# Zentrale Fehlerbehandlung in verteilten eingebetteten Systemen

Simon Spinner, B.Sc.
Angewandte Informatik
Hochschule Offenburg

5. Mai 2009

## Abstract

*Die Fehlerbehandlung ist ein wichtiger Teil bei der Implementierung von robusten und fehlertoleranten Systemen. In modernen Programmiersprachen gibt es meist bereits spezielle Konstrukte zur Fehlerbehandlung. Insbesondere in verteilten und nebenläufigen Szenarien sind die traditionellen, sequentiellen Fehlerbehandlungssysteme allerdings nicht ausreichend. Es werden mit Erlang, dem Guardian Modell und den CA Actions drei aktuellere Lösungsvorschläge für dieses Problem vorgestellt. Es wird dabei auch auf deren Einsetzbarkeit in eingebetteten Systemen eingegangen. Abschließend wird ein Vergleich der vorgestellten Systeme vorgenommen.*

## 1 Einführung

Eingebettete Systeme werden oft in Bereichen eingesetzt, in denen hohe Anforderungen an die Robustheit, Zuverlässigkeit und Fehlertoleranz gestellt werden. Um diese Anforderungen zu erfüllen muss beim Entwurf solcher Systeme, neben der Fehlervermeidung und Fehlermaskierung, ein besonderer Wert auf die Behandlung von Fehlern und Ausnahmesituationen gelegt werden.

Für sequentielle Anwendungen wurden in der Vergangenheit Fehlerbehandlungsmechanismen bereits tiefgehend untersucht und sind bei neueren, objekt-orientierten Programmiersprachen, wie z.B. C++, Java und C#, schon im Sprachumfang enthalten. Solche Mechanismen bestehen normalerweise aus folgenden drei Schritten:

1. Der Fehler wird vom Programm oder von der Laufzeitumgebung erkannt, z.B. das Teilen durch Null.
2. Der Fehler wird signalisiert, bzw. ausgelöst, z.B. mit der *throw* Anweisung.
3. Es wird nach einem passenden Fehlerhandler im Programm gesucht und dieser wird ausgeführt. Falls kein Handler gefunden wird, wird entlang des Aufrufstacks nach einem Handler gesucht.

Für verteilte, nebenläufige Systeme sind die exisitierenden sequentiellen Fehlerbehandlungsmechanismen jedoch nicht ausreichend. In [5], [8] werden folgende drei Hauptgründe hierfür genannt:

1. Eine Ausnahme kann asynchron in einem Prozess ausgelöst werden, aufgrund einer Signalisierung in einem anderen Prozess. Ein Prozess muss jedoch im richtigen Kontext sein um die Ausnahme, die von einem anderen Prozess signalisiert wurde, korrekt zu behandeln.
2. Die Ausnahmebehandlung in verschiedenen Prozessen muss möglicherweise koordiniert werden.
3. Es können gleichzeitig mehrere Ausnahmen im System ausgelöst werden. Diese Ausnahmen können kausal zusammenhängen und die gleiche Ursache haben. Zum Beispiel kann bei einem zweimotorigen Flugzeug die Schubkraft ausgleichend reguliert werden, falls ein Motor ausfällt. Falls jedoch beide Motoren ausfallen, dürfen die Ausnahmen nicht getrennt behandelt werden [10].

Grundsätzlich lassen sich drei Arten von Nebenläufigkeit unterscheiden [6],[1]. *Unabhängige* Nebenläufigkeit liegt vor, wenn verschiedene Prozesse zwar parallel abgearbeitet werden, ohne dass sie jedoch miteinander kommunizieren oder sich Ressourcen teilen. Bei der *konkurrierenden* Nebenläufigkeit greifen die Prozesse zwar auf gemeinsame Objekte zu, sie arbeiten jedoch nicht zusammen an der Lösung einer Aufgabe. Diese Systeme basieren oft auf verteilten Transaktionen, die im Fehlerfall

zurückgerollt werden. Bei eingebetteten, verteilten Systeme handelt es sich allerdings meist um *kooperierende* Nebenläufigkeit. Dabei arbeiten die Prozesse an einem gemeinsamen Ziel, indem jeder eine Teilaufgabe erledigt. In solch einem Szenario ist die Fehlerbehandlung besonders schwierig. Im Folgenden wird hauptsächlich diese Art der Nebenläufigkeit betrachtet.

