kann unter Umständen lokal stack unwinding nicht in beliebig Tiefe stattfinden. Dies trifft insbesondere für cooperative conccurency zu. Stattdessen müssen die Exceptionkontexte in allen von derselben Exception betroffenen Teilnehmern synchronisiert werden.

10.2.6.1.2 Wiederaufnehmen / resume

Bei **Wiederaufnehmen** wird mit der Ausführung am **point of raise** fortgefahren. Diese Variante eignet sich meist nur bei disjoint concurrency.

10.2.6.1.3 globales Signalisieren

Globales Exception Handling [PE02] ist erforderlich, wenn die Exception Handling Aktion über den Einflussbereich eines Teilnehmer hinaus geht, da auch andere Teilnehmer von der Exception betroffen sind, oder ein Teilnehmer keine Entscheidung aufgrund ungenügender Informationen treffen kann, weil der lokale Kontext unzureichend ist. Durch Werfen oder Wiederaufnehmen in einem lokalen System, können verschachtelte try-Blocks durchschritten werden. Dies entspricht einem Durchwandern unterschiedlicher Kontexte. Es kann der Fall eintreten, dass eine Exception in einem lokalen Kontext signalisiert wurde, und da kein passender Exceptionhandler gefunden wurde, oder weil zusätzliche Exceptions signalisiert wurden, ein Handler ausgeführt wird, der die Exception nicht behandelt, aber den Befehl enthält, die Exception global zu signalisieren.

Als Beispiel um den Vorteil eines Global Exception Handling zu verdeutlichen kann hier ein deadlock in einem System dienen. Tritt ein Deadlock auf und jeder Teilnehmer führt nur lokales Exception Handling aus, so ist ungewiss ob nur ein Teilnehmer sich dieses Problems gewiss ist, oder auch andere Teilnehmer. Nun kann auf lokaler Ebene nur jeder wissende Teilnehmer seine Ressourcen freigeben. Bei dem nächsten Versuch wieder die Ressourcen anzufordern wird der nächste deadlock aber nicht lange auf sich warten lassen.

10.2.6.2 verteiltes Weiterreichungsmodell / Exceptionhandler-Suchpfad

Das Weiterreichungsmodell legt fest auf welche Weise nach einem Handler gesucht wird, genauer gesagt wird durch das Modell die Reihenfolge festgelegt nach der die **handler clauses** durchsucht werden. Es entscheidet aber nicht welcher Handler in einer handler clause ausgewählt wird. Im Fall von disjoint concurrency kann auf das lokale Weiterreichungsmodell zurück gegriffen werden.

Xu, Romanosvky und Randell [XRR98] unterscheiden dazu die führungsrollenbestimmte und die rollenbestimmte Exception-Weiterreichung.

führungsrollenbestimmte Exception-Weiterreichung Die erste Variante ist dass eine **Führungsrolle** die Verantwortung trägt, eine vereinbarte Exception an den umschließenden gemeinsamen Exceptionkontext zu signalisieren. Es wird ein passender Exceptionhandler im Kontext der Führungsrolle gesucht.

rollenbestimmte Exception-Weiterreichung Die zweite Variante besteht darin, dass jede Rolle verantwortlich ist seine eigene Exception an den umschließenden Exceptionkontext zu signalisieren. Der Exceptiontyp der Exception, die diese Rollen signalisieren sollten im Allgemeinen gleich sein, aber können sich unterscheiden. Um gleichzeitig auftretende Exceptions behandeln zu können, werden Exceptions die gleichzeitig in einem gemeinsamen verschachtelten Exceptionkontext signalisiert werden, so behandelt, als ob sie gleichzeit in dem umschließenden gemeinsamen Exceptionkontext signalisiert worden wären.

Mostinckx et al [MDB+06] unterscheiden vier Ansätze.

asynchrone Exception Weiterreichung Bei Verwendung des try-catch Konstrukts, könnte durch asynchrone Kommunikation der aufrufende Prozess schon den Kontext des try-Blocks verlassen haben bevor eine Exception von dem aufgerufenen Prozess weitergereicht würde. Die Schwierigkeit besteht darin Exceptions, die in nebenläufigen Prozessen signalisiert wurden, in dem richtigen Kontext zu fangen.

concerted exceptions Asynchrone Kommunikation und die Definierung eines gemeinsamen Kontextes führt dazu, dass zeitgleich mehrere Exception signalisiert werden können. Eine bestimmte Kombination von aufgetretenen Exceptions kann ein konkretes Problem beschreiben dass zu einer concerted exception zusammengefasst werden kann. Alle Teilnehmer eines gemeinsamen Zusammenschlusses führen dann einen Exception-Handler aus, der zu der concerted exception passt.

Collaborative Exception Handling Da auch nach Verbindungsabbrüchen Prozesse weiterarbeiten können sollen, arbeiten diese mit optimistischen Annahmen. Tritt eine Exception auf, werden diese Annahmen wahrscheinlich verletzt und somit müssen alle Teilnehmer des gemeinsamen Zusammenschlusses Exceptionhandling durchführen.

loosely-coupled Exception Handling Die Autonomie jedes Teilnehmer muss gewährleistet bleiben, und so muss Exceptionhandling ohne zentralen Knoten auskommen. Um Prozesse nicht undefiniert lange auf einen Kommunikationspartner warten zu lassen, müssen nach Mostinckx et al [MDB+06] von der Umgebung langanhaltende Verbindungsabbrüche entdeckt und mittels Exceptions an das Programm signalisiert werden.

10.2.6.3 Exception-Auflösung

In einem nebenläufigen (nicht-verteilten bzw. verteilten) System können mehrere Teilnehmer zeitgleicht Exceptions signalisieren. Wenn die Exceptions Bezug zueinander haben oder sich gegenseitig bedingen, entweder aufgrund desselben fehlerhaften Zustandes oder eines kaskadierenden Effekts, so handelt es sich um **Causal Exceptions**. Ansonsten sind die Exceptions bezugslos oder unabhängig voneinander und werden als **non-causal** Exceptions bezeichnet.

Die Annahme bei **Exception-Auflösung** oder **exception resolution** ist, dass alle Exceptions **Causal Exceptions** sind. Dies ist gegeben, wenn die Exceptions im selben Kontext signalisiert worden sind. Je breiter der Kontext ist, desto unspezifischer sind die Exceptiondaten, und umso schwieriger ist es Exceptionhandling durchzuführen. Laut Romanovsky et al. [PE02] haben existierende Exceptionhandlingsysteme eine vage Anschauung was ein Kontext ist und daher ist es schwierig Kausalität zu bestimmen, da der Kontext breit sein kann. So kann die Unterscheidung in Subsysteme schwerfallen. Nebenläufig signalisierte Exceptions können nicht separat behandelt werden, da sie sich anscheinend im selben Kontext befinden. Eine adäquate Ausnahmebehandlung kann nicht gefunden werden, und so muss der Prozess meistens angehalten werden. So ist es häufig der Fall, dass es das Standardvorgehen ist, während der Behandlung eines fehlerhaften Zustandes, den Prozess abzubrechen oder das System anzuhalten. Dies liegt teilweise daran, dass es programmiertechnisch schwer ist festzustellen, ob die zweite Exception in Bezug zu der ersten steht, oder unabhängig von ihr behandelt werden kann.

Zur Auflösung der Exceptions kann zwischen den folgenden Ansätzen unterschieden werden. [XRR98]

Priorisierung Wenn zeitgleich mehrere Exceptions signalisiert werden, so können die Exceptions aufgelöst werden, indem sie priorisiert werden. Bei diesem Ansatz wird angenommen, dass die erste signalisierte asynchrone Exception, die exceptional condition schon sehr gut beschreibt, und es in der Regel nicht notwendig ist, die anderen asynchronen Exceptions, die in Folge der ersten signalisiert werden, zu betrachten. Auflösung findet deswegen statt, indem alle anderen Exceptions ignoriert werden, und stattdessen die erste signalisierte Exception weitergereicht wird.

Exceptiopn-Tree bzw. Exception-Graph Der Nachteil an den vorhergehenden Vorgehensweise ist dass keine Situationen dargestellt werden können, in der die zeitgleich signalisierten Exceptions, eine andere, kompliziertere Exception darstellen. Eine allgemeine Vorgehensweise kann mittels eines **Exception-Tree** 10.1 oder erweitert mittels eines **Exception-Graph** erreicht werden, die eine Exceptionhierachie repräsentieren. Werden mehrere Exceptions zeitgleich signalisiert dann können diese zu der Exception aufgelöst werden, die die Wurzel des kleinsten Teilbbaums bzw. Teilgraphs ist, der alle signalisierten Exceptions enthält. Diesem Prinzip folgend muss jede Coordinated-Atomic-Action ihren eigenen **Exception-Tree** bzw. **Exception-Graph** besitzen.

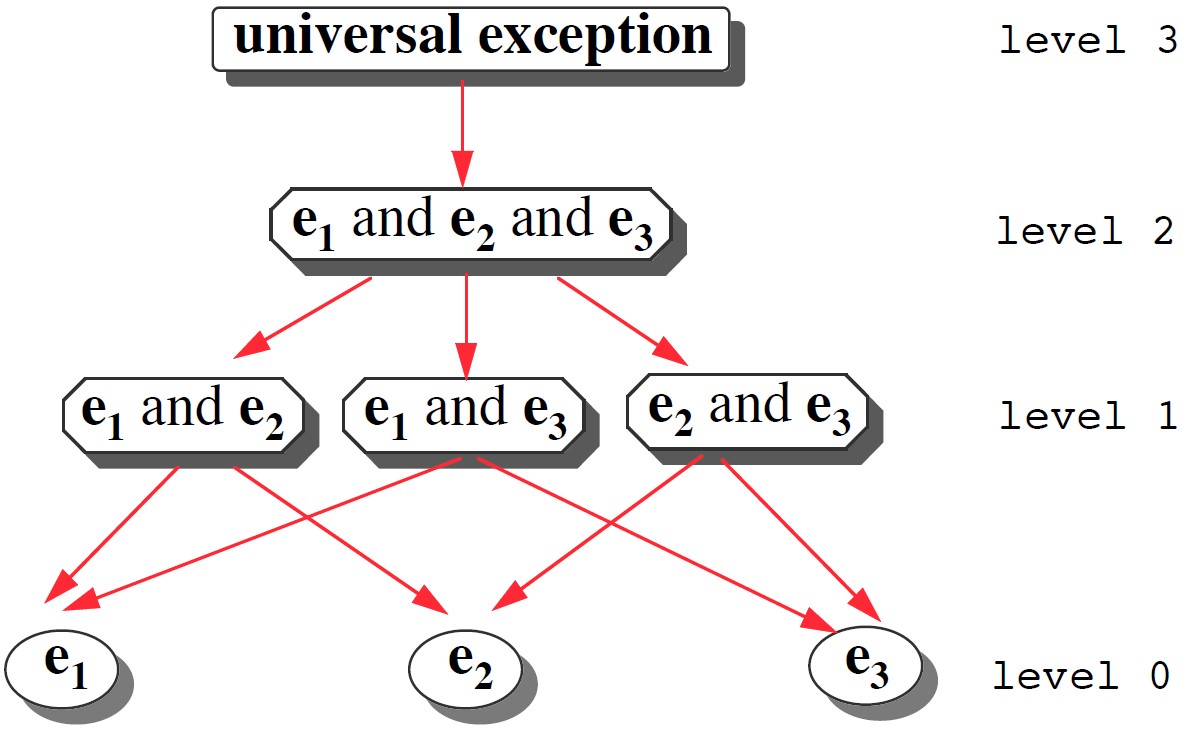


Bild 10.1: Illustration eines Exceptiongraph

Romanovsky et al. [XRR98] sind zu dem Schluss gekommen dass in der Praxis eine Exceptionhierachie komplizierter als eine Baumstruktur ist und deshalb den **Exception-Graph** aus dem Exception-Tree weiterentwickelt. Ein Exception-Graph ist ein direkter Graph, der alle in der Coordinated-Atomic-Action definierten Exceptiontypen enthält. Jeder Exceptiontyp wird durch einen Knoten dargestellt und es bestehen einfache Beziehungen zwischen zwei Exceptiontypen durch direkte Kanten, durch die ausgedrückt wird welcher Exceptiontyp höher als der andere Exceptiontyp steht. Oder anders ausgedrückt welcher Exceptiontyp der Elternteil und welcher der Kindteil ist. Der Eingangsgrad eines Knoten bezeichnet wieviele Elternknoten, und der Ausgangsgrad wieviele Kindknoten dieser hat. Damit gibt es drei Typen von Knoten. Knoten mit Ausgangsgrad 0 repräsentieren die primitivsten Exceptiontypen, die keine weiteren Exceptiontypen abdecken. Knoten mit Eingansgrad und Ausgangsgrad ungleich 0 sind auflösende Exceptions die einen Menge von Exceptiontypen abdecken. Der Knoten mit dem Eingangsgrad 0 ist die Wurzel des Graphen und repräsentiert eine besondere **universal-exception**. Das Signalisieren dieser Exception führt zum Signalisieren einer **undo-Exception** oder **failure-Exception**.

Ein Exception-Graph mit (n+1) Ebenen kann durch n primitiven Exceptions auf Ebene 0 erzeugt werden. Die erste Ebene kann n x (n-1) / 2 auflösende Exception enthalten, die zweite Ebene n x (n-1) (n-2) / 6 usw. Durch diese allgemeingültige Defintion ist es möglich automatisiert einen Exception-Graph zu erstellen. Für eine tatsächliche Anwendung ist aber aus Gründen des Platzes und der Leistungssteigerung ein einfacher Exception-Graph erforderlich.

10.2.7 Behandeln von Exceptions

Die Behandlung von Exceptions kann in verteilten Systemen in zwei Dimensionen eingeteilt werden. In der vertikalen Dimension, werden Exceptions entlang geschachtelten Kontexten weitergereicht. Ist nur ein **node** davon betroffen, so ist dies äquivalent zu der Behandlung von Exceptions in lokalen Systemen, und die geschachtelten Kontexte entsprechen den **stack frames** auf dem Stack, von dem stack frame, in dem die Exception signalisiert, bis zu dem stack frame, in dem sich der passende Exceptionhandler zu der signalisierten Exception befindet. Sind mehre **nodes** von derselben ursächlichen Exception betroffen, so ist ein Kontext von den geschachtelten Kontexten, die Konkatenation von den semantischen äquivalenten Kontexten auf jedem **node**. Dies stellt die

horizontale Dimension der Behandlung von Exceptions dar. Wegen der horizontalen Dimension, setzt sich die vertikale Dimension wenn mehr als ein node von der selben ursächlichen Exception betroffen ist, aus dem zeitgleichen Behandeln jeder Exception in dem semantisch äquivalenten Kontext der einzelnen nodes zusammen.

10.2.7.0.1 lokale Exception-Kontrollflussmechanismen in Verteilten Systemen

termination model Nach Aussagen von Campéas et al [CDUV05] sollte das termination model in verteilten Systemen nur Anwendung bei schweren nichtkorrigierbaren Fehlern finden. Je stärker ein System mit Entitäten aus der realen Welt zu tun hat, desto mehr nicht-fehlerbehfatete Zustände können auftreten, die dennoch behandelt werden müssen. Beispielsweise wird hier ein Netzwerkkabel das wieder eingesteckt werden muss angeführt. Der Prozess der eine Netzwerkschnittstelle verwendet, muss angehalten werden, eine andere Entität behebt den außergewöhnlichen Zustand, und es kann an der Stellte fortgefahren werden.

resumption model Das resumption model wird in der Literatur bisher explizit nur in DOOCE [TY00] verwendet. Es wird aber nur lokal von dem node verwendet, dass der an andere nodes Aufgaben delegiert hat. Eine verteilte Verwendung, bei der auf mehreren nodes parallel bei Auftreten einer gemeinsamen Exception eine Wiederaufnahme stattfindet, konnte leider nicht in der Literatur gefunden werden.

Die Entwickler von Erlang (nach Aussage von Campéas et al [CDUV05]) entschieden nicht das resumption model zu verwenden, da es ihr Ziel ist Software zu bauen die Jahre lang ohne Unterbrechung läuft und keine inkrementellen Erweiterungen erlaubt. Deswegen legten sie Wert auf eine einfache aber ausdrucksstarke Sprache. Das resumption model würde die Komplexität des Codes erhöhen und selten getestete Codepfade einführen. Im Fall von hunderttausenden nebenläufigen Prozessen ist eine fail fast Strategie die robusteste und sicherste Lösung.

retrying model Das retrying model ist in DOOCE [TY00] mit analoger Verwendung zu dem resumption model zu finden. In der Literatur ist bisher leider keine Variante für verteilte Systeme zu finden.

