Architekturen und Frameworks für zuverlässige und adaptive Informationssysteme

Seminar Dependable Adaptive Systems

Christoph R. Hartel

13. Januar 2006

Betreuer:
Dipl.-Inf. Jürgen Göres,
AG Heterogene Informationssysteme,
TU Kaiserslautern

Inhaltsverzeichnis

1 Einleitung . . . 1
  1.1 Motivation . . . 1
  1.2 Abgrenzung . . . 2
  1.3 Vorgehen . . . 3
2 Grundlagen und Terminologie . . . 3
  2.1 Architekturen und Frameworks . . . 3
  2.2 Grundlagen verlässlicher Systeme . . . 4
  2.3 Grundlagen adaptiver Systeme . . . 5
  2.4 Bedeutung der Systemarchitektur für DAIS . . . 6
3 Grundprinzipien von DAIS-Architekturen . . . 6
  3.1 Komponentenbasierte Architektur . . . 7
  3.2 Separation of Concerns . . . 7
  3.3 Architectural Reflection . . . 10
  3.4 Redundanz . . . 11
  3.5 Realisierungsprobleme . . . 12
4 Architekturen verlässlicher Systeme . . . 15
  4.1 N-Version-Software . . . 15
  4.2 Recovery Blocks . . . 16
  4.3 Multi-Versioning Connectors . . . 17
  4.4 RAIC . . . 18
  4.5 Component Redundancy . . . 20
  4.6 Simplex-Architektur . . . 22
  4.7 Exception Handling auf Architekturebene . . . 24
  4.8 Versatile Dependability . . . 26
5 Architekturen adaptiver Systeme . . . 28
  5.1 Architekturstile . . . 28
  5.2 Contract-based Architecture . . . 30
  5.3 Distributed Configuration Routing . . . 33
  5.4 Dynamic Dispatch . . . 34
6 Zusammenfassung und Ausblick . . . 36

1 Einleitung

Dieses Kapitel soll der Motivation zur Beschäftigung mit Architekturen und Frameworks verlässlicher, adaptiver Systeme dienen. Zudem nehmen wir in Abschnitt 1.2 eine Abgrenzung des inhaltlichen Umfangs dieser Arbeit vor und geben in Abschnitt 1.3 einen Überblick über das weitere Vorgehen.

1.1 Motivation

Informationssysteme sind in nahezu allen Bereichen des menschlichen Lebens und Wirkens vertreten und finden kontinuierlich größere Verbreitung. In vielen Anwendungsbereichen spielt dabei die Verlässlichkeit der Systeme eine zentrale Rolle. Dazu zählen Systeme in der Automobil-, Luft- und Raumfahrttechnik, Unternehmensanwendungen, Handelsplattformen, Bank- und Versicherungssysteme, Steuerungssoftware in nuklearen Einrichtungen und industrielle Fertigungssysteme, um nur einige zu nennen.

Ein Ausfall dieser Systeme ist mit hohen materiellen, finanziellen und u.U. sogar lebensbedrohlichen Risiken verbunden. Gleichzeitig steigen jedoch auch die Anforderungen an die Leistungsfähigkeit und den Funktionsumfang der Systeme und damit unausweichlich auch ihre Größe und Komplexität. Dabei ist bereits in heutigen (realen) Systemen in aller Regel weder eine formale Spezifikation der Umgebungsbedingungen und Anforderungen noch eine Verifikation des eigentlichen Systems möglich. Selbst bei relativ kleinen Systemen etwa im eingebetteten Bereich scheitern diese Methoden in vielen Fällen; in großen Informationssystemen wird ihre Anwendung noch für lange Zeit unmöglich sein und es vielleicht sogar immer bleiben.

Den Aspekt unscharfer Spezifikationen adressieren adaptive Systeme, die eine selbstständige Anpassung eines Systems zu seiner Laufzeit ermöglichen sollen. Neben Unschärfen in den ursprünglichen Anforderungen berücksichtigen diese auch die kontinuierliche Veränderung von Anforderungen und Umgebungsbedingungen, die gerade vor dem Hintergrund sehr langlebiger und wartungsintensiver Systeme insbesondere im Unternehmensbereich von hoher Bedeutung sind. Aber auch in kurzfristigeren Zeiträumen bieten adaptive Systeme Vorteile, etwa, indem sie auf kontinuierlich schwankende Systemlasten (z.B. Benutzerzugriffe) oder Umweltbedingungen (z.B. Verfügbarkeit von Ressourcen) flexibel und zuverlässig reagieren können.