Zusätzlich werden von eingebetteten Systemen häufig Echtzeiteigenschaften gefordert. Solche Systeme müssen auch im Fehlerfall geforderte Zeitschranken einhalten. Bei Echtzeitsystemen muss das Fehlerbehandlungssystem vorhersagbar sein. Das Werfen einer Ausnahme und die Auswahl eines Handlers sollte zeitbeschränkt und vorzugsweise eine Komplexität von O(1) haben [4]. Außerdem sollte die Priorität für den Fehlerhandler angegeben werden können, damit keine höherprioren Tasks blockiert werden. So ist es zum Beispiel in einem Flugkontrollsystem wichtiger, dass der Algorithmus, der den Weiterflug des Flugzeugs regelt, nicht von einer Ausnahme wegen eines fehlerhaften Temperatursensors gestört wird [4]. Diese Aspekte werden bei der folgenden Betrachtung von Fehlerbehandlungssystemen miteinbezogen.

Der nächste Abschnitt beschäfigt sich mit den allgemeinen Konzepten, die bei der Fehlerbehandlung in verteilten eingebetteten Systemen von Bedeutung sind. Abschnitt 3 stellt dann die verteilten Fehlerbehandlungssysteme vor, die in den letzten Jahren entwickelt wurden. In Abschnitt 4 werden die Systeme dann anhand von bestimmten Kriterien verglichen. In Abschnitt 5 findet eine Schlussfolgerung statt.

# 2 Allgemeine Konzepte

Die Konzepte für die Fehlerbehandlung in verteilten Systemen sind denen im sequentiellen Fall ähnlich. In entscheidenden Punkten unterscheiden sie sich allerdings und sind entsprechend erweitert worden um den Anforderungen gerecht zu werden.

## 2.1 Fehlererkennung

Das Verfahren wie man einen Fehler feststellen kann, hängt maßgeblich von der Art des Fehlers ab. In [4],[3] wird folgende Klassifizierung für Fehler vorgeschlagen:

- *Softwarefehler*, wie z.B. eine Division durch Null.
- *Hardwarefehler*, wie z.B. der Ausfall eines Sensors.
- *Zustand-/Umgebungsfehler*, falls der vom System wahrgenommene Zustand sich vom tatsächlichen unterscheidet.
- *Timingfehler*, falls in einem Echtzeitsystem die Zeitschranken nicht eingehalten werden können.

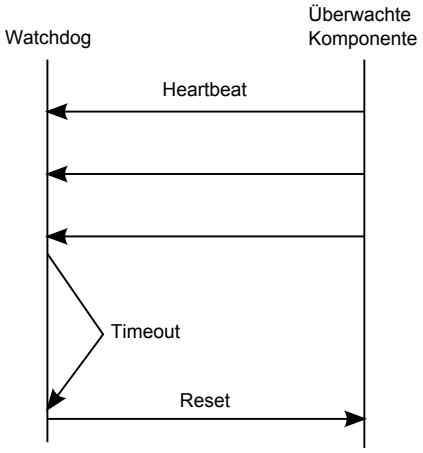


**Abbildung 1:** Watchdog Pattern

*Software- und Hardwarefehler* lassen sich durch *Assertions* im Programmcode aufdecken, die Vor- und Nachbedingungen und Invarianten überprüfen. Außerdem gibt es noch die Möglichkeit durch zusätzliche Komponenten, die *Watchdog* genannt werden, sicherzustellen, dass alle Soft- und Hardwaresysteme funktionsfähig sind. Diese erhalten in regelmäßigen Abständen Heartbeat-Nachrichten von allen Komponenten. Falls diese Nachrichten von einer Komponente ausbleiben schlägt der Watchdog Alarm. Dies wird in Abbildung 1 verdeutlicht. Bei einem Timeout stößt der Watchdog im einfachsten Fall ein Reset der betroffenen Komponente an. Er kann aber auch weitergehende Recoverymechanismen anstoßen. Weitergehende Information finden sich in [2].

*Zustand-/Umgebungsfehler* lassen sich nur sehr schwierig erkennen. Dies hängt von den Informationen ab, die das System von seinen Sensoren geliefert bekommt. *Timingfehler* werden gewöhnlich von der Laufzeitumgebung, wie z.B. einem Echtzeitbetriebssystem, festgestellt.