10.2.7.1 Exception-Kontrollflussmechanismen in Verteilten Systemen

Die folgende Einteilung basiert auf der Einteilung der Kontrollflussmechanismen von Christophe Dony et al. [DUV06]

10.2.7.1.1 Single-Distributed-Exception Exception-Handling

Eine einzige asynchrone Nachricht ist die kleinste Granularitätsstufe für ein verteiltes Exceptionhandlingsystem. **Nicht-blockierende Kommunikation** kann durch die zwei verschiedenen Kommunikationsmechanismen **Asynchronous Messages** und **Tuple Spaces** erreicht werden.

Asynchronous Messages Exception Handling kann mit asynchroner Nachrichtenkommunikation vereinbart werden wenn mit jeder Nachricht eine **complaint-address** mitgeschickt wird. Wird eine Exception signalisiert so wird das Objekt, das unter der complaint-address zu erreichen ist, mit einer vordefinierten Nachricht (Bsp. handle) und dem Exception-Objekt als Parameter benachrichtigt. Werden **Futures** verwendet so müssten diese in dem Exceptionhandlingmechanismus berücksichtigt werden.

Tuple Spaces Laut Serugundo und Romanvosky [DMS03] wird für Exceptionhandling in verteilten Systemen die tuple spaces benutzen am besten ein externer Mechanismus verwendet, um sicherzustellen dass jede Exception korrekt behandelt wird und nicht davon abhängig ist ob der geeignete Prozess das **Exception tuple** liest und es behandelt. Das CAMA system fordert deshalb das jedes Tupel mit der Referenz zu einer **tuple space trap** versehen werden, die über die Exception informiert werden und entscheiden können sie zu dem Aufrufer, einem dedizierten Handler-Agenten oder einem Zusammenschluss von (betroffenen) Agenten weiterzureichen.

Diese beiden Mechanismen haben jedoch den Nachteil dass sie nur eine gesendete Nachricht und damit nur eine Exception betrachten. Somit werden keine **concerted exceptions** unterstützt.

10.2.7.1.2 Multiple-Distributed-Exceptions-Single-Handler Caller-Based Exception-Handling

Durch diesen Mechanismus kann ein einziger Exception-Handler eine Block von asynchronen Nachrichtenbefehlen zugeordnet werden. Dadurch ist es möglich das in diesem Block nebenläufig singalisierte Exceptions zu einer **concerted exception** aufgelöst werden können. Es existieren folgende verschiedene Mechanismen [MDB+06] um zu einer **concerted exception** aufzulösen:

Diese Mechanismen erfordern dass der Sender der Nachrichten die concerted exception behandelt. Es findet also kein collaborative Exception-Handling statt. Es können also keine optimistischen Annahmen von den Teilnehmern getroffen werden.

ProActive versteckt die Nebenläufigkeit und signalisiert nur die zuerst aufgetretene Exception innerhalb eines try-Blocks. Dieses Vorgehen basiert auf der Annahme dass alle asynchrone Aufrufe starken Bezug zueinander haben und so die zuerst aufgetretene Exception das vorliegende Problem gut beschreibt.

SaGE Beim Auftreten jeder Exception wird eine **concert**-Funktion ausgeführt, die entscheidet ob die Exception ignoriert, eine concerted Exception erstellt wird oder die Exception nur geloggt wird.

Arche aggregiert alle nebenläufig signalisierten Exceptions und schickt diese durch eine **resolution function** die eine **concerted eception** signalisiert. Im Unterschied zu SaGE wird hier eine Exception basierend auf **vollständigen Informationen** signalisiert, was den gesamten Prozess vereinfacht, aber zu Leistungseinbußen führt.

DOOCE verwendet ein try-catch-finally Konstrukt mit folgender Semantik. Alle Exceptions die in dem try-Block signalisiert werden, können von dem catch-Block behandelt werden. Exceptions die nicht von dem innersten verschachtelten catch-Block behandelt werden, werden von dem finally-Block gesammelt und im finally-Block behandelt wenn alle asynchrone Aufrufe zurückgekehrt sind.

10.2.7.1.3 Multiple-Distributed-Exceptions-Single-Handler Collaborative-Based Exception-Handling

Open Multi-threaded Transactions beinhalten eine Gruppe von zusammenarbeitenden Threads, die mittels gemeinsamer Objekte miteinander kommunizieren. Diese Objekte verwalten ihre eigene Konsistenz um sicherzustellen dass eine Exception in einem Thread nicht auf andere Threads überschlagen kann. Somit behandelt jeder Thread Exceptions lokal und es findet *kein* **collaborative Exception-Handling** statt.

Coordinated Atomic Actions Der Exception-Handling-Mechanismus löst nebenläufig auftrretende Exceptions mittels eines **Exception-Graph** zu einer **concerted Exception** auf und signalisiert diese in allen Teilnehmern. Jeder Teilnehmer muss die Coordinated Atomic Action mit dem selben Ergebnis verlassen, ein Verbindungsabbruch eines Teilnehmers sorgt deshalb zu einem Abbruch des gemeinschaftlichen / collaborative Exceptionhandling.

The Guardian Model *vermeided* **transaktionsähnliche** Mechanismen, sondern jeder Prozess verwaltet Kontexte mit einem symbolischen Namen die auf einen Kontext-Stapel gelegt werden. Wenn eine Exception signalisiert wird muss ein Exception-

Handler Kontext angegeben werden. Alle Prozesse die sich in dem selben Kontext befinden betreiben **collaborative Exceptionhandling**. Da jeder Prozess autonom seinen Kontext ändern kann, muss bei der Signalisierung jeder Exception jeder Prozess kontaktiert werden. Da alle Prozesse für das Exceptionhandling erforderlich sind, ist *kein* **loosely-coupled exception handling** möglich.

10.3 Bewertungskriterien

Nach Eric Platon et al [PHS06] ist es sinnvoll ein Exceptionhandlingmechanismus für verteilte Systeme zu bewerten wie weit sie die quantitativen Kriterien **Verteilungsgrad** und **Kopplungsgrad** des verteilten Systems beeinflussen oder verändern. Desweiteren können folgende Kriterien verwendet werden um Exceptionhandlingmechanismen zu bewerten. [MS04]

Verteilungsgrad beschreibt, ob der Mechanismus zentral oder verteilt ist.

Kopplungsgrad beschreibt, wie unabhängig die Agenten sind.

MAS-Offenheit bezeichnet, ob der Exceptionhandlingmechanismus für offene Systeme geeignet ist, zu dem beliebig Teilnehmer beitreten oder gehen können, und nicht gefordert werden muss dass alle Agenten nur wohlwollend sind.

generisch Möglichkeit dass Exceptionhandling an verschiedene Systemvoraussetzungen anzupassen (beispielsweise Businessanwendungen, Simulationen, etc.).

konkret Die Technik soll in die Implementierung von konkreten Systemen führen.

nicht-invasiv Interne Zustände von Agenten sollen nicht von außen zugänglich sein.

redundant Die Anwesentheit von nicht-kollaborativen Agenten in offenen System bedingt die Verwendung von Heuristiken um Alternativen zu finden, wenn ein Agent eine Exception nicht beheben kann.

Kontext-Scope beschreibt, wie weit sich der Kontext eines Teilnehmers erstrecken kann. Ein Teilnehmer verfügt meistens nur über teilweisen Zugriff auf das gesamte Wissen des Systems. Dieser lokale Kontext-Scope ist nur eine Teilmenge der Umgebung, des Gesamt-Kontext-Scopes. Der Guardian Ansatz nach Miller und Tripathi [MT02], und der Sentinel Ansatz von Haegg [Häg96], zeigen dass der Kontext in verteilten Systemen effizient ausgenutzt werden kann um einen passenden Exceptionhandler zu finden. Kontext bezeichnet hier nach Weyns et al. [WVP+05], die Informationen und die Ressourcen die in der Agentenumgebung verfügbar sind.

dynamische Exceptionbedeutung Das Behandeln von Exceptions setzt voraus dass das **domain knowledge** in die Entscheidungsfindung miteinbezogen wird. Eine bestimmte Exception hat für Teilnehmer desselben Protokoll in unterschiedlichen Domänen eine andere Gewichtung. Eine Verzögerungs-Exception hat so in einem Prozess zur Zahlung für ein Hotelzimmer eine geringere Gewichtung, als in einem Versandprozess von lebensrettenden Medikamenten. Exceptions müssen somit in verschiedenen Domänen unterschiedlich behandelt werden. Es ist nötig eine Abstraktion zu schaffen, die domän-unabhängige Beziehungen zwischen Teilnehmern heraus faktorisieren kann.

dynamische Erweiterung von Exceptionhandlern Unerwartete Exceptions erfordern eine Änderung des Prozessmodells zur Laufzeit. Es wird eine Abstraktion benötigt, die es erlaubt **pragmatic exceptions** einzuführen, die es erlauben über Bedeutungen in einem Kontext nachzudenken.

Im folgenden werden nun die bereits im Abschnitt zuvor erwähnten Exceptionhandlingmodelle vorgestellt und nach den hier aufgestellten Kriterien bewertet.

10.4 JCilk

Jcilk [DACL07] ist eine Java-basierte multithreaded Programmiersprache zur parallelen Programmierung, welche die Semantik von Java erweitert, in dem sie Sprachkonstrukte aus Cilk zur parallelen Steuerung einführt.

10.4.1 Exceptionhandling

Dazu werden die Befehle cilk, spawn und sync eingeführt. cilk ist ein Methodenmodifizierer der eine Methode zu einer cilk Methode macht. Wird eine cilk Methode aufgerufen, so werden die aufrufende Methode und die cilk Methode, als aufgerufene Methode, parallel ausgeführt. Innerhalb einer Methode kann auch der spawn Befehl verwendet werden, um weitere Methoden (Kindprozesse) nebenläufig auszuführen. Der Befehl sync blockiert so lange, bis alle nebenläufigen Kindprozesse ihre Ausführung beendet haben. spawn und sync können nur innerhalb von cilk Methoden verwendet werden. In der ursprünglichen Cilk Programmiersprache können Methoden nur normal beendet werden, aber keine Exceptions signalisieren.

Wird eine Exception signalisiert, so müssen alle Seiten-Berechnungen, die von dem Aufruferkontext gestartet worden sind, auch abgebrochen werden, um keine inkonsis-

tenten Zustände zu hinterlassen. Dazu wird die Exception CilkAbort eingeführt, die von Throwable abgeleitet ist, um Aufräumarbeiten durchführen zu können. Desweiteren wird zur Realisierung des hier vorgeschlagenen Exceptionhandling ein Konstrukt namens inlet benötigt, das ein Codestück ist, das innerhalb eines Elternteils im Auftrag eines Kindprozesses operiert.

Wenn eine cilk Methode nebenläufig ausgeführt wird, wird ein Program cursor für die Methodeninstanz erstellt, der äquivalent zu einem program counter ist, aber sich auf der Sprachebene und nicht auf der Maschinenebene befindet. Terminiert die Methode, so wird der Program cursor zerstört. Der Program cursor einer cilk Methode wird als primärer Program cursor bezeichnet, für die nebenläufige Ausführung von Kindprozessen, werden sekundäre Program Counter angelegt. Über die sekundären Program Counter werden auch Rückgabewerte an die Eltern-Methode übergeben.

Alle Abbrüche in Cilk geschehen semisynchron, d.h. alle Program Cursor bleiben innerhalb der Threadgrenzen, wenn eine Methode abgebrochen wird. Wenn beispielsweise in 10.2 C eine Exception signalisiert während D ausgeführt wird, so kehrt der Thread der D ausführt von D zurück und schreitet bis zum sync in Zeile 6 zurück, bevor er abgebrochen wird. JCilk garantiert keine sofortigen Abbrüche, wenn eine Exception signalisiert wird und der primäre program cursor die nächste Threadgrenze erreicht. Es wird aber sichergestellt, dass wenn ein Abbruch stattfindet, der primäre Cursor innerhalb der Threadgrenze verbleibt (entsprechendes gilt für sekundäre Cursor).

```
1 cilk int f1() {
2  int w = spawn A();
3  int x = B();
4  int y = spawn C();
5  int z = D();
6  sync;
7  return w + x + y + z;
8 }
```

Bild 10.2: Beispiel in JCilk, das zeigt wie nebenläufig auftretende Exceptions behandelt werden können

Ein cilk try-Block kann vom primären Program Cursor verlassen werden, bevor die Ausführung des try-Blocks vollständig beendet ist. So kann der primäre Cursor in 10.2 nachdem Aufruf der Methode A in Zeile 4, noch ein weiteren Kindprozess B in Zeile 9 aufrufen, bevor f3 in den Zustand suspended übergeht, bis die beiden Kindprozesse beendet werden. Der primäre Cursor kann also außerhalb der cilk try-Blöcke fortgeführt werden, obwohl Thread A und Thread B noch Exceptions signalisieren können. Wird

eine Exception signalisiert, so wird aber durch ein catchlet (ist ein spezielles inlet), der entsprechende catch-Block der dem cilk try-Block zugeordnet wurde, ausgeführt. finally-Blöcke sind auch möglich und werden entsprechend von einem finallet (ist ein spezielles inlet) ausgeführt.

```
1  cilk int f3() {
2    int x, y;
3    cilk try {
4     x = spawn A();
5    } catch(Exception e) {
6     x = 0;
7    }
8    cilk try {
9      y = spawn B();
10   } catch(Exception e) {
11     y = 0;
12   }
13   sync;
14   return x + y;
15 }
```

Bild 10.3: Beispiel in JCilk, das die Verwendung eines cilk-try-Blocks zeigt

Wird eine Exception signalisiert, so müssen auch Seiten-Berechnungen abgebrochen werden. Seiten-Berechnungen beinhalten alle Methoden, die dynmaisch von der catch-clause, die die Exception behandelt, eingeschlossen werden. Dies kann acuh den primären Program Cursor der Methode, die den cilk try-Block enthält, wenn sich der Cursor immer noch innerhalb des cilk-try Blocks befindet.

Der Exceptionhandlingmechanismus folgt in folgenden sechs Schritten ab:

(1) Eine Exception wird ausgewählt (wenn mehrere Exceptions gleichzeitig signalisiert werden) von dem nähesten dynamisch umschließenden catch-Handler behandelt.

(2) Es muss eine Exception signalisiert werden, dass die Seiten-Berechnungen abgebrochen werden sollen.

(3) Es wird gewartet, bis alle dynamisch umschlossenen Kindprozesse beendet werden. Dies geschieht entweder durch normales Beenden oder durch einen Abbruch auf Grund von Schritt 2.

(4) Es wird gewartet bis der primäre Program Cursor den cilk try-Block verlässt. Dies geschieht entweder durch normales Beenden oder durch einen Abbruch auf Grund von Schritt 2.

(5) Das catchlet dass mit der signalisierten Exception verknüpft ist, wird ausgeführt.

(6) Wenn dem cilk try-Block ein finally-Block zugeordnet wurde, so wird das entsprechende finallet ausgeführt.

Werden mehrere Exceptions signalisiert, so wird nur eine Exception ausgewählt und behandelt. Falls sich unter den Exceptions eine CilkAbort Exception befindet so wird diese ausgewählt, ansonsten die Exception, die zu dem nächstgelegenen umschließenden Exceptionhandler passt.

10.4.2 Bedingungen

Soll cilk das Exceptionhandling übernehmen, so müssen die entsprechenden Codestellen mit dem Befehlswort cilk gekennzeichnet werden.

10.4.3 Leistungen

Zu den Leistungen des Modells zählt das optimistische asynchrone Aufrufen von mehreren Methoden. Die Exceptionhandler können den asynchronen Aufrufen zugeordnet werden. Es müssen keine gesonderten Exceptionhandler beim Abruf der zurückgegebenen Werte verwendet werden. Weitere aufgerufene Methoden von asynchronen Aufrufen, die eine Exception signalisieren, werden automatisch beendet.

10.4.4 Bewertung

Das Abbrechen der Seiten-Berechnungen erspart dem Programmierer viel Arbeit, da er sich nicht allen Abhängigkeiten bewusst sein muss. Es dürfen, aber keine Zugriffe auf gemeinsam verwendete Objekte erfolgen, da diese sonst in einem inkonsistenten Zustand hinterlassen werden würden. Um dies zu garantieren muss ein Transaktionssystem verwendet werden.

Verteilungsgrad Der Mechanismus ist auf Threads verteilt.

Kopplungsgrad Die einzelnen Threads sind durch die Einführung der Program Cursor sehr weit entkoppelt. Nur an wenigen Synchronisationspunkten besteht Kopplung.

MAS-Offenheit Keine Aussage möglich.

generisch Nein.

konkret Ja.

nicht-invasiv Die Inlets werden automatisch ausgeführt und sind dem Thread untergeordnet, der sie normalerweise ausführen würde. Das Geheimnisprinzip wird dadurch nicht verletzt. Daher ist das Modell nicht invasiv.

redundant Nein.

Kontext-Scope Der Kontext-Scope erstreckt sich nur auf den umschließenden cilk try-Block.

dynamische Exceptionbedeutung Nein.

dynamische Erweiterung von Exceptionhandlern Nein.