Dennoch befinden wir uns in dem Dilemma, dass der Verlässlichkeit von Informationssystemen einerseits eine zentrale Bedeutung zukommt und diese sich andererseits – zumindest formal – nicht garantieren lässt. Daher muss durch den Einsatz bewährter Entwicklungsverfahren und Werkzeuge und ein sorgfältiges Vorgehen die *Wahrscheinlichkeit* für das verlässliche Funktionieren eines Systems erhöht werden. Neben dem Vorgehen bei der Entwicklung spielt allerdings auch der Aufbau eines Informationssystems – namentlich seine Architektur – eine bedeutende Rolle für seine Verlässlichkeit, obwohl dieser Bereich in der Vergangenheit vernachlässigt wurde.

Dabei sind die Konzepte, die bei der Architektur verlässlicher Systeme zum Einsatz kommen, keineswegs neu. So zitiert bspw. [Aviz95] einen Artikel von Dionysius Lardner aus dem Jahre 1834, in dem dieser schreibt:

> „The most certain and effectual check upon errors which arise in the process of computation, is to cause the same computations to be made by separate and independent computers[1]; and this check is rendered still more decisive if they make their computations by different methods.“

Damit definiert Lardner das Konzept der Redundanz[2], welches für verlässliche und adaptive Systeme von grundlegender Bedeutung ist, wie wir an späterer Stelle im Detail erläutern werden. Neben diesem sind auch zahlreiche andere Konzepte schon seit langer Zeit bekannt, finden aber erst in jüngerer Vergangenheit in systematischer Weise im Zusammenhang mit DAIS Verwendung.

In dieser Arbeit erläutern wir daher die Grundlagen verlässlicher, adaptiver Informationssysteme und diskutieren insbesondere, welche Rolle die Architektur bei ihrer Entwicklung spielt. Wir identifizieren die verschiedenen Architekturprinzipien im DAIS-Bereich und geben einen Überblick über die darauf basierenden, konkreten Architekturen. Der Schwerpunkt liegt dabei auf dem Vergleich der einzelnen Ansätze und einer Evaluation ihrer Eignung für reale Systeme.

Ebenfalls von Bedeutung für die Entwicklung von DAIS ist die Wiederverwendung bewährter und ausgereifter Implementierungen. Neben einer Steigerung der Verlässlichkeit ermöglicht dies inbesondere auch eine deutliche Aufwandsreduktion bei der Erstellung, Anpassung und Wartung von Systemen. Um eine systematische Wiederverwendung zu ermöglichen, sind insbesondere Frameworks erforderlich. Diese stellen Bausteine in Form von Klassen oder Komponenten zur Verfügung, welche sich gezielt zu komplexen Systemen kombinieren und erweitern lassen. Inwiefern Frameworks für die Entwicklung von DAIS-Systemen mit den beschriebenen Architekturen existieren, soll in dieser Arbeit daher ebenfalls untersucht werden.

1.2 Abgrenzung

Wir beschränken uns in dieser Arbeit auf die Betrachtung von *Software*-Architekturen und -Frameworks zur Realisierung verlässlicher, adaptiver Informationsysteme. Dennoch sollte ein Großteil der im Folgenden diskutierten Konzepte auch auf andere Bereiche – etwa die Computer-Hardware – übertragbar sein. Wir bewegen uns dabei auf der Ebene allgemeiner Konzepte, die nicht auf die Verwendung in bestimmten Umgebungen (etwa bestimmte Anwendungsdomänen oder technische Bereiche wie eingebettete Systeme,

[1] Dabei bezieht sich das Wort „Computer“ nicht auf ein elektronisches Gerät im heutigen Sinne, sondern vielmehr auf die *Person* (den „Berechner“), welche die fragliche Berechnung durchführt [Aviz95].

[2] Strenggenommen definiert Lardner damit sogar das Konzept der sog. Design Diversity (vgl. Kapitel 3), das sich in jüngerer Zeit (wieder) großer Popularität in der Literatur erfreut und grundlegend für DAIS-Architekturen ist.

Middleware, etc.) beschränkt sind. Damit verfolgen wir das Ziel, einen umfassenden Überblick über den aktuellen Stand der Technik und mögliche Entwicklungen bei DAIS-Architekturen zu geben und grundlegende Muster und Prinzipien zu identifizieren. Für detaillierte Beschreibungen der einzelnen Architekturen verweisen wir an entsprechender Stelle auf spezielle Quellen.