## 2.2 Fehlersignalisierung

Für die Signalisierung eines Fehlers ist eine zuverlässige Middleware notwendig, an die je nach System noch weitergehende Forderungen wie geordnete Auslieferung und garantierte Zustellzeiten gestellt werden. Die Kommunikation für Signalisierung ist meist nachrichtenorientiert und asynchron. Das Funktionieren der Fehlerbehandlungssysteme hängt entscheidend von der Verfügbarkeit dieser Middleware ab. In sicherheitskritischen Systemen

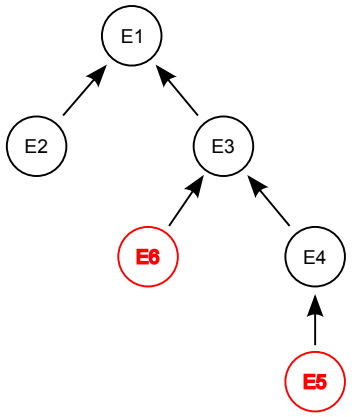


**Abbildung 2:** Beispiel für eine baumartige Hierarchie von Ausnahmen

sollten Fehler auf dieser Ebene durch Redundanz maskiert werden.

Die Fehlersignalisierung kann direkt über einen verteilten Algorithmus erfolgen, der sicherstellt, dass alle betroffenen Systeme über den Fehler informiert werden. Eine Alternative ist die Errichtung einer zentralen Instanz, die von den Komponenten, die den Fehler entdeckt haben, informiert wird und dann entsprechend eine Fehlerbehandlung einleitet. Dies ist unter dem Begriff Safety Executive Pattern bekannt [2]. Die zentrale Variante ist meist performanter und einfacher zu implementieren, während der verteilte Algorithmus eine höhere Ausfallsicherheit bietet.

Um dem Fall, dass mehrere Ausnahmen nebenläufig in einem verteilten System auftreten, gerecht zu werden, haben manche Fehlerbehandlungssysteme einen Mechanismus zum Auflösen mehrerer Ausnahmen integriert. Das Resultat einer solchen Auflösung ist, dass verschiedene zusammenhängende Ausnahmen zu einer Einzigen korreliert werden. In [10] wird argumentiert, dass die Fehlerauflösung notwendig ist, da aufgrund von Nachrichtenlaufzeiten in einem verteilten System nicht alle Knoten sofort unterbrochen werden können. Außerdem breitet sich ein Fehler, der nicht direkt im verursachenden Knoten erkannt wird, im System aus und kann Folgefehler in benachbarten Knoten auslösen.

In [5] werden drei Möglichkeiten zur Fehlerauflösung dargestellt:

1. Alle Ausnahmen, die von einem Knoten nicht lokal behandelt werden können, resultieren in einer vordefinierten *Terminate*-Ausnahme in den anderen Knoten. Dies führt dazu, dass in allen betroffenen Systemen die aktuelle Transaktion abgebrochen oder das System neugestartet wird.
2. Alle Typen von Ausnahmen bilden eine *baumartige Hierarchie*, wobei eine Behandlungsroutine für eine Ausnahme auch eine beliebige Menge von Ausnahmen, die im Baum unterhalb angeordnet sind, behandeln kann. Die *aufgelöste Ausnahme* ist somit die Wurzel des kleinsten Unterbaums, in dem alle nebenläufig signalisierten Ausnahmen enthalten sind. Dies wird in Abbildung 2 an einem Beispiel gezeigt: Falls Ausnahmen E5 und E6 parallel ausgelöst wurden, ergibt sich E3 als aufgelöste Ausnahme. Ausnahmen, die in keiner Beziehung zu einander stehen, bilden getrennte Bäume und werden nicht zu einer Einzigen aufgelöst.
3. Eine *anwendungsspezifische Auflösungsfunktion* löst eine Menge nebenläufiger Ausnahmen zu einer Einzigen auf.

## 2.3 Fehlerbehandlung

Ein Fehler wird meist durch einen speziellen Fehlerhandler behandelt, der einem bestimmten Programmkontext zugeordnet ist. Ein solcher Kontext kann zum Beispiel ein Block oder eine Prozedur sein. Einem Kontext können mehrere Handler zugeordnet sein, wobei für einen bestimmten Typ von Fehler immer nur ein Handler ausgewählt und ausgeführt wird. Diese Handler haben die Aufgabe das System wieder in einen gültigen Zustand zu überführen.