10.5 Proactive

Proactive [CC05] ist eine GRID Java Bibliothek für parallele, verteilte und nebenläufige Berechnungen und weiterer Funktionalität wie Mobilität und Sicherheit in einem einheitlichen Framework.

10.5.1 Exceptionhandling

Kommunikation findet zwischen aktiven Objekten statt. Jedes aktive Objekt hat seinen eigenen Thread um Anfragen, die sich in der Warteschlange für eintreffende Nachrichten befinden, auszuführen. Methodenaufrufe (im Client) sind damit, meist asynchrone Nachrichten, an die Eingangbox der aktiven Objekte (Server). Ein solcher Methodenaufruf, gibt im Client ein Future-Objekt zurück. Wenn das Ergebnis bereit steht, so liefert das Future-Objekt den berechneten Wert zurück. Ist das Ergebnis noch nicht bekannt, aber der Wert des Future-Objekts wird abgefragt, so blockiert der Aufruf, bis der berechnete Wert zur Verfügung steht. Tritt in der aufgerufenen Methode eine Exception auf, so wird die Exception erst im Aufrufer signalisiert, wenn auf den Wert des Future-Objekts zugegriffen wird. Der Zugriff findet aber meist in einem ganz anderen Kontext, als der Methodenaufruf statt. Die geeigneteste Stelle auf die Exception zu reagieren wäre aber nicht beim Abfragen des Wertes, sondern beim Methodenaufruf. Dafür ist es nötig ein Stack Unwinding für asynchrone Exceptions nachzubilden. Synchrone Exceptions werden normal behandelt und von dem asynchronen Behandlungsmechanismus unangetastet gelassen.

Dazu wird vor dem try-Block die trywithCatch() Methode 10.4 aufgerufen, die als Argumente eine Liste aller asynchronen Exceptions erhält, die von den asynchronen

Methoden im try-Block signalisiert werden können. Diese Methode legt einen Exceptionhandler auf einen virtuellen Stack. Am Ende des try-Blocks muss die Methode endtrywithCatch() aufgerufen werden, die auf die Rückkehr aller asynchron aufgerufenen Methoden wartet, die eine asynchrone Exceptions werfen können, die der trywithCatch() Methode übergeben wurden. Falls Future-Objekte Exceptions enthalten, werden diese durch die endtrywithCatch() Methode signalisiert. In dem folgenden finally-Block muss durch den Aufruf der removeTryWithCatch() Methode, der Exceptionhandler auf dem virtuellen Stack für asynchrone Exceptions entfernt werden.

```
1  public class   RemoteObject {
2      public DangerousThing dangerousMethod()  throws
           DangerousException  {}
3
4      public static void  main(String[] args)   {
5          ProActive.tryWithCatch(DangerousException.class);
6
7          try {
8              DangerousThing[] dt; RemoteObject[] ro;
9
10             for (int i = 0;  i < dt.length; i++)
11                 dt[i] = ro[i].dangerousMethod();
12
13             ProActive.endTryWithCatch();
14         } catch  (DangerousException  de)  { ... }
15         finally {
16             ProActive.removeTryWithCatch();
17         }
18     }
19 }
```

Bild 10.4: ProActive: try-catch Beispiel für asynchrone Exceptions

Treten gleichzeitig mehrere Exceptions auf, so wird nur die erste Exception signalisiert und die anderen ignoriert. Die Argumentation ist, dass die Exceptions gemeinsamen Ursprung haben, und die zweite Exception bei einem synchronen Aufruf nicht auftreten würde.

10.5.2 Bedingungen

Kommunikation zwischen Teilnehmern findet durch aktive Objekte statt, die Nachrichten austauschen.

10.5.3 Leistungen

Asynchrone Aufrufe werden durch Future-Objekte realisiert. Dadurch wird Kopplung zwischen Aufrufer und aufgerufener Methode realisiert. Mehrere auftretende Exceptions werden behandelt indem die exceptional condition auf eine einzige signalisierte Exception zurückgeführt wird.

10.5.4 Bewertung

Durch Future-Objekte wird eine einfache Möglichkeit bereitgestellt geringe Kooplung zu erreichen. Jede Anweisung in der auf den Wert eines Future-Objekts zugegriffen wird, muss mittels eines try-catch-Blocks geschützt werden.

Verteilungsgrad Der Mechanismus ist verteilt.

Kopplungsgrad Der Kopplunggrad ist gering.

MAS-Offenheit Nein.

generisch Nein.

konkret Ja.

nicht-invasiv Ja.

redundant Nein.

Kontext-Scope Der Kontext-Scope beschränkt sich auf den Kontext im Aufrufer.

dynamische Exceptionbedeutung Nein.

dynamische Erweiterung von Exceptionhandlern Nein.

10.6 SaGE

SaGE [CDUV05] ist eine Erweiterung, die auf Java basiert. In SaGE besteht aus einem Protokoll, das eine festgelegte Menge von Interaktionen zwischen Gruppen nebenläufiger und möglicherweise verteilten Entitäten definiert. Aufbauend auf das Protokoll wird ein Exceptionhandlingsystem definiert, das als **Service** bezeichnet wird.

10.6.1 Exceptionhandling

Ein Service fasst einen Agenten, einen Namen, eine Menge von formalen Parametern und eine Funktionseinheit zusammen. Das Protokoll ist semi-synchron, da ein aufgerufener

Prozess asynchron ist bis zu einem späteren Zeitpunkt wenn die Ausgabe abgerufen wird, die es sich verpflichtet hat bereitzustellen.

Für Java wird die SaGE Exception eingeführt. Jeder Service kann eine **handler clause** bestehend aus einer beliebigen Menge von abgeleiteten Exceptions der SaGE Exception führen. Zusätzlich zu dem Java Exceptionhandlingmechanismus verfügt SaGE über die **concertation**-Funktion. Diese sorgt dafür wenn ein redundanter Sub-Service versagt nicht der ganze Service-Baum zerstört wird. Dazu filtert diese besondere Funktion die Exceptions und entscheidet ob die Exception nach oben weitergereicht werden muss oder ignoriert werden kann.

10.6.2 Bedingungen

Es muss ein Protokoll definiert werden.

10.6.3 Leistungen

Anstelle von einer Aufrufreihe von Methoden-Aufrufen in nebenläufigen objektorientierten Programmen, wird in SaGE ein Baum von Service-Aufrufen verwendet. Dies geschieht aus zwei Gründen. Zum einen kann ein Agent parallel viele Instanzen beliebiger Services aufrufen, wobei die Service-Aufrufe von verschiedenen Agenten erfolgen können. Zum anderen kann ein Service einen anderen, weiteren Service-Aufruf tätigen. Dies ist beliebig rekursiv fortsetzbar. In diesem Fall ist der Service nicht synchron an seine normale oder durch Exceptions-beeinflusste Ausgabe gebunden. Dies ermöglicht es Sub-Services redundant auszuführen und sorgt dafür dass die **worst-case** Laufzeit auf die verteilten Sub-Services verteilt wird, anstatt aus der Summe aller Serviceaufrufe zu bestehen.

10.6.4 Bewertung

Mit SaGE können nur Probleme gelöst werden wenn sie sich mit der Idee des Service vereinen lassen. Die Effizienz der Nebenläufigkeit ist sehr hoch da möglichst wenige Einschränkungen auf die Ausführreihenfolge der Sub-Services gelegt werden. Die einzige Einschränkung ist dass der Supervisor-Service vor allen Sub-Services startet und erst nach Beendigung aller Sub-Services beendet wird. Aus diesem Grund müssen Seiteneffekte, wie Sperrungen von Ressourcen und **race conditions** mit low-level Konstrukten wie **locks** und **mutexes** realisiert werden umso eine Serialisierung und Reihenfolge festzulegen. Da

dies aber schwierig zu verifizieren ist und Skalierungsprobleme auftreten können, ist ein Transaktionssystem erforderlich.

Verteilungsgrad Nach Eric Platon et al. [PHS06] ist dies durch das **concerted exception handling** erfüllt.

Kopplungsgrad Nach Eric Platon et al. [PHS06] ist SaGE nur für objekt-orientierte oder reaktive Agenten ausgerichtet, da Agenten als Softwarekomponenten betrachtet werden.

MAS-Offenheit Nach Eric Platon et al. [PHS06] wird diese nicht efüllt, da nur von wohlwollenden Agenten ausgegangen wird.

generisch Nein.

konkret Ja.

nicht-invasiv Ja. Wird nach Eric Platon et al. [PHS06] von SaGE erfüllt.

redundant Ja.

Kontext-Scope Der Kontext-Scope ist nur lokal auf den node beschränkt.

dynamische Exceptionbedeutung Nein.

dynamische Erweiterung von Exceptionhandlern Nein.

10.7 Arche

Arches [Iss01] Exception Handling Konzept basiert auf dem termination model des synchronen Exceptionhandlings für lokale Systeme.

10.7.1 Exceptionhandling

Für kooperative Prozesse, die einen gemeinsamen Synchronisationspunkt besitzen, wurde **global exception handling** eingeführt. In der Deklaration eines Prozesses müssen alle Exceptiontypen aufgelistet werden, die der Prozess signalisieren kann. Wenn eine Exception aufgetreten ist, so wird die Exception am Sychronisationspunkt der anderen Teilnehmer signalisiert.

Desweiteren wurde **concerted exception handling** in Arche eingeführt. Wenn mehrere Prozesse Teil einer parallelen verschachtelten Operation bilden, und einer dieser Prozesse aufgrund einer exceptional condition beendet werden muss, so führt dass zur

Beendigung der verschachtelten parallelen Operation. Es können auch mehrere Exceptions auftreten, wenn mehr als ein Prozess aus dem Beispiel eine Exception signalisiert. Die exceptional condition kann dadurch von der Zusammensetzung der signalisierten Exceptions abhängen. Eine Exception, die sich aus der Signalisierung mehrere Exceptions ergibt, wird als **concerted** bezeichnet. Um eine solche concerted exception zu berechnen, bedarf es semantisches Wissen über die Exceptions und muss deswegen durch eine **resolution function** berechnet werden.

Das Weiterreichen von Exceptions muss explizit erfolgen. Dadurch können die Exceptionhandler für jede Exception statisch bestimmt werden.

Default exception handling kann unterstützt werden, indem der Default Exception-Typ **failure** weitergereicht wird.

Eine multiprocedure besteht aus einer Menge von Blöcken, die nebenläufig ausgeführt werden. Jede Blockkomponente, erhält eine Teilmenge der Eingabeparameter und gibt eine Teilmenge des Ergebnisses zurück.

Arche ist für verteilte Berechnungen innerhalb eines local area networks ausgelegt.

Das coordinated Exceptionhandling basiert auf eine führungsrollenbestimmte Exception-Weiterreichung. Der node, der die nötigen Exceptionhandler verfügt, wird in Arche als **coordinator** bezeichnet. Er enthält eine Liste aller Teilnehmer in der coordinated Aktion. Falls ein Teilnehmer nicht mehr vefügbar sein sollte, wird eine timeout Exception dem coordinator signalisiert.

10.7.2 Bewertung

Verteilungsgrad Verteilt.

Kopplungsgrad Kopplung besteht nur am Synchronisationspunkt.

MAS-Offenheit Nein.

generisch Nein.

konkret Ja.

nicht-invasiv Ja.

redundant Nein.

Kontext-Scope Der Kontext-Scope ist lokal auf den node begrenzt.

dynamische Exceptionbedeutung Nein.

dynamische Erweiterung von Exceptionhandlern Nein.

10.8 DOOCE

DOOCE steht für ”distributed object-oriented computing environment” und stellt eine Programmiersprache und eine Laufzeitumgebung in C++ von Tazuneki et al. [TY00] bereit.

10.8.1 Exceptionhandling

Es wird zusätzlich zu dem Addressbereich des Betriebssystems, ein virtueller flacher Addressbereich für Objekte bereitgestellt. Durch diesen virtuellen Addressbereich können zwischen den Objekten Nachrichten ausgetauscht werden. Dies ist auch transparent für die Verteilung, d.h. auch nicht-lokalen Objekten können so Nachrichten geschickt werden.

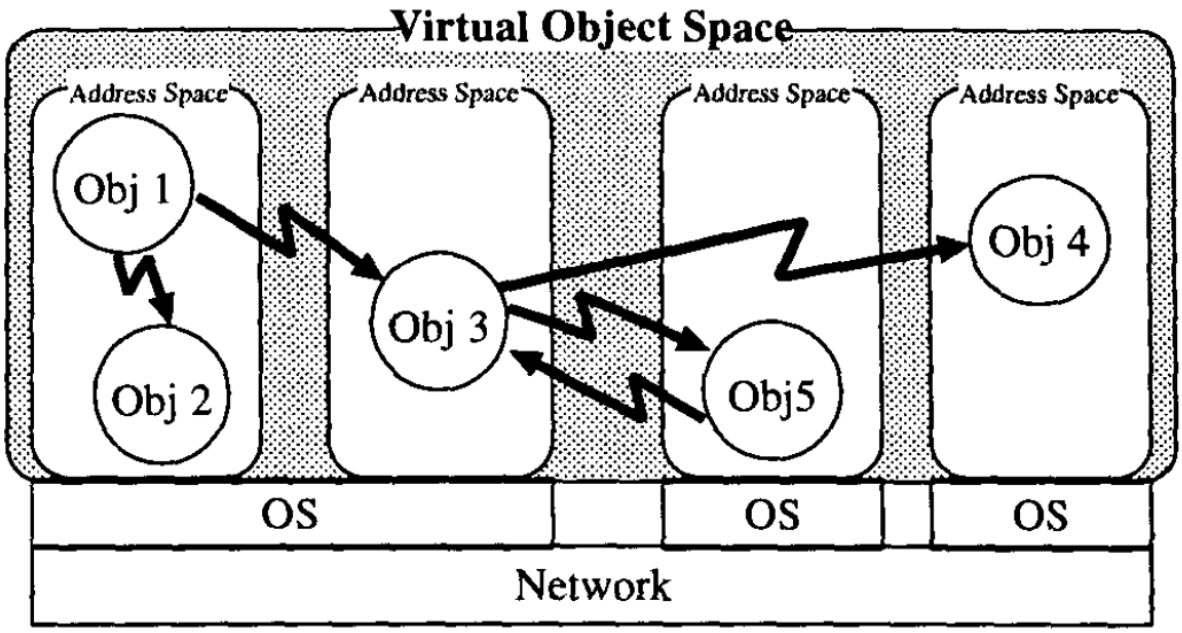


Bild 10.5: DOOCE virtueller Addressbereich

Jedes Objekt hat eine Message Queue, die mit FIFO Semantik funktioniert und seriell abgearbeitet wird.

Es ist blockierender und nicht-blockierender Nachrichtenversand möglich. **par** beschreibt einen Block, indem Nachrichtenbefehle stehen können, die gleichzeitig ausgeführt werden. Der par-Block wird erst verlassen, wenn alle Ergebnisse zu den einzelnen Nachrichten erhalten worden sind. Der nicht-blockierende Nachrichtenversand erfolgt durch das async Sprachkonstrukt. Wird allerdings auf den Wert eines Ergebnisses zurückgegriffen, der noch nicht empfangen worden ist, so blockiert der Befehl. (Future-Semantik)

Es wird das termination model, resumption model und retry model unterstützt. Als Default ist das termination model eingestellt. Durch die Befehle resume und retry, die in einem catch-Block aufgerufen werden können, kann das resumption bzw. retry model explizit verwendet werden.