1.3 Vorgehen

Zunächst klären wir in Kapitel 2 einige wesentliche Begriffe und geben einen Überblick über die Grundlagen verlässlicher und adaptiver Systeme. In Abschnitt 2.4 erläutern wir zudem, warum die Architekturebene für diese Systeme von Bedeutung ist. Kapitel 3 stellt generische Grundprinzipien von DAIS-Architekturen vor, die in allen Architekturen Verwendung finden und als Qualitätsmaß für diese dienen. Kapitel 4 und 5 beschreiben ausgehend von diesen Grundprinzipien konkrete Architekturansätze und stellen sie vergleichend gegenüber. In Kapitel 6 fassen wir die Kernergebnisse noch einmal kurz zusammen und geben einen Ausblick auf mögliche Entwicklungen.

2 Grundlagen und Terminologie

In diesem Kapitel erläutern wir die Grundlagen verlässlicher, adaptiver Systeme und definieren einige wesentliche Begriffe, auf die wir im Folgenden zurückgreifen.

2.1 Architekturen und Frameworks

Als die *Architektur* eines Systems bezeichnen wir seine Organisation, d.h. seine Komponenten, deren Beziehungen untereinander und zu ihrer Umgebung und die Prinzipien, welche das Design und die Entwicklung des Systems bestimmen (vgl. ANSI/IEEE Std. 1471-2000). Unter einem *Framework* verstehen wir in Anlehnung an [Poet03] ein erweiterbares und anpassbares System kollaborierender Softwareeinheiten, das für eine allgemeine, übergeordnete Aufgabenstellung Kernfunktionalitäten mit entsprechenden Bausteinen bereitstellt. Eine besondere Betonung liegt dabei auf der Bereitstellung einer *generischen, wiederverwendbaren Kernimplementierung*, die das Hauptunterscheidungsmerkmal zu einer Architektur darstellt.

In der Literatur – sowohl im Bereich verlässlicher als auch adaptiver Systeme – finden sich zahlreiche Beschreibungen sogenannter „Frameworks“. Diese stellen allerdings im Sinne obiger Definitionen lediglich *Architekturen* bzw. Architekturansätze – nicht jedoch Frameworks – dar, da sie keine konkreten Implementierungen von Kernfunktionalitäten (geschweige denn Bausteine) enthalten. In einigen Fällen verweisen Autoren neben o.g. Architekturbeschreibungen auf beispielhafte Implementierungen zumindest von Teilaspekten. Auch dabei handelt es sich

nicht um Frameworks, da diese Implementierungen nicht auf *Wiederverwendbarkeit* ausgelegt sind, sondern vielmehr um Referenzimplementierungen im Sinne eines Proof-of-Concepts[3] sind.

Dennoch existieren zumindest Projekte, welche die Erstellung von Frameworks im engeren Sinne verfolgen. Diese beschränken sich jedoch i.d.R. auf bestimmte Domänen (z.B. eingebettete Systeme) und implementieren (noch) nicht die existierenden Architekturkonzepte. (Einen guten Überblick über Projekte dieser Art gibt z.B. [MuKi04].) Vor diesem Hintergrund verzichten wir im Folgenden gänzlich auf die Verwendung des Begriffs Framework im Zusammenhang mit verlässlichen, adaptiven Systemen. Stattdessen beschreiben wir Architekturen und verweisen ggf. auf existierende Referenzimplementierungen.

2.2 Grundlagen verlässlicher Systeme

Unter *Verlässlichkeit* (engl. Dependability) verstehen wir nach [ALRL04] die Fähigkeit eines Informationssystems, Ausfälle von Systemfunktionen (engl. *Service Failures*) zu vermeiden, die häufiger auftreten und/oder schwerwiegender sind als im Kontext des Systems zulässig ist. Eine Abweichung des Systemzustandes, die zu einem Service Failure führen kann, bezeichnen wir als *Error*, die Ursache eines Errors als *Fault*.

Faults unterscheiden wir weiter in *passive* und *aktive* Faults. Ein passiver Fault existiert in einem System, hat aber während dessen aktueller Nutzung (noch) nicht zu einem Error geführt. Verursacht ein Fault einen Error, so heißt er aktiv. Den Zustandsübergang bezeichnen wir als *Aktivierung* eines Faults. Insbesondere können daher Faults, die nicht behoben werden, erneut aktiviert werden, also zu einem Error führen.