Hierfür gibt es zwei grundlegende Arten von Vorgehensweisen: Backward und Forward Recovery [5]. Bei der *Backward Recovery* wird das System in einen früheren, gültigen Zustand zurückgeführt. Alle Änderungen, die in der Zwischenzeit erfolgt sind, gehen verloren. Um dies zu ermöglichen muss der aktuelle Zustand eines Systems in regelmäßigen Abständen permanent gespeichert werden. Hierfür bietet sich ein Checkpointing-Verfahren, das für eingebettete Echtzeitsysteme optimiert werden kann, wie in [11] beschrieben. Der Vorteil der Backward Recovery ist, dass keine Kenntnisse über den aufgetretenen Fehler notwendig sind und das System auch nach unbekannten Fehlern wieder in einen gültigen Zustand zurückversetzt werden kann. Deswegen eignet sich Backward Recovery auch insbesondere für den Standardhandler, der aufgerufen wird falls kein anderer Handler bestimmt werden kann [9]. Bei der Backward Recovery besteht allerdings in einem verteilten System die Gefahr, dass beim Zurückrollen eines Prozesses eine Reihe weiterer Prozesse auch zurückgerollt werden müssen, da sie miteinander kommuniziert haben. Dies kann möglicherweise zu einem kaskadierten Zurückrollen führen [5]. Dies bezeichnet man auch als den Domino-Effekt. Außerdem ist es nicht immer möglich alle vorgenommenen

Änderungen wieder rückgängig zu machen. Insbesondere Änderungen durch Aktoren an der Umgebung sind nicht umkehrbar [8].

Bei der *Forward Recovery* wird versucht die Auswirkungen des Fehlers zu begrenzen und zu korrigieren. Das Ziel ist es, das System wieder in einen gültigen Zustand zu überführen, damit die normalen Operationen fortgesetzt werden können. Dies ist allerdings nur bei vorhergesehen Fehlern möglich, da eine genauere Analyse des Fehlers notwendig ist und basierend auf der Fehlerart anwendungsspezifische Korrekturmaßnahmen vorgenommen werden [8].

Falls ein Fehler an der Stelle, an der er auftritt, nicht behandelt werden kann, wird er in sequentiellen Programmen meist an die darüberliegende Schicht weitergegeben. Dieser Vorgang wird *Fehlerpropagierung* bezeichnet. Bei verteilten Systemen ist dabei zu beachten, dass eine Fehlerpropagierung in manchen Fällen in zwei Dimensionen ablaufen muss [8]. Das Weitergeben an eine höhere Schicht kann nämlich zur Folge haben, dass nun zusätzliche Systeme, die bisher nicht betroffen waren, in die Fehlerbehandlung miteinbezogen werden müssen.

# 3 Überblick über verteilte Fehlerbehandlungssysteme

Im Folgenden werden drei Modelle zur Fehlerbehandlung in verteilten Systemen vorgestellt. Zuerst wird mit Erlang auf einen Vertreter eingegangen, der bereits eine breitere Verwendung in der Praxis gefunden hat. Dieser ist allerdings sehr einfach gehalten und setzt die oben genannten Anforderungen nur in eingeschränktem Maße um. Mit dem Guardian Modell und den Coordinated Atomic Actions wird dann nachfolgend noch auf umfassendere Lösungsvorschläge eingegangen.

## 3.1 Erlang

Erlang ist eine Programmiersprache, die speziell für die Programmierung von Telekommunikations-Switches entwickelt wurde. Sie wurde auf Effizienz, Robustheit und starke Nebenläufigkeit optimiert.

Ein *Prozess* in Erlang ist ein Codestück, das parallel ausgeführt werden kann. Prozesse können über asynchrone Nachrichten miteinander kommunizieren. Es werden zwei Arten von Prozessen unterschieden: *Worker* und *Supervisor*. Die Prozesse sind in einem Baum angeordnet. Die Blätter des Baumes sind die Worker, die Knoten sind die Supervisor Prozesse. Falls in einem Worker-Prozess eine Ausnahme auftritt, die nicht lokal behandelt werden kann, wird sie an den Supervisor-Prozess signalisiert und der Worker-Prozess stirbt sofort. Der Supervisor-Prozess entscheidet dann je nach Art des Worker-Prozesses, ob dieser neugestartet wird. Ansonsten werden keine weiteren Aktionen eingeleitet. Außerdem kennt Erlang noch die Möglichkeit beliebige Prozesse miteinander zu verlinken. Falls ein Prozess abstürzt, werden auch alle mit ihm verbundenen Prozesse beendet.

Das Ziel von Erlang ist es, die Auswirkungen eines Fehlers auf einen Unterbaum von Prozessen zu beschränken und dass ein Fehler möglichst schnell zum Neustart der betroffenen Prozesse führt. Die Idee hinter dieser "Fail-Fast"-Strategie ist, dass eine intelligentere Fehlerbehandlung aufwändig und fehleranfällig ist [1]. In der Praxis hat sich dieses Vorgehen bei Telekommunikations-Switches, bei denen sehr viele Prozesse gleichzeitig ablaufen, als effizient und robust erwiesen. Allerdings ist dieses Verfahren nicht allgemein anwendbar, da z.B. das Neustarten eines Prozesses zu teuer ist oder bereits ausgeführte Aktionen wieder revidiert werden müssen. In solchen Fällen ist eine größere Kontrolle durch den Programmierer über die Fehlerbehandlung notwendig.