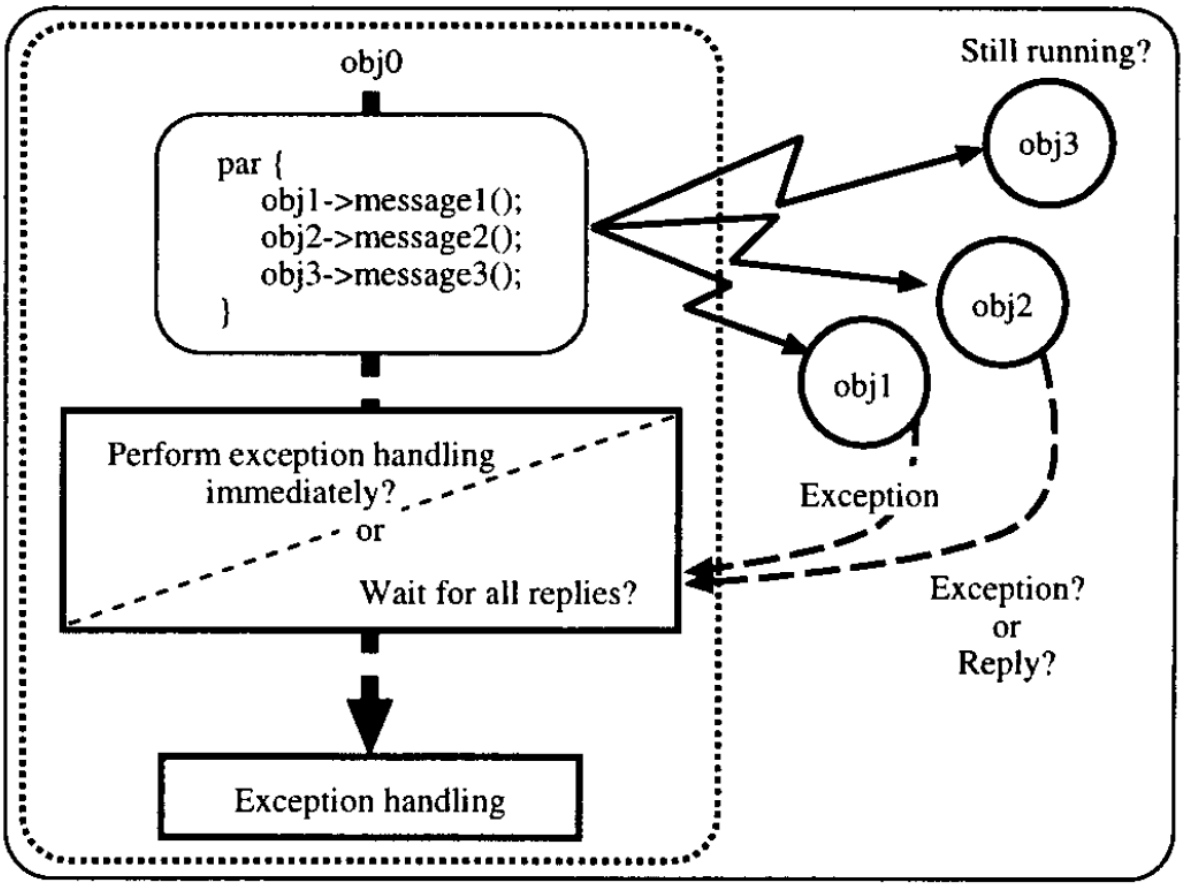


Bild 10.6: Exceptionhandling in DOOCE

Der Aufruf eines Exceptionhandlers erfolgt als Standardeinstellung erst wenn die nebenläufigen Objekte ihren Endzustand erreicht haben. Soll der Exceptionhadler sofort aufgerufen werden, so muss dies durch das Senden einer Notification Nachricht erfolgen, die durch Verwendung des try_noti anstelle des try-Konstrukts automatisch gesendet wird. Die Message Queue eines Objekts speichert auch einkommende Exceptionnachrichten. Im Falle einer Notification Nachricht, werden alle Exceptions außer der ersten (entspricht der Exception in der Notification Nachricht) ignoriert. Signalisiert ein anderes Objekt eine Exception während es eine Notification Nachricht empfängt, so wird die empfangene Nachricht ignoriert, da diese nur dazu dient eine Exception anzudeuten. Auf eine Notification Nachricht kann mit den Befehlen terminate und ignore reagiert

werden. ignore ignoriert die Nachricht sodass der Zustand derselbe ist, als ob die Nachricht nicht empfangen worden wäre. terminate beendet die Ausführung einer Methode, schickt aber davor eine Exception dass die Methode terminiert wird. Dadurch kann eine Exceptionhandlingmethode ausgeführt werden bevor die Methode terminiert wird.

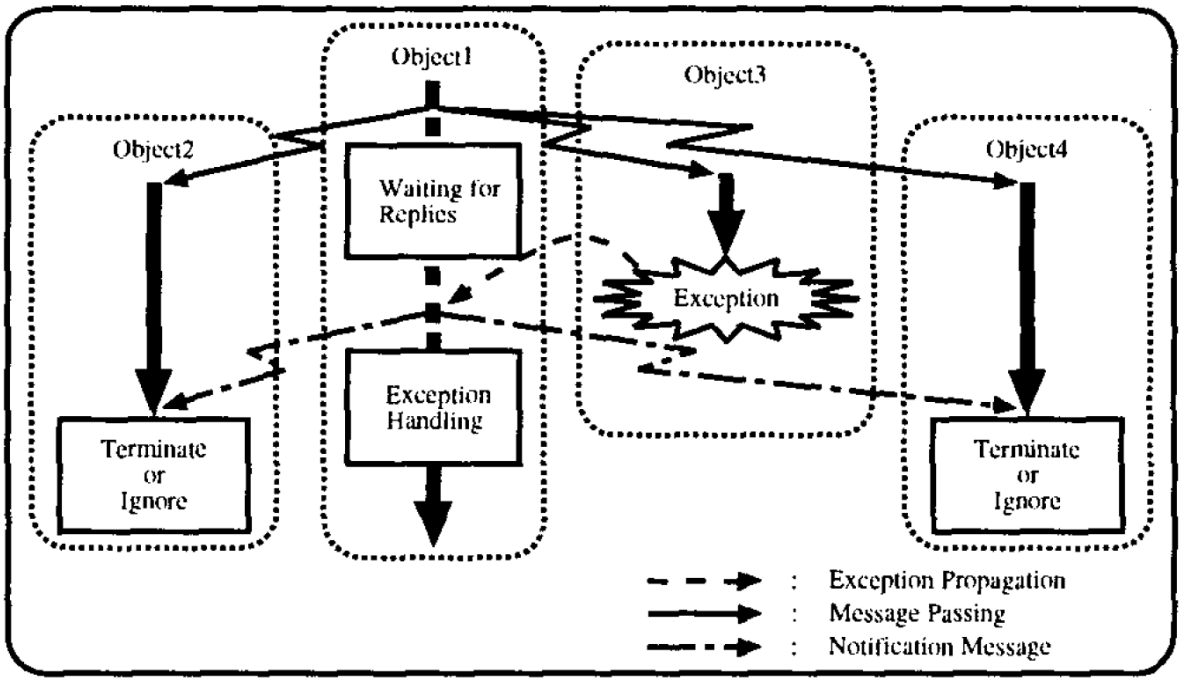


Bild 10.7: Notification Nachricht in DOOCE

Alle Objekte in DOOCE werden von der Klasse Notification abgeleitet, und die Methode Notification kann überschrieben werden, um das Verhalten anzupassen. Der Zustand nebenläufiger Ausführungen kann nicht bestimmt werden, deshalb können Inkonsistenten entstehen. Um dies zu verhindern soll in DOOCE in Zukunft noch ein Transaktionsverwaltungsmechanismus eingebaut werden. Es handelt sich um ein Führungsrollenbestimmtes Weiterreichen der Exceptions. Der Exceptionkontext liegt in dem Objekt, dass den anderen Objekten Nachrichten geschickt hat. Durch die Notification-Nachricht reicht der Führer, die Exception an die anderen Objekte weiter. Wenn gleichzeitig mehrere Exception auftreten, können einige Exceptionhandler asynchron ausgeführt werden. Um dies zu verhindern, schlagen Tazuneki et al. [TY00] vor, die Ausführungen der nebenläufigen Objekte zu terminieren um zu verhindern, dass normaler und Exceptionhandlercode gleichzeitig ausgeführt werden kann.

Gleichzeitig auftretende Exceptions können wie folgend behandelt werden. Im catch-Block können Komma-separiert Exceptions angegeben werden. Nur wenn alle aufgelisteten Exceptions aufgetreteten sind, wird der Exceptionhandler betreten. Die Reihenfolge der

catch-Blöcke ist entscheidend. Sind Exceptions A und B aufgetreten 10.8, und existieren catch-Blöcke für A, für B und für A und B, so werden die Exceptions A und B bereits durch die einzelnen catch-Blöcke für A und B behandelt, und der catch-Block für das gleichzeitige Auftreten von A und B wird nicht mehr aufgerufen. Deshalb ist darauf zu achten, dass Exceptionhandler die gleichzeitig auftretende Exceptions behandeln vor Exceptionhandler stehen, die nur eine Teilmenge der Exceptions anderer Exceptionhandler abedecken.

```
1  try
2  {
3    // Programmcode, der eine Exceptions vom Typ ExceptionA und
       ExceptionB auslösen kann
4  }
5  catch ( ExceptionA ex)
6  {
7    // Programmcode zum Behandeln der Exception A
8  }
9  catch ( ExceptionB ex)
10 {
11   // Programmcode zum Behandeln der Exception B
12 }
13 catch ( ExceptionA exA, ExceptionB exB)
14 {
15   // Programmcode zum Behandeln der Exceptions A und B
16 }
```

Bild 10.8: Beispiel in DOOCE, das zeigt wie nebenläufig auftretende Exceptions behandelt werden können

10.8.2 Bedingungen

10.8.3 Leistungen

Mehrere asnychrone Methodenaufrufe können in einen par-Block gekapselt und gemeinsam als Exceptionkontext behandelt werden. Der par-Block ist dabei blockierend. Desweiteren kann mittels Notification Nachricht unmittelbares Exceptionhandling eingeleitet werden. Die Behandlung

10.8.4 Bewertung

Exceptionhandling wird zentral von dem Aufrufer der anderen Objekte durchgeführt. Der par-Block erzeugt eine starke Kopplung zwischen dem Aufrufer und den asynchron

aufgerufenen Methoden. Exceptionhandler für bestimmte Kombinationen von Exception registrieren zu können bietet einen Vorteil gegenüber anderen Modellen, in denen wenn möglich dafür eine resolution-Funktion geschrieben werden muss.

Verteilungsgrad Zentral. Exceptionhandling wird nur von dem Aufrufer durchgeführt.

Kopplungsgrad Gerine Kopplung.

MAS-Offenheit Nein.

generisch Nein.

konkret Ja.

nicht-invasiv Ja.

redundant Nein.

Kontext-Scope Bechränkt sich nur auf lokalen Scope.

dynamische Exceptionbedeutung Nein.

dynamische Erweiterung von Exceptionhandlern Nein.

10.9 Open Multi-Threaded Transactions

Open Multi-Threaded Transactions wurden von Jörg Kienzle [Kie01] eingeführt und sind Transaktionen, denen Threads auch nach Beginn der Transaktion beitreten können. Threads, die Teil einer open multithreaded transaction sind, werden auch als Teilnehmer bezeichnet. Diese Transaktionen erfüllen die ACID Eigenschaften. Threads unterliegen nur zwei Einschränkungen, wenn sie Teil einer Transaktion sind: Ein Thread, der außerhalb einer open multithreaded transaction erstellt wurde, darf nicht innerhalb der Transaktion terminieren. Wenn ein Thread innerhalb einer open multithreaded transaction erstellt wurde, so muss er innerhalb dieser Transaktion terminieren.

Durch diese zwei Regeln kann die Alles-oder-Nichts Semantik von Transaktionen zu jeder Zeit erfüllt werden.

10.9.1 Exceptionhandling

Innerhalb einer Transaktion können Threads auf eine Menge von Transaktionsobjekten zugreifen. In diesem Fall muss Synchronisation stattfinden, um Konsistenz über die zugegriffenen Transaktionsobjekte zu garantieren. Eine Transaktion kann von jedem

Thread gestartet werden. Startet ein Thread eine neue Transaktion innerhalb einer bereits bestehenden Transaktion, so werden die Transaktionen verschachtelt. Es können auch mehrere Transaktionen von unterschiedlichen Teilnehmern einer bereits bestehenden Transaktion gestartet werden. In diesem Fall werden diese eingebetteten Transaktionen nebenläufig ausgeführt. So lange eine Transaktion offen ist, können ihr weitere Teilnehmer beitreten. Für einen Beitritt muss die Identität der Transaktion bekannt sein, die statisch bekannt oder zur Laufzeit ermittelt werden kann. Eine Transaktion kann jederzeit von einem Teilnehmer der Transaktion geschlossen werden um weiteres Beitreten zu unterbinden. Die Transaktion wird automatisch geschlossen, wenn alle Teilnehmer ihre Teilnahme an der Transaktion mittels eines commit- oder abort-Befehls beenden.

Open multithreaded Transactions unterscheiden zwischen internen und externen Exceptions. Interne Exceptions werden von den Teilnehmern deklariert und müssen innerhalb der Transaktion behandelt werden. Externe Exceptions werden außerhalb der Transaktion signalisiert. Jede Transaktion verfügt über die externe Exception Transaction_Abort. Jeder Teilnehmer hat für jede interne Exception, die während seiner Ausführung signalisiert werden kann, einen entsprechenden Exceptionhandler. Kann der Teilnehmer die Exception nicht behandeln so kann eine externe Exception signalisiert werden. Existiert kein Exceptionhandler für eine interne Exception so wird die externe Exception Transaction_Abort signalisiert.

Es können von jedem Teilnehmer eigene externen Exceptions signalisiert werden. Jeder Teilnehmer einer Transaktion gehört einem übergeordneten Exceptionkontext an. Wenn eine Exception von einem anderen Teilnehmer signalisiert wird, so wird diese zu dem übergeordneten Exceptionkontext weitergereicht. Wenn mehrere Teilnehmer eine externe Exception signalisieren, so wird reicht jeder Teilnehmer seine eigene Exception zu seinem eigenen Kontext weiter. Falls irgendein Teilnehmer eine externe Exception signalisiert, so wird die Transaktion abgebrochen und die Tranasction_Abort Exception wird allen anderen Teilnehmern der Transaktion signalisiert, die bereits ein commit-Befehl gesendet haben. Für verschachtelte Transaktionen, müssen die Exceptionhandlingregeln rekursiv von dem Programmierer angewendet werden. Alle externen Exceptions einer Transaktion, stellen interne Exceptions für die umschließende Transaktion dar.

Das folgende Beispiel 10.9 zeigt, dass Thread A eine Transaktion startet, der wenig später Thread B beitritt. Thread A startet einen Thread A', der nach kurzer mit einem commit beendet wird. In Thread A wird eine Exception X signalisiert, die aber lokal behandelt werden kann und die Transaktion nicht beeinträchtigt. Thread A führt nach erfolgreicher Beendigung einen commit-Befehl aus. In Thread B wird eine Exception Y

signalisiert, die vom lokalen Exceptionhandler nicht behandelt werden kann und deshalb wird die externe Exception Z signalisiert. Die Exception Z wird dem übergeordneten Exceptionkontext von Thread B signalisiert, und damit allen anderen Teilnehmern. Thread A, der bereits einen commit-Befehl ausgeführt hat, erhält die Exception Transaction_Abort.

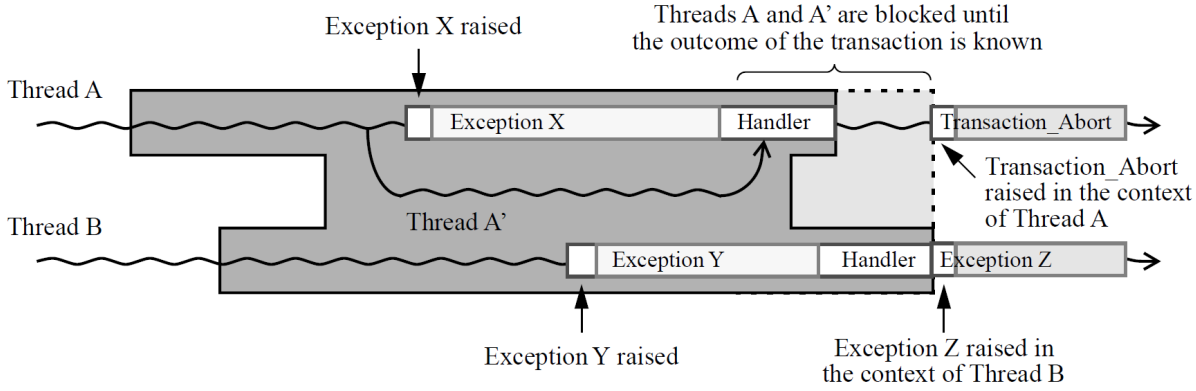


Bild 10.9: Open multithreaded Transactions Beispiel

In Open multithreaded Transactions findet kein koordiniertes Exceptionhandling statt. Jeder Teilnehmer hat seinen eigenen Exceptionkontext und behandelt Exceptions separat. Dadurch ist keine Synchronisation zwischen den Teilnehmern erforderlich, und die Teilnehmer sind nur sehr lose gekoppelt. Gegen koordiniertes Exceptionhandling spricht, dass in anderen Teilnehmern keine Bedeutung haben, da sie den Teilnehmer nicht beeinflussen, oder nicht alle Exceptions definiert sind.

10.9.2 Bedingungen

Ein Thread, der außerhalb einer open multithreaded transaction erstellt wurde, darf nicht innerhalb der Transaktion terminieren. Wenn ein Thread innerhalb einer open multithreaded transaction erstellt wurde, so muss er innerhalb dieser Transaktion terminieren.

10.9.3 Leistungen

Teilnehmer einer Transaktion werden ACID Eigenschaften garantiert.

10.9.4 Bewertung

Verteilungsgrad Verteilt.

Kopplungsgrad Kopplung besteht nur an Synchronisationspunkten. Geringe Kopplung, da jeder Teilnehmer lokales Exceptionhandling durchführt.