Der Umgang mit Faults ist offensichtlich kritisch für die Verlässlichkeit eines Informationssystems. [ALRL04] unterscheidet hierbei vier Arten des Umgangs: Fault Prevention, Fault Tolerance, Fault Removal und Fault Forecasting. *Fault Prevention* bedeutet, das Entstehen und die Aktivierung von Faults zu verhindern, *Fault Tolerance* hingegen, trotz des Vorhandenseins und der Aktivierung von Faults Service Failures zu vermeiden. Das Beheben von Faults bezeichnen wir als *Fault Removal*. *Fault Forecasting* bestimmt die Wahrscheinlichkeit des Auftretens von Faults und deren anzunehmende Auswirkungen.

Diese Arten des Umgangs sind nicht exklusiv, sondern – ganz im Gegenteil – ergänzend anzuwenden. In den in Kapitel 4 und Kapitel 5 beschriebenen Architekturen finden sie in verschiedenen Kombinationen Verwendung. Allerdings bestehen gewisse Einschränkungen bzgl. der Anwendbarkeit in Abhängigkeit vom jeweiligen Kontext. So ist es bei (großen) Systemen i.d.R. nicht möglich, *alle* Faults zur Entwicklungszeit vorherzusehen. Auch bei der Verwendung sog. *COTS-Komponenten* (Common/Commercial off the Shelf Components) ist die Vermeidung von Faults schwierig, da i.d.R. die Entwicklung nicht beeinflusst

[3] Beispielhaft seien an dieser Stelle die „Frameworks“ FaTC2 [LiGR02], CASA [MuGl03] und Component Redundancy [DiMu03] genannt.

werden kann [Sha01]. In diesen Szenarien beschränkt sich Fault Prevention daher auf die Verhinderung der Aktivierung von Faults zur Laufzeit. Gleichzeitig gewinnt die Fähigkeit eines Systems, Faults zu tolerieren, an Bedeutung.

2.3 Grundlagen adaptiver Systeme

In der Literatur finden sich sehr unterschiedliche Auffassungen von adaptiver Software und den Definitionen in diesem Kontext verwendeter Begriffe. Wir bezeichnen ein Softwaresystem als *(selbst-)adaptiv*, wenn es sein eigenes Verhalten evaluiert und – sofern die Evaluation ergibt, dass es die ihm angedachte Aufgabe nicht erfüllt, oder eine Verbesserung der Funktionalität oder der Performanz möglich ist – verändert [Ladd00].

Der Aspekt der selbstständigen Verhaltensanpassung unterscheidet adaptive von *adaptierbarer* (engl. adaptable) Software. Diese zeichnet sich lediglich dadurch aus, dass sie nach dem ursprünglichen Deployment angepasst werden kann, um z.B. bekannte Faults zu beheben. Damit ist (nahezu) jedes Softwaresystem adaptierbar[4] – wenn auch mit stark variierendem Aufwand, vor allem in Abhängigkeit von seiner Architektur –, aber bei weitem nicht jedes Softwaresystem ist auch adaptiv. Die beiden Arten von Systemen sind dabei nicht exklusiv: ein Softwaresystem kann sowohl (leicht) adaptierbar als auch adaptiv sein.

Um adaptives Verhalten in Softwaresystemen zu ermöglichen, müssen zwei Grundannahmen erfüllt sein:

1. Es existieren mehrere verschiedene Wege, ein bestimmtes Ziel zu erreichen.
2. Das System verfügt zur Laufzeit über ausreichendes Wissen über sich selbst.

Die erste Annahme führt zu zwei möglichen Varianten adaptiver Systeme: solche, die zur Laufzeit zwischen mehreren existierenden (d.h. zur Entwicklungszeit definierten) Implementierungen wählen, und solche, die ihre eigene Implementierung dynamisch modifizieren. (Letztere sind auch bekannt unter der Bezeichnung *selbstmodifizierende Systeme*.)

Während die dynamische Veränderung der Zusammensetzung eines Systems realistisch erscheint und zahlreiche Ansätze zur Umsetzung eines solches Vorgehens entwickelt werden, bleibt die Realisierbarkeit selbstmodifizierender Systeme ausgesprochen fraglich. Auch verschiedene Versuche, Erfahrungen aus anderen Wissenschaftsdisziplinen (z.B. der Biologie oder Psychologie) in die Informatik zu übertragen, verliefen bisher weitgehend erfolglos[5]. Im Folgenden konzentrieren wir uns daher auf erstere Variante, also Systeme, die nur existierende Implementierungen dynamisch austauschen.