## 3.2 Das Guardian Modell

Beim Guardian Modell gibt es einen Knoten, der die Rolle des Leaders übernimmt. Dieser koordiniert die Fehlerbehandlung. Die restlichen Knoten sind "Teilnehmer". In [5] ist auch ein Modell ohne zentrale Instanz vorgesehen, dies wird hier jedoch nicht weiter betrachtet.

Die einzelnen Knoten können mit den Primitiven *enableContext* und *disableContext* einem Fehlerkontext beitreten, bzw. ihn verlassen. Jedem Kontext ist ein benutzerdefinierter, eindeutiger Namen zugeordnet. Jedem Kontext ist eine Menge Fehlerhandler zugeordnet.

Falls in einem Knoten nun eine Ausnahme auftritt, die global behandelt werden muss (d.h. abgeleitet von *GlobalException*), wird diese dem Leader mit dem Primitiv *gthrow* signalisiert, wie in Abbildung 3 dargestellt. Der Ausnahme wird dabei mitgegeben an welchen Zielkontext sie geleitet werden soll. Der Zielkontext kann sich vom Kontext, aus dem die Ausnahme geworfen wird, unterscheiden. Der Leader sorgt dann dafür, dass alle Knoten die gerade im genannten Zielkontext sind, suspendiert werden. Die Knoten fragen nun mit *checkExceptionStatus* die vorliegenden Ausnahmen ab. Der Aufruf blockiert solange bis alle Teilnehmer suspendiert sind. Danach signalisiert der Leader den Knoten die Ausnahme und der passende Fehlerhandler wird aufgerufen.

Falls eine weitere Ausnahme in einem Knoten

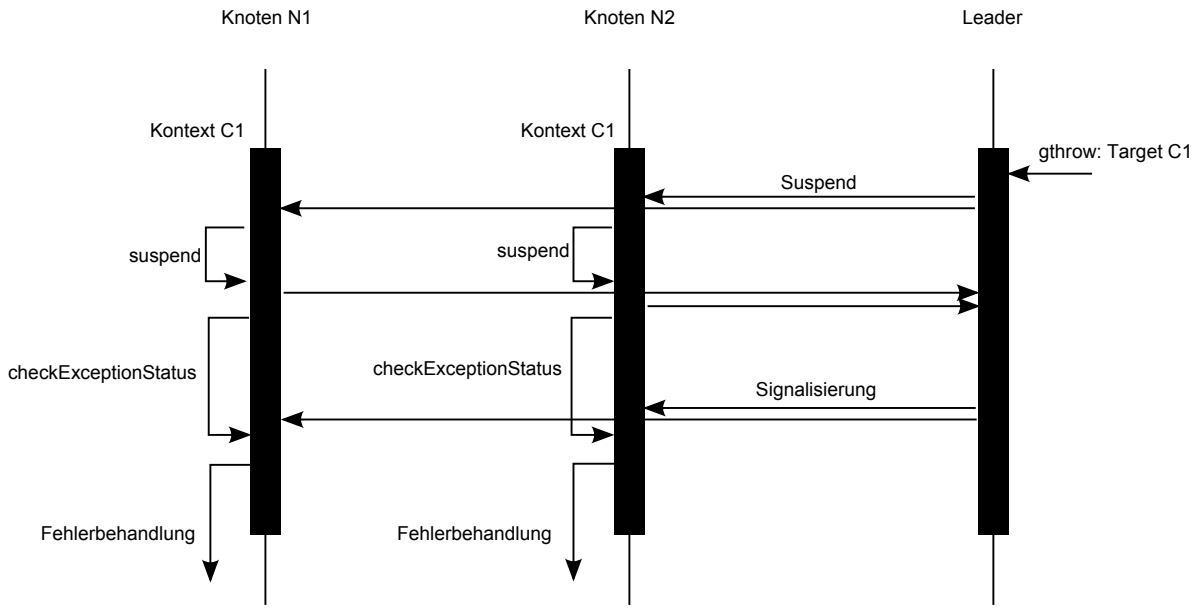


**Abbildung 3:** Ablauf der Fehlerbehandlung beim Guardian Modell.

geworfen wird, bevor alle Teilnehmer suspendiert sind, wird ein Fehlerauflösungsalgorithmus gestartet. Dieser basiert auf einer baumartigen Hierarchie wie in Abschnitt 2.2 beschrieben. Falls die Ausnahmen im gleichen Baum enthalten sind, wird die zusammengeführte Ausnahme signalisiert, ansonsten werden sie getrennt behandelt.