MAS-Offenheit Ja, Teilnehmer können auch nachträglich der Transaktion beitreten, so lange dies noch offen ist. Durch lokales Exceptionhandling für jeden Teilnehmer, sind die einzelnen Teilnehmer vor anderen Fehlverhalten anderer Teilnehmer weitestgehend geschützt.

generisch Nein.

konkret Ja.

nicht-invasiv Ja.

redundant Nein.

Kontext-Scope Bezieht sich nur auf den Kontext des einzelnen Teilnehmers.

dynamische Exceptionbedeutung Nein.

dynamische Erweiterung von Exceptionhandlern Nein.

10.10 Coordinated Exception Handling

Im folgenden wird ein Exceptionhandlingsmechanismus von Xu, Romanosvky und Randell [XRR98] betrachtet der eine allgemeine Lösung für verteilte Systeme mit verteilten Objekten anstrebt. Der Ansatz basiert auf der Verwendung von **Coordinated-Atomic-Action**s. Das ist eine erweiterte Form der atomic action Struktur die für **verteilte Objektsysteme** ausgelegt ist.

10.10.1 Allgemeines Modell

Ein **distributed object system** besteht aus **nodes** die durch ein **communication network** verbunden sind. Kommunikation findet durch **message passing** zwischen den **nodes** statt die über **network nodes** verfügen. In diesem Zusammengang wird unter **Exceptionhandling** ein allgemeiner Mechanismus verstanden, der außergewöhnliche Systemzustände und Fehler die durch Hardwareversagen oder Softwareversagen ausgelöst werden, bewältigen kann. Eine **Coordinated-Atomic-Action** stellt einen Mechanismus bereit um eine Reihe von Operationen auf einer Menge von Objekten auszuführen. Diese Operationen werden **kooperativ** von einer oder mehreren **Rollen** innerhalb einer Coordinated-Atomic-Action ausgeführt. In der Schnittstelle der Coordinated-Atomic-Action werden die externe Objekten mit den Rollen angegeben, die diese manipulieren.

Um eine Coordinated-Atomic-Action auszuführen wird eine Gruppe von Threads benötigt, von denen jeder eine der in der Schnittstelle spezifizierten Rolle ausführt, wobei jede Rolle nur von einem Thread ausgeführt werden kann.

10.10.2 Exceptions

Für eine Coordinated-Atomic-Action existieren interne und externe Exceptions. Die internen Exceptions sind vollkommen intern und werden von den Exception-Handlern der Coordinated-Atomic-Action behandelt. Die externen Exceptions sind in der Coordinated-Atomic-Action bekannt, werden aber an die Umgebung (z.B. den Aufrufer oder die umschließende Aktion) signalisiert. Alle Exceptions die innerhalb einer Coordinated-Atomic-Action signalisiert werden, müssen in der Definition dieser deklariert werden. Die entsprechenden Exceptionhandler werden den respektiven Rollen zugeordnet, die die teilnehmenden Threads ausführen. Die externen Exceptions müssen zusätzlich in der Schnittstelle spezifiziert werden und zeigen an, dass obwohl möglicherweise internes Exception-Handling durchgeführt wurde, eine nicht behebbare **exceptional condition** aufgetreten ist und/oder dass nur unvollständige Berechnungen von der Aktion ausgegeben werden können. Eine Coordinated-Atomic-Action die eine weitere Coordinated-Atomic-Action umschließt muss mindestens alle Exceptiontypen dieser in ihrer Definition aufnehmen. Diese Bedingung gilt rekursiv für beliebige Verschachtelungstiefen. Es existieren neben den selbstdefinierten Exceptiontypen noch zwei besondere Exception: die **undo-Exception**, wenn die Aktion abgebrochen und alles Auswirkungen rückgängig gemacht wurden, und die **failure-Exception**, die anzeigt dass die Aktion abgebrochen wurde aber diese Auswirkungen nicht vollständig rückgängig gemacht werden konnten.

10.10.3 Weiterreichungsmodell / Exceptionhandler-Suchpfad

Wenn ein Thread an einer Aktion teilnimmt und eine spezifizierte Rolle ausführt, befindet sich dieser im zugehörigen **Exceptionkontext**. Einige oder alle teilnehmende Threads werden verschachtelten Coordinated-Atomic-Actions beitreten. Das Verschachteln von Coordinated-Atomic-Action führt zum Verschachteln von Exceptionkontexten. Dies erfordert dass jedem teilnehmenden Thread beim Betreten einer weiteren Coordinated-Atomic-Action eine geeignete **handler clause** zugeordnet werden muss. Das Weiterreichen von Exceptions geschieht entlang den verschachtelten Exceptionkontexten und damit der Kette der verschachtelten Coordinated-Atomic-Actions.

Der Mechanismus funktioniert so dass die Rolle innerhalb einer Coordinated-Atomic-Action eine Exception E1 signalisiert, woraufhin die anderen Rollen derselben Coordinated-Atomic-Action über diese Exception informiert werden. Ist das Exceptionhandling innerhalb der Coordinated-Atomic-Action nicht möglich so wird eine neue Exception E2 an die umschließende Coordinated-Atomic-Action signalisiert. Es stehen mindestens zwei Wege zur Verfügung eine Exception von der inneren an die umschließende Coordinated-Atomic-Action zu signalisieren.

führungsrollenbestimmte Exception-Weiterreichung Die erste Variante ist dass eine **Führungsrolle** die Verantwortung trägt, eine vereinbarte Exception an die umschließende Coordinated-Atomic-Action zu signalisieren.

rollenbestimmte Exception-Weiterreichung Die zweite Variante besteht darin, dass jede Rolle verantwortlich ist seine eigene Exception an die umschließende Coordinated-Atomic-Action zu signalisieren. Der Exceptiontyp der Exception, die diese Rollen signalisieren, sollten im Allgemeinen gleich sein, aber können sich unterscheiden. Da eine Coordinated-Atomic-Action in der Lage sein muss gleichzeitig auftretende Exceptions zu behandeln, werden Exceptions die gleichzeitig in der verschachtelten Aktion signalisiert werden, so behandelt, als ob sie gleichzeit in der umschließenden Aktion signalisiert worden wären.

In dem Beispiel 10.10 treten zwei Threads einer CA Aktion bei. Beide Threads kommunizieren miteinadner um das gemeinsame Ziel zu erreichen. Aber während der Ausführung der CA Aktion tritt eine Exception e auf. Der andere Thread wird über die Exception informiert und beide Threads kommunizieren miteinander und führen jeweils einen Exceptionhandler mit Vorwärtsbehebung aus und können die CA Aktion erfolgreich abschließen.

10.10.4 Exception-Kontrollflussmechanismus

Als Exception-Kontrollflussmechanismus wird das **termination model** verwendet.

10.10.5 Externe Objekte

In dem vorgestellten Modell kann die Auswirkung einer Coordinated-Atomic-Action nur durch den veränderten Zustand von externen Objekten festgestellt werden. Wenn eine Exception signalisiert wird müssen die von der Coordinated-Atomic-Action veränderten Objekte in einen konsistenten Zustand überführt werden. Eine Lösung ist sie in ihren

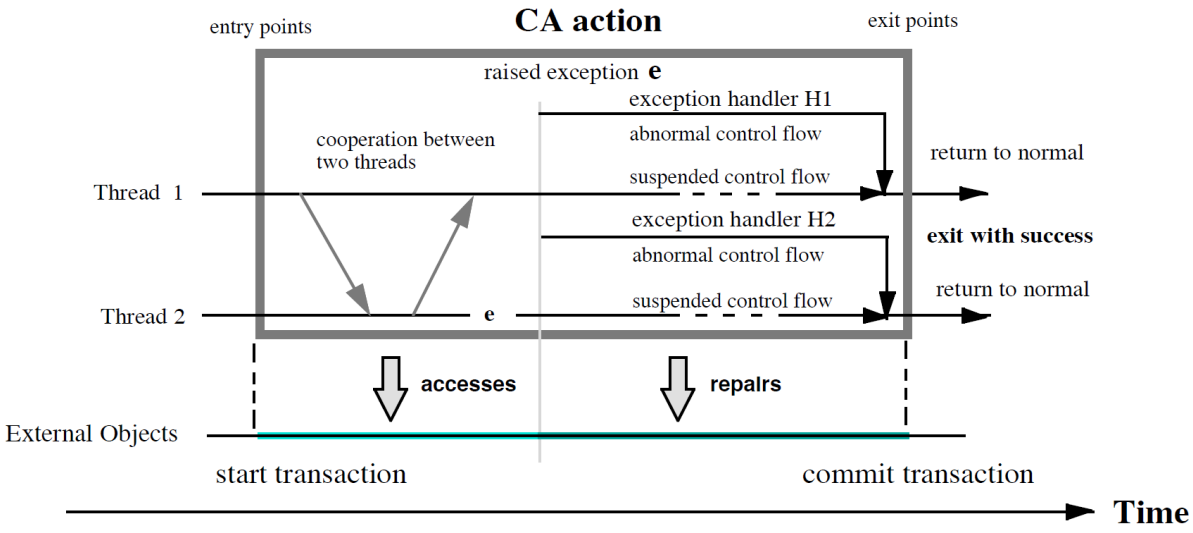


Bild 10.10: Beispielhafte Verwendung einer CA Action

vorherigen Zustand zu überführen (**Rückwärtsbehebung**). Dies istnicht immer möglich aber Exception-Handler können diese oft in einen neuen gültigen Zustand überführen (**Vorwärtsbehebung**). Eine Exception sorgt aber nicht immer dafür dass alle externen Objekte in einen gültigen Zustand überführt werden und die Behebung von den externen Objekten kann auch scheitern. In diesem Fall muss eine **failure-exception** an die umschließende Coordinated-Atomic-Action signalisiert werden.

10.10.6 Exception-Auflösung

Der vorgestellte Ansatz verwendet einen Exception-Graph zur Auflösung der Exceptions.

10.10.7 Bewertung

Nach Campéas et al. [CDUV05] wird bei der Verwendung von **Coordinated Atomic Actions** nicht zwischen Konsistenzsicherung und dem Behandeln von verteilten nebenläufigen Exceptions unterschieden. Es sind zwei unabhängige Anforderungen an ein System die zu trennen sind.

Randbedingungen: Der Algorithmus toleriert kein Ausfall von **nodes** oder **communication lines**.

Für jede Coordinated-Atomic-Action wird angenommen dass jeder teilnehmende Thread alle anderen teilnehmenden Threads kennt und den selben Exception-Graph benutzt der statisch definiert wurde. Jeder Thread verfügt dazu über eine Liste mit den verschachtelten Coordinated-Atomic-Actions die er noch betreten wird. Während der

Ausführung des Algorithmus kann ein Thread sich in dem Zustand Normal, Exceptional oder Suspended befinden. Jeder Thread hat eine Datenstruktur E die alle signalisierten Exceptions und suspendierte Zustände von Threads speichert, und eine Datenstruktur A, die alle Namen der betretenen verschachtelten Coordinated-Atomic-Action speichert. Der Exceptionhandlingmechanismus verfügt über eigenes von der Anwendung-unabhängiges **message passing**.

Exception-Nachricht Mit der Exception-Nachricht sendet ein Thread allen anderen Threads das eine Exception aufgetreten ist.

Suspended-Nachricht Die Suspended-Nachricht wird von einem Thread gesendet, der eine Exception- oder Suspended-Nachricht von einem anderen Thread empfangen hat und daraufhin in den Zustand Suspended übergegangen ist.

commit-Nachricht Die commit-Nachricht wird von einem ausgewählten Thread zu allen anderen Threads in der Coordinated-Atomic-Action gesendet nach dem die Auflösung der Exceptions zu einer Exception E_A erfolgt ist. Daraufhin wird ein geeigneter Exception-Handler für E_A in jedem Thread nach Erhalt der commit-Nachricht ausgeführt.

Wenn in einem Thread das Exceptionhandling fehlschlägt, muss die besondere **undo-Exception** oder **failure-Exception** gesendet werden. Erhalten die anderen Threads die undo-Exception so werden diese auch versuchen ein undo auszuführen. Schlägt dies in mindestens einem Thread, fehl so wird dieser die failure-Exception an die anderen Threads senden. Sendet ein Thread die failure-Exception, so verwerfen die anderen Threads ihre Exceptions und senden auch die failure-Exception an die umschließende Coordinated-Atomic-Action.

Verteilungsgrad Zentral im Falle der Führungsrollenbestimmten Exceptionweiterreichung. Halb-verteilt bzw. Halb-Zentral, wenn die Exception

Kopplungsgrad Innerhalb der CA action ist die Kopplung hoch.

MAS-Offenheit Nein.

generisch Nein.

konkret Ja.

nicht-invasiv Ja.

redundant Nein.

Kontext-Scope Der Scope ist lokal.

dynamische Exceptionbedeutung Nein.

dynamische Erweiterung von Exceptionhandlern Nein.

10.11 Guardian

Das Guardian Model [PE02] [CDUV05] basiert auf einem zeitlich abgestimmten asynchronen Berechnungsmodel, Global Exception Handlers, Trennung von Exceptionhandling, und einem erweiterten fault model. Das Model stellt eine Trennung zwischen Programmausführungs- und Exceptionhandlingpfaden bereit, und erlaubt so modulare high-level Exceptionhandling, die durch wenige Änderungen im Anwendungsprogramm geändert werden können.

Das Grundelement des Guardian Model ist der **Guardian**. Der Guardian ist ein verteilter globaler Exceptionhandler, der einer Aktivität zugeordnet wird. Eine Aktivität ist ist eine Menge kooperativer Prozesse mit Zeitbeschränkungen auf Nachrichtenkommunikation so wie Verarbeitung. Ein kooperativer Prozess wird auch als Teilnehmer bezeichnet. Der Guardian wird logisch in allen Umgebungen der Teilnehmer repliziert. Ein **Guardian Member** ist ein Ko-Prozess in der Umgebung eines Teilnehmers. Die Guardian Member einer Aktivität formen die logische Gruppe, die die globale Abstraktion für den zugehörigen Guardian implementieren. Die Member in dieser Gruppe kommunizieren miteinander über verlässliche Gruppenkommunikationsprimitiven.

Ziele des Guardian Exception Model ist es globale Exceptionhandling-Aktionen zu ermöglichen und das Exceptionhandling zwischen globaler und lokaler Ebene trennen zu können. Wenn ein Teilnehmer eine globale Exception an die anderen Teilnehmer der Aktion senden möchte, so signalisiert er die Exception an seinen **Guardian Member**. Der **Guardian Member** benutzt seine Gruppe um die Exception an die Teilnehmer zu senden. Werden nebenläufige Exceptions signalisiert, so ordnet die Gruppe die Exceptions. Wenn eine globale Exception durch die Gruppe gesendet wird, so verarbeiten alle **Guardian Member** die Exception um die Exceptionhandling-Aktion zu bestimmen.

Das Verarbeiten unterteilt sich in die Schritte: Diagnose, Analyse und Entscheidung. Diagnose erlaubt es einem **Guardian Member** den Zustand des Teilnehmer oder der Laufzeitumgebung abzufragen. Analyse kann durch eine Vielzahl unterschiedlicher Methoden realisiert werden, aber die von Romanovsky et al. [PE02] vorgeschlagene Methode sind Exceptionhandling Patterns, wie beispielsweise der **citizen Ansatz** [PE02]. Jeder **Guardian Member** entscheidet kooperativ bei einer Aktion, und jeder **Guardian**

Member übermittelt die Entscheidung zu seinem lokalen Teilnehmer mittels einer Exception. Die benötigten Aktionen sind suspend, terminate oder query. Ein Teilnehmer muss nicht alle dieser Aktionen ausführen können, aber Exceptionhandler für diese signalisierten Exceptions besitzen.

10.11.0.1 Bedingungen

10.11.0.2 Leistungen

Es wird eine **collective acitivity** eingeführt an denen mehrere Agenten teilnehmen können, die als Teilnehmer bezeichnet werden. Es gibt einen ausgesonderten Teilnehmer der die **Führungsrolle** erhält. Der Bereich des Exception Handling ist die collective activity. Exception Handling auf der Ebene der Teilnehmer wird als global bezeichnet. Aus diesem Grund heißt der von Guardian in Java eingeführte Exceptiontyp GlobalException.

Für die Teilnehmer werden Primitiven zur Verfügung gestellt die das Java Exceptionhandlingsystem nachbilden und zusätzlich Methoden bereitstellen, die lokale Kontexte aktivieren und deaktivieren können. Diese Kontexte sind eine Verallgemeinerung der lexikalischen Kontexte. Ein Guardian Kontext repräsentiert Programmphasen, die von der Blockstruktur der Programmiersprache losgelöst sind. Diese Phasen werden durch {enable|disable}Kontext Befehle realisiert.

10.11.0.3 Exceptionhandling

Nachdem eine globale Exception signalisiert wurde, wird der Führer informiert. Dieser sendet eine Nachricht an alle Teilnehmer in den Zustand suspend überzugehen und synchron Anweisungen zu erwarten. Danach werden alle wartenden globale Exceptions gesammelt.

Wenn alle Teilnehmer bereit sind, berechnet der Führer aus den signalisierten globalen Exceptions eine **concerted exception** die nun in allen Teilnehmern signalisiert wird.

10.11.0.4 Bewertung

Mit Guardian können SaGE-Services, coordinated atomic actions und conversations nachgebaut werden. Guardian skaliert leider schlecht, da alle Teilnehmer einer **collective activity** in den suspended Zustand übergehen müssen wenn eine globale Exception signalisiert wird. Guardian ist deswegen ungeeignet wenn die Teilnehmerzahl groß wird oder einzelne Teilnehmer nicht-unterbrechbare, **high-priority** oder Echtzeit- Servicefunktionalitäten durchführen. Der Schwachpunkt liegt bei der Führungsrolle. Versagt diese oder

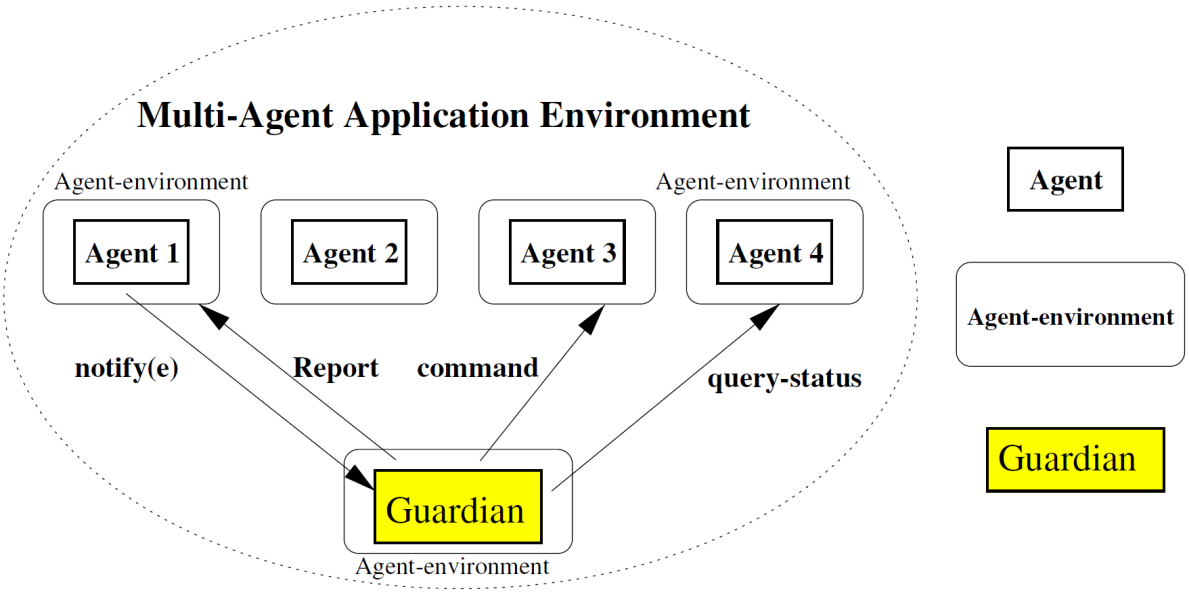


Bild 10.11: Guardian Modell

ist unerreichbar ist das ganze System betroffen. Ein weiterer Punkt ist das Guardian mehr ein Exceptionhandlingframework ist, das einen ermutigt einen Transaktionssystem mit Exceptionhandlingmechanismen zu bauen. Vielleicht besteht hier eine falsche Kopplung zwischen der Exceptionhandlingsemantik und der Ausführungsserialisierung. Diese sollten nicht im selben Paket zusammengefasst sein.

Nach Eric PLaton [PSH07] handelt es sich bei Guardian nur um ein Framework, dass es erlaubt traditionelles Exceptionhandling durchzuführen. Dies führt zwar dazu dass bekannte Mechanismen verwendet werden können, aber dieser Ansatz ist ungeeignet für die Charakteristiken von Agenten.

Der Vorteil von dem Guardian Model ist, dass Exceptions in ähnliche Weise wie in sequentiellen Systemen behandelt werden können.

Verteilungsgrad Nach Eric Platon et al. [PHS06] ist Guardian auf objekt-orientierte und reaktive Agenten beschränkt. Das Guardian Model ist weitestgehend zentralistisch azusgerichtet. Weitestgehend heißt, dass es nicht gänzlich zentralistisch ist, da einer Menge von Guardians nur eine Teilmenge von Agenten zugeordnet werden kann. Es gibt aber kein Koordinierungsmechanismus unter den Guardians.

Kopplungsgrad Nach Eric Platon et al. [PHS06] konzentriert sich der Guardian Ansatz an den Softwareaspekten von Agenten. Die autonomen Entscheidungsmöglichkeiten von Agenten werden ausgelassen. Deswegen geht der Ansatz nicht auf proaktive Agenten ein. Der Kopplungsgrad ist für Agentensysteme zu hoch. Der Guardian

kann dem Agenten Befehle erteilen, beispielsweise zu warten oder einen Task neuzustarten.

MAS-Offenheit Anscheinend liegen nach Eric Platon et al. [PHS06] Einschränkungen vor, darin dass zu viele Vorausetzungen erfüllt werden müssen, und dass Einschränkungen auf der Wahl der Architektur liegen.

generisch Nein.

konkret Ja.

nicht-invasiv Nein. Auf den Zustand der Agenten kann von außen zugegriffen werden.

redundant Nein.

Kontext-Scope Lokal.

dynamische Exceptionbedeutung Nein.

dynamische Erweiterung von Exceptionhandlern Nein.

10.12 Sentinel Agents

Open multi-agent systems [SCG07], kurz MAS, sind dezentralisierte und hochverteilte Systeme, die aus einer Vielzahl von loose-gekoppelten autonomen Agenten bestehen. ”Open” bedeutet dass unabhängig voneinander entwickelte autonome Agenten miteinander interagieren um ihre Ziele zu erreichen.

10.12.0.5 Bedingungen

Für jeden problemlösenden Agenten, existiert ein sentinel agent. Es findet hier eine 1 zu 1 Zuordnung statt.

10.12.0.6 Leistungen

Durch **ACL Nachrichten** können die sentinel agents von den problemlösenden Agenten Informationen abfragen. Die sentinel agents haben Wissen über die Rolle die von ihrem zugeordneten problemlösenden Agenten gespielt wird. Daher wissen sie wie ein Agent von den Rollenverantwortlichkeiten bei einem gegebenen coordination protocol abweichen kann. Die sentinel agents sind als heuritische Klassifikationssysteme implementiert.

10.12.0.7 Exceptionhandling

Der sentinel agent wird von dem problemlösenden Agenten als **Delegate** benutzt, und jegliche Kommunikation des problemlösenden Agenten findet durch den zugeordneten sentinel agent statt. Sendet der problemlösende Agent eine Nachricht durch seinen sentinel agent, so prüft der sentinel agent die Nachricht auf Abweichungen, in dem die Nachricht durch ein Detektionsmodul schickt. Dieses vergleich das Verhalten des Agenten mit dem idealen Rollenverhalten, dies es von dem Modell über das Rollenverhalten erhält. Werden Symptome gefunden, so werden diese an das Diagnostikmodul weitergeleitet, dass eine heuristische Klassifikationsfunktion auf die Domäne, das coordination protocol und die commitment strategy knowlege anwendet, um die unterliegenden Symptome aufzudecken. Werden keine Symptome entdeckt, so wird die Nachricht an den sentinel agent, des Kommunikationspartners weitergeleitet.

10.12.1 Bewertung

Verteilungsgrad Verteilt.

Kopplungsgrad Gering.

MAS-Offenheit Ja.

generisch Ja.

konkret Ja.

nicht-invasiv Ja.

redundant Ja.

Kontext-Scope Lokal.

dynamische Exceptionbedeutung Ja. Bedeutung kann durch knowledge base angepasst werden.

dynamische Erweiterung von Exceptionhandlern Nein.

10.13 Erlang

10.13.0.1 Bedingungen

Bei Erlang [CDUV05] stehen Effizienz, Robustheit und eine Programmiersprache zur Nutzung von Massenparallelrechner im Vordergrund. Erlang erlaubt keine gemeinsamen

Ressourcen zwischen nebenläufigen Entitäten. Kommunikation erfolgt daher ausschließlich durch asynchrones **message passing**.

10.13.0.2 Leistungen

Erlang ist eine **funktionale** Programmiersprache mit einem einfachen Exceptionhandling-mechanismus der ***keine*** **Wiederaufnahme** unterstützt. Es wird das **process** Konstrukt zur Verfügung gestellt dass einen unabhängigen, verteilten sequenziellen Code darstellt. Ein Erlang Programm ist ein Baum von **Supervisor-Prozessen** deren Blattknoten **Worker-Prozesse** sind. Das Erstellen neuer Prozesse und das message passing sind immer explizit.

10.13.0.3 Exceptionhandling

Wird eine Exception in einem Prozess P0 signalisiert, wird nach einem geeigneten lokalen Exception-Handler gesucht. Wird keiner gefunden oder die Behandlung war nicht erfolgreich wird eine **Fehler-Nachricht** an den direkten **Supervisor-Prozess** gesendet und der Prozess P0 wird zerstört. Je nach der Lebendsdauer (permanent, transient oder temporär) wird P0 von dem Supervisor-Prozess komplett neu erstellt und gestartet oder nichts passiert. Zusätzlich dazu können einzelne Prozesse miteinander verbunden werden, sodass wenn ein Prozess fehlschlägt, alle verbundenen Prozesse sich auch zerstören. Um solche Gruppenauslöschungen zu vermeiden wird die Funktionalität eines gutgeschriebenes Erlang Programm so in eine Baumstruktur unterteilt, sodass die Auswirkungen der Zestörung von Unterbäumen oder Clustern von Unterbäumen minimiert wird.

10.13.0.4 Bewertung

Erlang ist effizient und erlaubt auch den Aufbau hochkomplexer Systeme obwohl nur einfache Exceptionhandlingprimitiven angeboten werden.

Verteilungsgrad Verteilt.

Kopplungsgrad Gering.

MAS-Offenheit Nein.

generisch Nein.

konkret Ja.

nicht-invasiv Ja.

redundant Ja.

Kontext-Scope Lokal.

dynamische Exceptionbedeutung Nein.

dynamische Erweiterung von Exceptionhandlern Nein.

10.14 Ambient Conversation

An Ambient Conversation wird von Mostinckx et al [MDB+06] geforscht.

10.14.0.5 Bedingungen

Verbindungstrennungen sind die Regel und stellen keine exceptional conditon dar. Dieser Ansatz geht von drahtlosen Netzwerkverbindungen aus, von denen keine stabile Verbindung erwartet werden kann, aufgrund der Reichweitenbeschränkung des Netzwerks.

Es werden ausschließlich **Ambient Resources** verwendet. Ein Mechanismus sorgt für die dynamische Verwaltung der Menge der Ambient Resources. Dies ist notwendig da neue Resourcen dynamisch (nicht) verfügbar werden wenn sporadische Verbindungstrennungen folgen.

Autonomie der einzelnen Teilnehmer wird gefordert. Ohne zentrale Einheit muss jeder Teilnehmer in der Lage sein andere Teilnehmer zu finden und Interaktionen zu koordinieren. Exceptionhandling muss selbstständig durchzuführen sein falls ein anderer Teilnehmer nicht verfügbar wird, ohne bis zu dessen Rückkehr blockieren zu müssen. Aus diesem Grund findet nur asynchrone Nachrichtenübertragung statt. Die Nachrichten werden von jedem Actor serialisier verarbeitet um race conditions zu vermeiden. Nach der Wiederherstellung werden alle Nachrichten die aufgrund der Trennung nicht gesendet werden konnten vom Sender gesendet.

classless object model: In Ambient werden keine Klassen definiert und diese zur Laufzeit davon Objekten instanziiert. Programmcode (entspricht der Klasse) und Daten werden zusammen direkt in ein Objekt gepackt. Dies erleichtert die Verteilung von Objekten über ein Netzwerk.

10.14.0.6 Leistungen

Asynchrone Nachrichten haben keinen Rückgabewert. Deswegen werden **Callback** Methoden benötigt, die Ergebnisse oder Exception zurückliefern. In Ambient werden dafür

Futures benutzt, Platzhalter-Objekte die nach dem Senden einer asynchronen Nachricht unmittelbar zurückgegeben werden. Nachdem das Ergebnis berechnet wurde ist das Future gelöst und der Wert kann frei verwendet werden. Wird das Future benutzt bevor es gelöst wurde, darf es in Ambient nicht blockieren, da dadurch die Autonomie verloren gehen würde. Als Vorlage dienen die Futures aus der **Programmiersprache E**. Nachrichten zu den Futures werden gepuffert und atomatisch zu dem aktuellen Ergebnis weitergeleitet sobald das Future gelöst ist. Die Nachrichten zu den Futures müssen asynchron sein und liefern damit erneut ein Future zurück. Die Lösung dieses Futures setzt die Lösung des vorherigen Futures voraus. Deshalb wird dies auch **Future-Pipelining** genannt. Damit kann eine Aufrufkette von asynchronen Nachrichten aufgebaut werden, ohne dass die Ergebnisse bereits berechnet sein müssen.

Zu den Leistungen gehören folgende Eigenschaften:

- Erleichtert das Weiterarbeiten von zusammenarbeitenden Prozessen um ihre Autonomität aufrechtzuerhalten. Dies kann zu Inkonsistenzen führen weswegen jeder Prozess eine explizite Darstellung seiner Kommunikationsdetails speichert.
- Namensfindung statt Adressen
- bei Verbindungstrennung und Wiederaufnahme sollen lokale Änderungen synchronisiert werden
- Zusammenarbeit wird bei Verbindungstrennung nicht beendet

10.14.0.7 Exceptionhandling

Das Ambient Conversation Modell besteht aus folgenden vier Sprachkonstrukten.

when-catch ermöglicht asynchrone Exception Weiterreichung auf der Granularitätsstufe eines asynchronen Aufrufs.

group-resolve ermöglicht die Verwendung des Multiple-Distributed-Exceptions-Single-Handler Schemas. Die signalisierten Exceptions von mehreren asynchronen Aufrufen können so von einem Exception-Handler behandelt werden, da die nebenläufigen Exceptions zu einer einzigen concerted exception aufgelöst werden.

conversation ermöglicht die Verwendung von collaborative exception handling. Eine Exception die von einem Teilnehmer signalisiert wird, wird von allen Teilnehmern behandelt.

when-catch-due realisiert loosely-coupled exception handling.

10.14.0.7.1 Lokales Exceptionhandling

Durch ein try-catch Konstrukt kann lokales Exception-Handling stattfinden.

10.14.0.7.2 single-asynchron-exception Single-Exception-Handling

Kann der Teilnehmer der die Exception signalisiert hat, diese nicht behandeln, so wird die Exception anstelle des Wertes des Futures zurückgegeben. Bei Future-Pipelining wird die Exception durch alle Future-Objekte durchgegeben und jedes Future in der Kette gibt anstelle eines berechneten Wertes das Exception-Objekt zurück. Durch das when-catch Konstrukt 10.12 können Kontexte in der Future-Pipeline gebildet werden. Der Befehl **when** wartet auf das Ergebnis des Futures und leitet dieses in den Bezeichner der mit dem **becomes** Befehl angegeben wird. Der Code in den darauffolgenden eckigen Klammern wird ausgeführt wenn das Ergebnis des Futures gelöst wurde. Wird eine Exception signalisiert so wird nach einer passenden catch-clause die dem **when-becomes** Block folgt gesucht. Kann die Exception behandelt werden so erhält das nächste Future in der Future-Pipeline das korrigierte Ergbenis, ansonsten wird die Exception weitergereicht.

```
1 when (Future1) becomes(Result1)
2 {
3  // Programmcode, der ausgeführt wird, wenn das Future-Objekt mit der
       Bezeichnung Future1 gelöst wurde
4  // der berechnete Wert steht unter der Bezeichnung Result1 zur
       Verfügung
5 }
6 catch ( ... )
7 {
8   // Programmcode zum Behandeln der Exception
9 }
```

Bild 10.12: When-Becomes Konstrukt in Ambient

10.14.0.7.3 multiple-asynchron-exceptions synchronised Single-Exception-Handling

Um Exceptions zu behandeln, die nicht in einer Future-Pipeline auftreten, gibt es das **group-resolve** Konstrukt 10.13. Das Ende einer group-clause wartet so lange bis alle Futures die innerhalb der group-clause erstellt wurden, auch gelöst wurden. Wenn Exceptions signalisiert wurden, wird die **resolve clause** ausgeführt und ihr ein Array aller nebenläufig signalisierten Exceptions übergeben. Wenn die Exceptions toleriert werden können, gibt die resolve clause ein Wert zurück, ansonsten signalisiert sie eine **concerted exception**.

```
method Methode1(sharedObject1)
{
    group
    {
        for participant in AllParticipants
        {
            participant#merge(sharedObject);
        }
    }
    resolve( concurrentExceptions)
    {
        // toleriere die Exception(s), oder berechne und signalisiere
           eine concerted exception
    }
}
```

Bild 10.13: Group-Resolve Konstrukt in Ambient

10.14.0.7.4 single-asynchron-exception collaborative-Exception-Handling

Das **conversation** Sprachkonstrukt stellt ein Mechanismus bereit Exceptions an alle Teilnehmer weiterzureichen, die zusammenarbeiten und an der conversation teilnehmen. Wenn eine conversation erstellt wird, wird dieser eine Liste aller Teilnehmer übergeben. Jeder Teilnehmer der conversation hat eine **propagate** Methode, an die ein Exception-Objekt übergeben werden kann, das zu allen anderen Teilnehmern weitergereicht wird. In der conversation können when-catch Konstrukte benutzt werden um auf die Exceptions angemessen reagieren zu können. Damit die conversation autonom bleibt, besitzt jeder Teilnehmer ein eigenes Replikat der conversation, das durch den Aufruf der Methode **startConversation** zurückgegeben wird. Eine Exception wird so zu den anderen Teilnehmern weitergereicht, indem die **propagate** Methode des lokalen Replikat aufgerufen wird.