Das Guardian Modell wurde bisher noch in keine Programmiersprache direkt integriert. Die entsprechenden Primitive müssen somit durch Bibliotheksfunktionen realisiert werden. Außerdem wird eine zuverlässige und reihenfolgengetreue Nachrichtenmiddleware vorausgesetzt. Zu Informationen über die Realisierung eines Systems basierend auf dem Guardian Modell siehe [5].

Das Guardian Modell ist ein sehr umfassendes Modell zur Fehlerbehandlung in verteilten Systemen. Auf der Basis dieses Modells lassen sich auch andere Modelle umsetzen, wie z.B. die nachfolgend beschriebenen Coordinated Atomic Actions, siehe [5]. Allerdings skaliert es nicht sehr gut: Bei vielen Teilnehmern kann es zu Performanceproblemen kommen, da bei jeder globalen Ausnahme alle suspendiert werden müssen. Aus dem gleichen Grund gibt es auch Probleme, falls bestimmte Knoten ununterbrechbare, wichtige Aufgaben zu erledigen haben [1]. Deshalb ist es auch nicht für Echtzeitsysteme geeignet.

## 3.3 Coordinated Atomic (CA) Actions

Das Konzept einer Konversation ist erfunden worden, um die Auswirkungen eines Fehlers in einem verteilten System einzugrenzen und den in Abschnitt 2.3 beschriebenen Domino-Effekt zu verhindern. Ein Prozess, der an einer Konversation teilnimmt, kann nur mit anderen Prozessen in der gleichen Konversation kommunizieren. Beim Betreten einer Konversation sichert ein Prozess seinen akutellen Zustand. Alle Prozesse in einer Konversation müssen einen Akzeptanztest beim Verlassen durchführen. Falls einer der Teilnehmer den Akzeptanztest nicht besteht oder ein Fehler während der Konversation auftritt, stellen alle Prozesse den ursprünglichen Zustand wieder her [5].

Konversationen können eingesetzt werden bei nebenläufigen Prozessen, die darauf ausgelegt wurden, miteinander zu kommunizieren. Hingegen sind Transaktionen einsetzbar um Zugriffe auf Objekte zu serialisieren, die von unabhängig entwickelten Aktionen verwendet werden [9].

Der Ansatz der Coordinated Atomic Actions kombiniert diese beiden Ansätze. Eine CA Action stellt den Rahmen einer Konversation dar. Beim Betreten einer CA Action wird ein Wiederherstellungspunkt erzeugt, falls Backward Error Recovery eingesetzt wird. Beim Verlassen der CA Action werden lokale oder globale Akzeptanztests durchgeführt, um sicherzustellen, dass alle Teilnehmer in einem gültigen Zustand sind.

Zugriffe auf externe Objekte starten eine atomare Transaktion. Dadurch wird sichergestellt, dass der Konversation keine Daten von außerhalb untergeschoben werden können. Falls nur ein Teilnehmer in einer CA Action ist, der auf externe Objekte zugreift, entspricht sie somit einer normalen Transaktion. Falls in einer CA Action nicht auf externe

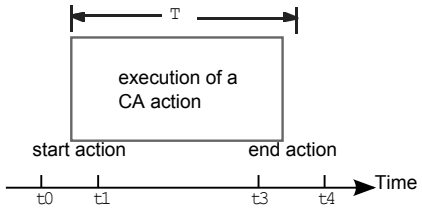


**Abbildung 5:** Timing Bedingungen bei einer CA Action [6]

Objekte zugegriffen wird, verhält sie sich wie eine Konversation. CA Actions können beliebig ineinander geschachtelt werden.

Falls nun ein Fehler in einem der Teilnehmer auftritt, der nicht lokal behandelt werden kann, oder ein Akzeptanztest nicht bestanden wird, so wird, wie in Abbildung 4 gezeigt, eine Ausnahme in alle Teilnehmern ausgelöst. Dadurch werden entsprechende Fehlerhandler in den Teilnehmern ausgeführt. Je nach Anwendung kann dabei Backward oder Forward Recovery eingesetzt werden. Im gezeigten Beispiel wird das System in einen neuen gültigen Zustand versetzt. Anderenfalls müssten sowohl die Änderungen an den Teilnehmern als auch die an den externen Objekte zurückgerollt werden.