10.14.0.7.5 loosely-coupled Exception-Handling als zusätzliche Dimension

Das Ziel bei losegekoppeltem Exception-Handling ist es die Kopplung unter den Teilnehmern möglichst gering zu halten. Kopplung entsteht hierbei durch das gegenseitige Senden von Nachrichten. Da die Nachrichten asynchron gesendet werden, kann kein Teilnehmer blockieren, aber es kann dazu kommen dass ein Teilnehmer keinen Fortschritt mehr macht. Es ist möglichdiese Kopplung ”weich” zu machen, indem eine Zeitrestriktion auf einen **guarded-block** gelegt wird. Wird die Zeitdauer überschritten so wird der Kontrollfluss in einen speziellen Exception-Handler Block, der **due-clause** geleitet. In Ambient steht dafür das **when-catch-due** 10.14 Sprachkonstrukt zur Verfügung. Der

due-clause kann in Millisekunden die Zeitdauer übergeben werden, die auf einen anderen Teilnehmer zur Berechnung eines Ergebnisses gewartet wird.

```
1  when (objekt1#methode1) becomes(Result1)
2  {
3   // Programmcode, der ausgeführt wird, wenn der Methodenaufruf
       erfolgreich war
4   // und das Future-Objekt das Ergebnis in Result1 gelöst hat
5  }
6  catch ( ... )
7  {
8    // Programmcode zum Behandeln von signalisierten Exceptions
9  }
10 due (MAX_TIMEOUT)
11 {
12  // Programmcode, der ausgeführt wird wenn die in Klammern angegebene
       Zeitdauer
13  // in Millisekunden überschritten wurde
14 }
```

Bild 10.14: When-Catch-Due Konstrukt in Ambient

10.14.0.8 Bewertung

Die Sprachkonstrukte des Ambient Conversation Modells müssen kombiniert werden um effektiv Exceptions behandeln zu können. Ausdruckstärkere Konstrukte wären wünschenwert, müssen aber erst aus Abstraktion von Exceptionhandlingstrategien gewonnen werden. Für das **conversation** Sprachkonstrukt existiert noch kein konkreter Entwurf wie nebenläufig ausgeführte Exception-Handler gemeinsam kommunizieren könnten.

Verteilungsgrad Verteilt.

Kopplungsgrad Gering.

MAS-Offenheit Ja.

generisch Nein.

konkret Ja.

nicht-invasiv Ja.

redundant Nein.

Kontext-Scope Lokal.

dynamische Exceptionbedeutung Nein.

dynamische Erweiterung von Exceptionhandlern Nein.

10.15 Commitment Protocols

Commitment Protocols [MS05] sind ein Forschungsprojekt von Mallya et al.

10.15.1 Bedingungen

Ein **commitment protocol** wird als eine Menge von Berechnungen modelliert. Jede Berechnung repräsentiert eine erlaubte Interaktion und gibt die Beteiligung von den Teilnehmern an. Prozesse werden durch Protokolle dargestellt. Ein Protokoll ist eine Spezifikation von Interaktionen zwischen autonomen Agenten. Protokolle tragen dazu bei dass Exceptions auf einer wiederverwendbaren Weise modelliert und behandelt werden können. Mallya et al. [MS04] zeigen wie commitment protocols erstellt werden.

Eine Exception erfordert besondere Verarbeitung, so dass Teilnehmer an dem Protokoll, die Verluste minimieren können, die aufgrund der Exception auftreten können. Das Behandeln von Exceptions setzt voraus dass das **domain knowledge** in die Entscheidungsfindung miteinbezogen wird. Eine bestimmte Exception hat für Teilnehmer desselben Protokolls in unterschiedlichen Domänen eine andere Gewichtung. Eine Verzögerungs-Exception hat so in einem Prozess zur Zahlung für ein Hotelzimmer, eine geringere Gewichtung, als in einem Versandprozess von lebensrettenden Medikamenten. Exceptions müssen somit in verschiedenen Domänen unterschiedlich behandelt werden. Es ist nötig eine Abstraktion zu schaffen, die domän-unabhängige Beziehungen zwischen Teilnehmern heraus faktorisieren kann.

Unerwartete Exceptions erfordern eine Änderung des Prozessmodells zur Laufzeit. Es wird eine Abstraktion benötigt, die es erlaubt **pragmatic exceptions** einzuführen, die es erlauben über Bedeutungen in einem Kontext nachzudenken.

10.15.2 Leistungen

Exceptions werden durch Präferenzstrukturen modelliert, die auf die Menge der Berechnungen angewendet werden. Die Präferenzstrukturen bestimmten statisch wie erwartete Exceptions behandelt werden. Für unerwartete Exceptions müssen die Strukturen dynamisch erweitert werden. Dazu können Exceptionhandler dynamisch einem Protokoll hinzugefügt werden.

Dieser Ansatz bietet ein Framework um Exceptions durch den Einsatz von commitment protocols generisch über verschiedene Domänen hinweg zu modellieren. Unerwartete Exceptions können zur Laufzeit durch Änderung des Prozessmodells behandelt werden.

10.15.3 Exceptionhandling

Da Protokolle, die einen Prozess darstellen, das Ziel der Teilnehmer an dem Prozess beschreiben, bieten sie die Basis um Exceptions zu modellieren und zu behandeln. Protokolle definieren den Kontext in dem Exceptions behandelt werden können, und werden beispielsweise in der Sprache OWL-P von Desai et al. [DMCS06] deklarativ spezifiziert. Die Spezifikation beinhaltet Rollendefinition, die Nachrichten, die ausgetauscht werden und Regeln, die die Menge der **runs** (run: Sequenz von Zuständen durch ein commitment protocol) des Protokolls beschränken indem Reihenfolge, Datenfluss und anderen Einschränkungen festgelegt werden.

Diese Protkolle können als Transitionssysteme dargestellt werden und generieren Berechnungen oder runs, die eine Sequenz von Zuständen beschreiben, die alle validen Ausführungen eines Protokolls beschreiben. Zustände sind von Vorbedingungen abhängig und Zustandsänderungen werden durch Nachrichten verursacht, die unter den Teilnehmern ausgetauscht werden.

Für erwartete Exceptions wird eine Präferenzstruktur auf die runs gelegt, wie beispielsweise welche runs anderen gegenüber bevorzugt sind. Dazu spezifiziert jedes Protokoll eine Hierachie über die bevorzugten runs. Diese Hierachie muss nicht vollständig sein, sodass in einigen Fällen keine Aussage getroffen werden kann, welcher run zu bevorzugen ist. Je nachdem in welcher Domäne das Protokoll genutzt werden soll, kann eine Menge von runs durch die Präferenzstruktur als außergewöhnlich gekennzeichnet werden. Für unerwartete Exceptions wird **protocol splicing** mittels eines Vereinigungsoperators durchgeführt.

10.15.4 Bewertung

Für erwartete Exceptions müssen mehr Abweichungen von der normalen Ausführung der Prozesse enkodiert werden. Dies führt zu einem Rückgang von Ergebnissen in der Prozessautomatisierung.

erwartete Exceptions: Durch den Ansatz der Präferenzstrukturen sind Exception nicht nur Trigger, die die Ausführung eines Exceptionhandler nach sich ziehen, sondern beziehen sich auf das Protokoll als Ganzes. Benutzer eines Protokolls können für ihre Domäne definieren, was für sie als Exception, und was nicht als Exception betrachtet werden soll.

unerwartete Exceptions: Für unerwartete Exceptions müssen die automatisierten Prozesse unterbrochen werden, damit Benutzer die Prozesse begradigen können. Dazu

kann der Benutzer eine Verfeinerung des Protokolls vornehmen. Um Exceptionhandlers zur Laufzeit zu trennen, ist eine Bibliothek von Exceptionhandlern und eine Durchsuchung dieser erforderlich.

In Business Process Execution Language (BPEL) [ACD+03] und Web Services Choreography Description Language (WS-CDSL) [KBR+04] werden nicht die Konsequenzen der Autonomie von ihren Teilnehmern festgehalten, da ihnen eine Beschreibung für commitments fehlt.

Verteilungsgrad Verteilt.

Kopplungsgrad Gering.

MAS-Offenheit Es handelt sich um ein offenes System.

generisch Ja.

konkret Keine Aussage möglich.

nicht-invasiv Keine Aussage möglich.

redundant Keine Aussage möglich.

Kontext-Scope Keine Aussage möglich.

dynamische Exceptionbedeutung Ja.

dynamische Erweiterung von Exceptionhandlern Ja.

10.16 Resümee

Die Semantik des Exceptionhandling von JCilk für nebenläufige Prozesse ist zum Programmieren und zur Verifikation sehr gut durchdacht. Die Einfachheit der Benutzbarkeit und die leicht verständliche Semantik wird von den anderen Ansätzen nur schwer übertroffen. Leider ist der Ansatz nur mit Threads implementiert. Es wäre interessant das semantische Konzept auch auf verteilte heterogene System angewendet zu sehen. In den protokollbasierten Exceptionhandlingmodellen wie bei SaGE und den Commitment Protocols liegt viel Potenzial durch das hohe Maß an angebotener Flexibilität. Vor allem die Verwendung einer knowledge base und von intelligenten resolution functions versprechen sehr spezifische Möglichkeiten des Exceptionhandling. Das Exceptionhandlingmodell von Ambient ist das einzige in der Literatur gesichtete Modell, das sämtliche Eigenschaften der aufgestellten Klassifikation anbietet und im höchsten Grad auf lose gekoppelte Systeme ausgerichtet ist. Es steht gänzlich dem Aufbau von dem bewährten

Modell von Erlang entgegen. Erlang besticht durch seine Einfachheit und Robustheit. Es werden weitere Untersuchungen nötig sein, ob die Mächtigkeit des Ansatzes von Ambient im Einsatz zum Tragen kommt.

11 Ausblick

Im Rahmen dieser Arbeit wurde Exceptionhandling grundlegend von allen Seiten beleuchtet. Während bei der Aufstellung der Klassifikation des Exceptionhandling für lokale Systeme noch in detailreicher Tiefe auf einzelne Aspekte eingegangen werden konnte, war es umso schwieriger für verteilte Systeme diese Detailtiefe beizubehalten. In verteilten Systemen spielen hingegen die semantischen Erweiterungen auf der Ebene der Kooperation der einzelnen Systemen zu einem kollaborativen Ganzen eine höhere Rolle. Vor allem die Abgrenzung der Exceptionkontexte der einzelnen Systeme, gegenüber den anderen Systemen, sowie die Frage der Koordination und der Autonomie der einzelnen Systeme. Es war zu erkennen, dass die einzelnen Forschergruppen ihre Projekte der Öffentlichkeit in Publikationen zugänglich machen, aber nur die wesentlichen Konzepte in diesen darlegen - leider aber nicht auf alle Aspekte die von Interesse sind, eingehen. So hoffe ich, dass diese Arbeit dazu beiträgt eine Klassifikation bereitzustellen, deren Gebrauch in jeder Vorstellung eines neuen Exceptionhandlingmodells dazu führen würde, dass die einzelnen Modelle miteinander vergleichbar wären und je nach Aufgabenstellung für ein zu entwerfendes Software-Projekt das beste Exceptionhandlingmodell ausgewählt werden könnte.

Im folgenden werden noch Fragen aufgeworfen, deren Beantwortung Exceptionhandling in verteilten Systemen großen Gewinn bringen würden:

Das Thema Sicherheit [MSD05] ist ein relevanter Aspekt, der zunehmend nach Reifung der einzelnen Exceptionhandlingmodelle an Beachtung gewinnen muss. Exception verletzten beispielsweise in Server/Client Architekturen teilweise das Geheimnisprinzip, da an den Client signalisierte Exception oft Implementierungsdetails enthalten. Beispielsweise verrät eine SqlException, dass ein Sql-Befehl und wahrscheinlich eine relationale Datenbank verwendet wird.

Liegt ein Exceptionhandlingmodell zugrunde, indem durch das Signalisieren einer Exception, andere Teilnehmer in den Zustand suspended übergehen, oder koordiniertes Exceptionhandling betreiben, so wurde bisher auf folgende mögliche Schwachstelle nicht eingegangen. In diesen Systemen finden sich meist heterogene Systeme. Wenn nun ein Teilnehmer einen Defekt hat und dauernd Exceptions an die umliegenden Teilnehmer

signalisiert, so kann das gesamte verteilte System in seiner Leistung massiv beeinträchtigt werden. Hier fehlt nach Meinung des Autors, ein ähnlicher Mechanismus, der auf Hardwareebene für einen Busfehler existiert. Dieses System müsste ausgeschaltet werden oder Exceptions müssten von dieser Quelle ignoriert werden. Bisher wurde dies nicht in der Architektur der Exceptionmodelle berücksichtigt.

In den Exceptionhandlingmodellen für verteilte Systeme ließ sich keine Information finden, wie das Weiterreichen der asynchronen Exceptions tatsächlich auf Implementierungsebene stattfindet. Findet kein explizites Polling vom Programmierer statt, sondern implizit durch die Programmiersprache, Laufzeitumgebung oder das Framework, so ist hier viel Potenzial zur Leistungsoptimierung gegeben. Wird zu selten polling betrieben, werden Exception verpasst oder zu spät behandelt. Wird es zu oft betrieben, so fehlen CPU-Zyklen für tatsächliche Berechnungen. Eine Verifikation oder Berechnung der besten Einstellungen wird durch die Komplexität aufgrund der undurchschaubaren Abhängigkeiten nicht gegeben sein. Eine dynamische Anpassung der Frequenz des Polling analog zu der Größe des Übertragungsrahmens beim TCP/IP-Protokoll wäre erstrebenswert. Werden lange Zeit keine Exceptions signalisiert, so wird die Frequenz des Polling sehr niedrig. Sobald eine Exception signalisiert wird, wird die höchste Frequenz für das Polling eingestellt und wird entsprechend der Auftrittswahrscheinlichkeit einer weiteren asynchronen Exception angepasst.

In diesem Zusammenhang ist auch ein Mechanismus für den angesprochenen Zeitaspekt im Ambient Exceptionhandlingmodell weiterzuverfolgen. Mostinckx et al. sind der Auffassung, dass der Zeitaspekt nötig ist um höchstmögliche Autonomie zu gewähren und gleichzeitig sicherzustellen, dass ein Fortschritt in der Berechnung des Programms gegeben ist, wenn die weitere Berechnung nur nach Erhalt eines Ergebnisses eines anderen Teilnehmers fortgesetzt werden kann. Die Zeitspanne, die gewartet werden soll, bevor von einer Unerreichbarkeit des anderen Teilnehmers ausgegangen werden kann, und eine erneute Anfrage abgeschickt werden muss, ist nicht trivial zu bestimmen. Hier wäre weitere Forschung erstrebenswert.

Auf einem einzelnen lokalen System lassen sich für synchrone Exceptions, beispielsweise durch SPEC Benchmark-Test die Exceptionhandlingsysteme vergleichen. Doch selbst für lokale Systeme konnten bisher in der Literatur keine einstimmig befürworteten Benchmark-Test gefunden werden, die aus einer Menge von Projekten zusammengesetzt wurden, die nach dem ”exception-driven” Grundsatz programmiert worden sind. Umso schwieriger ist es die Exceptionhandlingmodelle für verteilte Systeme in ihrer Leistung miteinadner zu vergleichen.

Appendices

Literaturverzeichnis

[Abr00] David Abrahams. Exception-safety in generic components. In Mehdi Jazayeri, Rüdiger Loos, and David Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 69–79. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-39953-4_6.

[ACD+03] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business processexecution language for web services, (bpel 1.1). Technical report, BEA Systems, International Business Machines Corporation,Microsoft Corporation, SAP AG, Siebel Systems., 2003.

[Act] Activiti. Activiti user guide. http://activiti.org.

[B.93] Stroustrup B. A history of c++. In *SIGPLAN Second History of Programming Languages Conference (HOPL-II)*, volume 28 of *ACM SIGPLAN Notices*, 03 1993.

[BKvSY80] D.M. Berry, R.A. Kemmerer, A. von Staa, and S. Yemini. Toward modular verifiable exception handling. In *Computer Languages*, 1980.

[BM00] Peter A. Buhr and W. Y. Russell Mok. Advanced exception handling mechanisms. *IEEE Trans. Softw. Eng.*, 26(9):820–836, September 2000.

[BMZ92] Peter A. Buhr, Hamish I. Macdonald, and C. Robert Zarnke. Synchronous and asynchronous handling of abnormal events in the ?system. *Software: Practice and Experience*, 22(9):735–776, 1992.

[c2.a] c2.com. Design with exceptions. http://c2.com.

[c2.b] c2.com. Homogenize exceptions. http://c2.com.

[c2.c] c2.com. Refine exceptions. http://c2.com.

[c2.d] c2.com. Tidy up before throwing. http://c2.com.

[Can07] Brett Cannon. How far to go with cleaning up exceptions. http://mail.python.org, 3 2007.

[CC05] Denis Caromel and Guillaume Chazarain. Trobust exception handling in an asynchronous environment. In *Proceedings of ECOOP'05 Workshop on Exception Handling in Object-Oriented Systems*, pages 14–26, 2005.

[CDUV05] Aurélien Campéas, Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Distributed exception handling: Ideas, lessons and issues with recent exception handling systems. In Nicolas Guelfi, editor, *Rapid Integration of Software Engineering Techniques*, volume 3475 of *Lecture Notes in Computer Science*, pages 605–605. Springer Berlin / Heidelberg, 2005. 10.1007/11423331_8.

[cpla] cplusplus.com. C++ tutorials. http://www.cplusplus.com.

[cplb] cplusplus.com. Reference for c++ <exception> header. http://www.cplusplus.com.

[cpp] cppreference.com. C++ reference. http://en.cppreference.com.

[CR86] Roy H. Campbell and Brian Randell. Error recovery in asynchronous systems. *IEEE Trans. Softw. Eng.*, 12(8):811–826, August 1986.

[DACL07] John S. Danaher, I-Ting Angelina, Lee Charles, and E. Leiserson. Programming with exceptions in jcilk. Technical report, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, 2007.

[dD00] Christophe de Dinechin. C++ exception handling. Technical report, IEEE, 2000.

[dLR01] R. de Lemos and A. Romanovsky. Exception handling in the software lifecycle. In *International Journal of Computer Systems Science and Engineering, 16(2):167 – 181*, 2001.

[DMCS06] Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Owl-p: a methodology for business process development. In *Proceedings of the 7th international conference on Agent-Oriented Information Systems III*, AOIS'05, pages 79–94, Berlin, Heidelberg, 2006. Springer-Verlag.

[DMS03] Romonvosky Di Marzo Serugendo. Using exceptioon handling for fault-tolerance in mobile coordination-based environments, 2003. In ECOOP Workshop on Exception Handling on Object Oriented Systems: toward Emerging Application Areas and New Programming Paradigms.

[Doca] Oracle Docs. Catching multiple exception types and rethrowing exceptions with improved type checking. http://docs.oracle.com.

[Docb] Oracle Docs. Java se7 docs api. http://docs.oracle.com.

[Docc] Oracle Docs. The java tutorials. http://docs.oracle.com.

[Docd] Oracle Docs. Javase 7 docs api - runtimeexception. http://docs.oracle.com.

[Doce] Python Docs. 27.1. sys — system-specific parameters and functions. http://docs.python.org.

[Docf] Python Docs. 5. built-in exceptions. http://docs.python.org.

[Docg] Python Docs. Design and history faq. http://docs.python.org.

[Doch] Python Docs. Initialization, finalization, and threads. http://docs.python.org.

[Doci] Python Docs. The python tutorial. http://docs.python.org.

[Docj] Python Docs. Standard exception classes in python 1.5. http://www.python.org.

[DUV06] Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Advanced topics in exception handling techniques. In Christophe Dony, Jørgen Lindskov Knudsen, Alexander Romanovsky, and Anand Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, chapter Exception handling and asynchronous active objects: issues and proposal, pages 81–100. Springer-Verlag, Berlin, Heidelberg, 2006.

[EL95] Johann Eder and Walter Liebhart. The workflow activity model wamo. In *Proceedings of the 3rd international conference on Cooperative Information Systems (CoopIs*, pages 87–98, 1995.

[Geh92a] N. H. Gehani. Exceptional c or c with exceptions. *Software: Practice and Experience*, 22(10):827–848, 1992.

[Geh92b] N. H. Gehani. Exceptional c or c with exceptions. *Software: Practice and Experience*, 22(10):827–848, 1992.

[Gooa] Phil Goodwin. Catch what you can handle. C2M.

[Goob] Phil Goodwin. Let exceptions propagate. C2M.

[Goo75] John B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, December 1975.

[Groa] William Grosso. http://c2.com.

[Grob] William Grosso. Don't throw generic exceptions. http://c2.com. C2M.

[GSK11] M. Gregoire, N. Solter, and S. Kleper. *Professional C++*. John Wiley & Sons, 2011.

[GvR11] Fred L. Drake Guido van Rossum. *Python Frequently Asked Questions*. Python Software Foundation, 2011.

[Häg96] Staffan Hägg. A sentinel approach to fault handling in multi-agent systems. In *Proceedings of the Second Australian Workshop on Distributed AI, in conjunction with Fourth Pacific Rim International Conference on Artificial Intelligence (PRICAI'96*, 1996.

[IBM] IBM. Handling exceptions and faults. IBM.

[Ins] Inspirel. Exception chaining for c++. http://www.inspirel.com.

[Iss01] Valérie Issarny. Concurrent exception handling. In Alexander Romanovsky, Christophe Dony, Jørgen Knudsen, and Anand Tripathi, editors, *Advances in Exception Handling Techniques*, volume 2022 of *Lecture Notes in Computer Science*, pages 111–127. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45407-1_7.

[KBR+04] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web service choreography description language (ws-cdl 1.0). Technical report, W3C, 2004.

[KD99] Mark Klein and Chrysanthos Dellarocas. Exception handling in agent systems. In *Proceedings of the third annual conference on Autonomous Agents*, AGENTS '99, pages 62–68, New York, NY, USA, 1999. ACM.

[Kie01] Jörg Kienzle. *Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming*. PhD thesis, Software Engineering Laboratory, Computer Science Departement, EPFL, 1015 Lausanne, Switzerland, Software Engineering Laboratory, Computer Science Departement, EPFL, 1015 Lausanne, Switzerland, 2001.

[Kie08] Jörg Kienzle. On exceptions and the software development life cycle. In *Proceedings of the 4th international workshop on Exception handling*, WEH '08, pages 32–38, New York, NY, USA, 2008. ACM.

[Knu87] J. L. Knudsen. Better exception-handling in block-structured systems. *IEEE Softw.*, 4(3):40–49, May 1987.

[Knu00] Jorgen Lindskov Knudsen. Exception handling versus fault tolerance. Technical report, Aarhus University, 2000.

[Kra] David Krauss. Should exceptions be chained in c++? http://stackoverflow.com.

[KRAD03] Mark Klein, Juan-Antonio Rodriguez-Aguilar, and Chrysanthos Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):179–189, July 2003.

[Kuca] A.M. Kuchling. What's new in python 2.1. http://docs.python.org.

[Kucb] A.M. Kuchling. What's new in python 2.2. http://docs.python.org.

[Kucc] A.M. Kuchling. What's new in python 2.5. http://docs.python.org.

[Kuc06] A.M. Kuchling. Python 1.4 update. http://www.amk.ca, 12 2006.

[Lac91] Serge Lacourte. Exceptions in guide, an object-oriented language for distributed applications. In Pierre America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 268–287. Springer Berlin / Heidelberg, 1991. 10.1007/BFb0057027.

[Lan98] Danny Lange. Mobile objects and mobile agents: The future of distributed computing? In Eric Jul, editor, *ECOOP'98 — Object-Oriented Programming*,

volume 1445 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0054084.

[Lev77] Roy Levin. *Program structures for exceptional condition handling.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1977. AAI7804937.

[LS79] B. H. Liskov and A. Snyder. Exception handling in clu. *IEEE Trans. Softw. Eng.*, 5(6):546–558, November 1979.

[LS98] Jun Lang and David B. Stewart. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Trans. Program. Lang. Syst.*, 20(2):274–301, March 1998.

[LS10] Giovanni Lagorio and Marco Servetto. Strong exception-safety for java-like languages. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, FTFJP '10, pages 3:1–3:7, New York, NY, USA, 2010. ACM.

[LYK+00] Seungll Lee, Byung-Sun Yang, Suhyun Kim, Seongbae Park, Soo-Mook Moon, Kemal Ebcioğlu, and Erik Altman. Efficient java exception handling in just-in-time compilation. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 1–8, New York, NY, USA, 2000. ACM.

[MDB+06] Stijn Mostinckx, Jessie Dedecker, Elisa Gonzalez Boix, Tom Van Cutsem, and Wolfgang De Meuter. Ambient-oriented exception handling. In Christophe Dony, Jørgen Lindskov Knudsen, Alexander Romanovsky, and Anand Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, pages 141–160. Springer-Verlag, Berlin, Heidelberg, 2006.

[Mey92] Bertrand Meyer. *Eiffel: the language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[MMS78] J G Mitchell, W Maybury, and R E Sweet. *Mesa language manual.* Xerox Res. Cent., Palo Alto, CA, 1978.

[MS04] Ashok U. Mallya and Munindar P. Singh. A semantic approach for designing commitment protocols. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '04, pages 1364–1365, Washington, DC, USA, 2004. IEEE Computer Society.

[MS05] Ashok U. Mallya and Munindar P. Singh. Modeling exceptions via commitment protocols. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 122–129, New York, NY, USA, 2005. ACM.

[MSDa] MSDN. Eh (exception handling model). http://msdn.microsoft.com.

[MSDb] MSDN. Exception handling in visual c++. http://msdn.microsoft.com.

[MSDc] MSDN. How to: Handle exceptions thrown by a web service method. MSDN.

[MSDd] MSDN. How to: Throw exceptions from a web service created using asp.net. MSDN.

[MSDe] MSDN. Transporting exceptions between threads. http://msdn.microsoft.com.

[MSD05] MSDN. Web service security. Technical report, Microsoft Corporation., 2005.

[MSR77] P. M. Melliar-Smith and B. Randell. Software reliability: The role of programmed exception handling. In *Proceedings of an ACM conference on Language design for reliable software*, pages 95–100, New York, NY, USA, 1977. ACM.

[MT97] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In *In Object-Oriented Programming, 11th European Conference (ECOOP*, pages 85–103. Springer-Verlag, 1997.

[MT02] R. Miller and A. Tripathi. The guardian model for exception handling in distributed systems. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pages 304 – 313, 2002.

[NC82] S. Dulli N. Cocco. A mechanism for exception handling and its verification rules. In *Computer Languages*, 1982.

[OMG11] OMG. Business process model and notation (bpmn) version 2.0. Technical report, OMG, 2011.

[Ora] Oracle. The java language specification, java se 7 edition. http://docs.oracle.com.

[PE02] Alexander Romanovsky Paul Ezhilchelvan. Exception handling in timed asynchronous systems. In Paul Ezhilchelvan Alexander Romanovsky, editor, *Concurrency in Dependable Computing*, pages 212–224. Kluwer Academic Publishers, Berlin, Heidelberg, 2002.

[PHS06] Eric Platon, Shinichi Honiden, and Nicolas Sabouret. Challenges in exception handling in multi-agent systems. In *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, SELMAS '06, pages 45–50, New York, NY, USA, 2006. ACM.

[Pla06] Eric Platon. Smart environment for smarter agents in e-markets. In *FLAIRS Conference*, pages 176–177, 2006.

[Pon11] Julien Ponge. Better resource management with java se 7: Beyond syntactic sugar, 5 2011.

[Poo] Martin Pool. Unhandled exception. C2M.

[pro] programmerbase.net. Tutorial / java grundlagen / exceptions. http://www.programmersbase.net.

[PSH07] Eric Platon, Nicolas Sabouret, and Shinichi Honiden. A definition of exceptions in agent-oriented computing. In *Proceedings of the 7th international conference on Engineering societies in the agents world VII*, ESAW'06, pages 161–174, Berlin, Heidelberg, 2007. Springer-Verlag.

[RdLFF05] C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. C. Filho. Exception handling in the development of dependable component-based systems. In *Software Practice and Experience, 35(3):195–236,*, 2005.

[RK01] Alexander B. Romanovsky and Jörg Kienzle. Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In *Advances in Exception Handling Techniques (the book grow out of a ECOOP 2000 workshop)*, pages 147–164, London, UK, UK, 2001. Springer-Verlag.

[Sak09] Kenneth Saks. Ejb core contracts and requirements. Technical report, Sun Microsystems, 2009.

[SCG07] Nazaraf Shah, Kuo-Ming Chao, and Nick Godwin. Exception diagnosis architecture for open multi-agent systems. In Ricardo Choren, Alessandro Garcia, Holger Giese, Ho-Fung Leung, Carlos Lucena, and Alexander Romanovsky, editors, *Software Engineering for Multi-Agent Systems V*, pages 77–98. Springer-Verlag, Berlin, Heidelberg, 2007.

[Sch98] Jonathan L. Schilling. Optimizing away c++ exception handling. *SIGPLAN Not.*, 33(8):40–47, August 1998.

[She] Srikanth Shenoy. Best practices in ejb exception handling. IBM.

[Sil10] Bruce Silver. *BPMN METHOD & Style*. Cody-Cassidy Press, 2010.

[SS04] Kevin Simons and Judith Stafford. Container-managed exception handling framework. Technical report, Tufts University, 2004.

[Str] Bjarne Stroustrup. Bjarne stroustrup's c++ style and technique faq. http://www2.research.att.com.

[Str94] Bjarne Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[Ten77] R. D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:97–112, 1977. 10.1007/BF00289243.

[TM01] Anand Tripathi and Robert Miller. Exception handling in agent-oriented systems. In *Advances in Exception Handling Techniques (the book grow out of a ECOOP 2000 workshop)*, pages 128–146, London, UK, UK, 2001. Springer-Verlag.

[TY00] S.-I. Tazuneki and T. Yoshida. Concurrent exception handling in a distributed object-oriented computing environment. In *Parallel and Distributed Systems: Workshops, Seventh International Conference on, 2000*, pages 75 –82, oct 2000.

[Ull] Christian Ullenboom. Java ist auch eine insel.

[Unb] Unbekannt. Exception reporter. C2M.

[Ven97a] Bill Venners. http://www.artima.com, 1 1997.

[Ven97b] Bill Venners. http://www.artima.com, 2 1997.

[vRa] Guido van Rossum. 8.3 handling exceptions. http://docs.python.org.

[vRb] Guido van Rossum. What's new in python 3.0. http://docs.python.org.

[vR91] Guido van Rossum. Interactively testing remote servers using the python programming language. Technical report, CWI, 1991.

[vR05] Brett Cannon; Guido van Rossum. Required superclass for exceptions. http://www.python.org, 10 2005.

[vRNC05] Guido van Rossum; Nick Coghlan. The "withßtatement. http://www.python.org, 5 2005.

[WB] Ping Wang and Russell Butek. Web services programming tips and tricks: Exception handling with jax-rpc. IBM.

[Wika] Wikipedia. Java version history. http://en.wikipedia.org.

[Wikb] Wikipedia. Resource acquisition is initialization. http://en.wikipedia.org.

[Wik11] Wikipedia. More c++ idioms/copy-and-swap. http://en.wikibooks.org, 2011. [Online; Stand 23. April 2012].

[WVP+05] Danny Weyns, H. Van, H. Van Dyke Parunak, Fabien Michel, Tom Holvoet, and Jacques Ferber. Environments for multiagent systems, state-of-the-art and research challenges. Technical report, CiteSeerX - Scientific Literature Digital Library and Search Engine [http://citeseerx.ist.psu.edu/oai2] (United States), 2005.

[XRR98] Jie Xu, Alexander Romanovsky, and Brian Randell. Coordinated exception handling in distributed object systems: From model to system implementation. In *Proceedings of the The 18th International Conference on Distributed Computing Systems*, ICDCS '98, pages 12–, Washington, DC, USA, 1998. IEEE Computer Society.

[YB85a] Shaula Yemini and Daniel M. Berry. A modular verifiable exception handling mechanism. *ACM Trans. Program. Lang. Syst.*, 7(2):214–243, April 1985.

[YB85b] Shaula Yemini and Daniel M. Berry. A modular verifiable exception handling mechanism. *ACM Trans. Program. Lang. Syst.*, 7(2):214–243, April 1985.

[Zad] A.M. Kuchling;Moshe Zadka. What's new in python 2.0. http://docs.python.org.