Falls eine Ausnahme in der aktuellen CA Action nicht behandelt werden kann, wird diese an die darüberliegende Schicht signalisiert. Bei nebenläufig auftretenden Ausnahmen ist ein Auflösungsalgorithmus basierend auf einer baumartigen Hierarchie, wie in 2.2 beschrieben, vorgesehen. Die Koordinierung kann eine zentrale Instanz übernehmen oder durch einen verteilten Algorithmus gelöst werden [9]. Weitere Details zu CA Actions finden sich in [9], [6], [4].

Das Modell wird in [6] um Konstrukte für Echtzeitsysteme erweitert. Wie in Abbildung 5 verdeutlicht, lassen sich zu einer CA Action Zeitbedingungen angegeben. Es lässt sich spezifizieren in welchem Zeitraum $[t_0, t_1]$ die CA Action gestartet werden muss, in welchem Zeitraum $[t_3, t_4]$ sie enden muss, und wie lange (T) sie maximal dauern darf. Damit lassen sich zeitgesteuerte Tasks und Deadlines umsetzen.

Das Konzept der CA Actions wurde bereits vor über 10 Jahren vorgeschlagen [9] und wurde seitdem fortwährend weiterentwickelt. Es wurde tiefgehend untersucht und formal beschrieben. Am Forschungszentrum Informatik in Karlsruhe, wird anhand einer Fallstudie die Anwendbarkeit für verteilte Kontrollanwendungen in der Industrie untersucht. In diesem Zusammenhang wurde auch eine Implementierung der CA Actions für diese Fallstudie vorgenommen [7].

# 4 Vergleich

In Tabelle 1 sind die verschiedenen Fehlerbehandlungssysteme gegenübergestellt. Dabei werden folgenden Kriterien betrachtet:

**Komplexität:** Gibt an, wie komplex es ist, das System zu implementieren.

**Sprachintegration:** Gibt an, ob das Fehlerbehandlungssystem in eine Programmiersprache integriert ist oder als zusätzliche Bibliothek verwendet werden muss.

**Fehlerauflösung:** Gibt an, nach welchem Verfahren gleichzeitig auftretende Ausnahmen aufgelöst werden.

**Fehlerbehandlung:** Gibt an, welche Art von Fehlerbehandlung mit dem System möglich ist.

**Echtzeit:** Gibt an, ob Echtzeitfähigkeiten im System vorgesehen sind.

**Skalierbarkeit:** Gibt an, wie gut ein System mit der Anzahl paralleler Prozesse skaliert.

Der große Vorteil des Fehlerbehandlungssystems von Erlang ist, dass es in die Programmiersprache integriert ist und relativ einfach einsetzbar ist. Beim Guardian Modell und den CA Actions fehlt eine solche Sprachintegration. Dies macht deren Einsatz komplizierter und fehleranfälliger, da der korrekte Einsatz der Primitive nur schwer sicherzustellen ist.

Ein weiterer Vorteil von Erlang ist, dass es sich resourcenschonend implementieren lässt und sehr gut skaliert, wie der Einsatz in Telekommunikations-Switches zeigt. Das Guardian Modell und CA Actions werden oft zusammen mit Backward Recovery eingesetzt. Obwohl dies auch mit eingebetteten Systemen durchaus möglich ist, sind negative Auswirkungen auf die Laufzeit unvermeidlich.

Das Problem bei Erlang ist, dass es keine Möglichkeiten bietet die Fehlerbehandlung anzupassen. Somit können damit nicht alle Arten von Anforderungen an die Fehlerbehandlung umgesetzt werden. Die anderen beiden Modelle schränken den Entwickler in dieser Hinsicht nicht ein und sind deutlich allgemeiner.

Ein wichtiger Aspekt bei eingebetteten Systemen ist auch die Echtzeitfähigkeit. Auf diesem Gebiet sind die CA Actions am weitesten fortgeschritten, da ihr Modell bereits für Echtzeitanwendungen erweitert wurde. Bei Erlang und dem Guardian Modell ist dies nicht der Fall, wobei bei letzterem zweifelhaft ist, ob dies möglich ist.

# 5 Schlussfolgerung

In diesem Paper wurden zuerst die allgemeinen Konzepte der Fehlerbehandlung in verteilten Systemen dargestellt. Danach wurden mit Erlang, dem

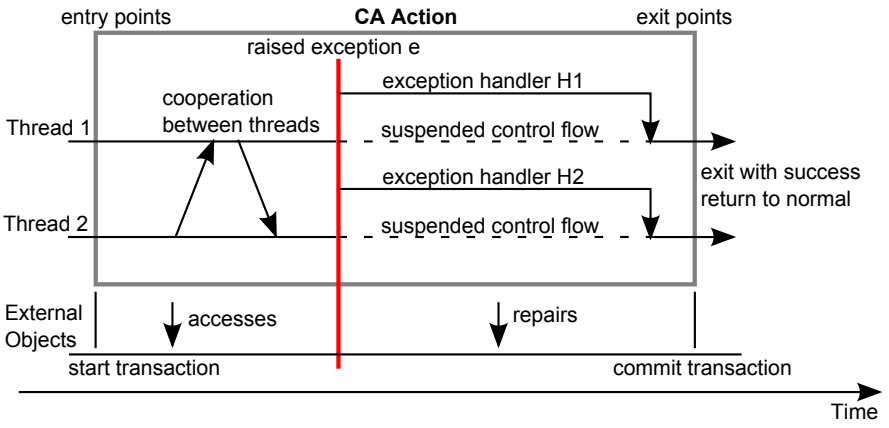


**Abbildung 4:** Fehlerbehandlung bei einer CA Action [6]

| | **Erlang** | **Guardian** | **CA Action** |
|---|---|---|---|
| **Komplexität** | einfach | hoch | hoch |
| **Sprachintegration** | ja | nein | nein |
| **Fehlerauflösung** | Terminate-Ausnahme | baumbasierte Hierarchie | baumbasierte Hierarchie |
| **Fehlerbehandlung** | Restart | Backward/Forward | Backward/Forward |
| **Echtzeit** | nein | nein | ja |
| **Skalierbarkeit** | hoch | gering | k.A. |

**Tabelle 1:** Vergleich der verschiedenen Fehlerbehandlungssysteme

Guardian Modell und den CA Actions drei aktuelle Ansätze für das Problem vorgestellt. Es wurde dabei auch ihre Anwendbarkeit auf eingebettete Systeme näher untersucht. Schließlich wurden die drei Ansätze vergleichend nebeneinander gestellt.

Es hängt dabei von den Anforderungen der zu erstellenden Anwendung ab, für welches der drei Modelle man sich am besten entscheidet. Es wurde allerdings auch deutlich, dass die Integration in Programmiersprachen ein Thema ist, das weiterverfolgt werden muss, um die Fehlerbehandlung in verteilten und nebenläufigen Systemen zu verbessern.

Außerdem ist es notwendig, deren Einsatz für eingebettete Systeme tiefergehend zu untersuchen. Dabei dabei muss insbesondere auch Wert auf den Ressourcenverbrauch und die Echtzeitfähigkeit gelegt werden.

## Literatur

[1] Aurelien Campeas. Distributed exception handling: Ideas, lessons and issues with recent exception handling systems. In Nicolas Guelfi, editor, *Rapid integration of software engineering techniques: First International Workshop, RISE 2004, Luxembourg-Kirchberg, Luxembourg, November 2 - 6, 2004*, volume 3475 of *Lecture notes in computer science*, pages 82–92. Springer, Berlin, 2005.

[2] Bruce Powel Douglass. *Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks, and Patterns*. Object Technology Series. Addison-Wesley, 1999.

[3] B. Gallina, N. Guelfi, and A. Romanovsky. Coordinated atomic actions for dependable distributed systems: the current state in concepts, semantics and verification means. *Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on*, pages 29–38, Nov. 2007.

[4] 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, 1998.

[5] R. Miller and A. Tripathi. The guardian model and primitives for exception handling in distributed systems. *Software Engineering, IEEE Transactions on*, 30(12):1008–1022, Dec. 2004.

[6] A. Romanovsky, J. Xu, and B. Randell. Exception handling in object-oriented real-time distributed systems. *Object-Oriented Real-time Distributed Computing, 1998. (ISORC*

*98) Proceedings. 1998 First International Symposium on*, pages 32–42, Apr 1998.

[7] J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A.F. Zorzo, E. Canver, and F. von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *Computers, IEEE Transactions on*, 51(2):164–179, Feb 2002.

[8] J. Xu, A. Romanovsky, and B. Randell. Coordinated exception handling in distributed object systems: from model to system implementation. *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, pages 12–21, May 1998.

[9] Jie Xu, B. Randell, A. Romanovsky, C.M.F. Rubira, R.J. Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 499–508, Jun 1995.

[10] Jie Xu, A. Romanovsky, and B. Randell. Concurrent exception handling and resolution in distributed object systems. *Parallel and Distributed Systems, IEEE Transactions on*, 11(10):1019–1032, Oct 2000.

[11] Ying Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 918–923, 2003.