---

[4] Laddaga [Ladd00] sagt in diesem Zusammenhang (allerdings ohne zwischen den Begriffen „adaptiv“ und „adaptierbar“ zu unterscheiden):

> „Any piece of software that is revised on the basis of problems found in the field can be thought of as adaptive, but with human intervention and very high latency.“

[5] Einen Überblick über die Erfolgsaussichten selbstmodifizierender, adaptiver Systeme gibt z.B. [AnCh04].

Die zweite Annahme, dass ein System ausreichendes Wissen über sich selbst haben muss, bedingt die Existenz eines Modells des gesamten Systems zur Laufzeit. Dieser Aspekt ist unter der – sehr ambitionierten – Bezeichnung *Self-Awareness* (dt. etwa „Selbstbewusstsein“) bekannt. Dabei spielt die Technik der Reflection, die wir in Abschnitt 3.3 diskutieren werden, eine wesentliche Rolle.

Eine spezielle Variante adaptiver Systeme sind *selbstheilende Systeme* (engl. self-healing/self-repairing Systems), wobei die Auslegung des Begriffs in der Literatur sehr stark variiert [Koop03]. Wir verstehen darunter im Folgenden adaptive Systeme, welche Mechanismen zur Durchführung von Fault Recovery und Fault Prevention – wie in Abschnitt 2.2 beschrieben – implementieren. Selbstheilende Systeme stellen somit eine Verbindung adaptiver mit verlässlichen Systemen dar. Ein Informationssystem, welches *alle* genannten Kriterien der Verlässlichkeit und Adaptivität erfüllt, bezeichnen wir als *verlässliches, adaptives Informationssystem* (engl. Dependable Adaptive Information System), kurz DAIS.

2.4 Bedeutung der Systemarchitektur für DAIS

Verlässlichkeit und Adaptivität sind nicht-funktionale Anforderungen, die – wie wir bereits in Abschnitt 1.1 diskutiert haben –, bei der Entwicklung vieler Informationssysteme eine hohe Priorität haben. Neben der sorgfältigen und systematischen Anwendung von Werkzeugen und Methoden kommt der Architektur dabei aus verschiedenen Gründen eine besondere Bedeutung zu:

Die Architekturphase beinhaltet wesentliche Grundsatzentscheidungen, die sowohl Struktur eines Systems als auch die bei seiner Entwicklung zugrundeliegenden Prinzipien bestimmen. (Dies ergibt sich direkt aus der in Abschnitt 2.1 diskutierten Definition des Architekturbegriffs.) Daher ist es wichtig, Verlässlichkeit und Adaptivität bereits auf dieser Ebene zu berücksichtigen, um ihre konsequente Umsetzung im gesamten System sicherzustellen. Gleichzeitig abstrahiert die Architekturebene von Details der jeweiligen Implementierung, und hilft damit, die Komplexität eines DAIS beherrschbar zu halten.

Ein weiterer wichtiger Grund ist die Verwendung von COTS-Komponenten, die zunehmend an Popularität gewinnt und – wie wir in Kapitel 3 noch mehrfach sehen werden – sich gut in den Kontext verlässlicher und adaptiver Systeme einfügt. Bei diesen Komponenten existiert i.d.R. seitens des Verwenders kein verlässliches Wissen über ihre Implementierungsdetails und keine Garantien, dass die betreffende Komponente verlässlich funktioniert. Es ergibt sich also – pointiert formuliert – die Herausforderung, zuverlässige Systeme aus unzuverlässigen Komponenten zu bauen. Die hierfür notwendige Betrachung eines Systems auf einer ausreichend hohen logischen Abstraktionsebene findet sich nur in seiner Architektur.

3 Grundprinzipien von DAIS-Architekturen

In der Literatur existieren zahlreiche Ansätze zur Berücksichtigung von Verlässlichkeit und Adaptivität auf der Architekturebene, die in den meisten

Fällen jedoch nur einzelne Aspekte der beiden Problembereiche abdecken. Allen Ansätzen ist allerdings gemein, dass sie jeweils auf einem oder mehreren einer Reihe von Grundprinzipien von Software-Architekturen aufbauen, die wir daher zunächst diskutieren.

Das wesentliche Grundprinzip, auf dem Architekturen verlässlicher Systeme aufbauen, ist die Vermeidung der Propagierung von Faults an andere Systembestandteile (vgl. z.B. [AvLR01]). Das heißt, ein Fault, der in einem bestimmten Teil des Systems (z.B. einer Komponente) auftritt, sollte möglichst in diesem oder dem direkt übergeordneten Systemteil erkannt und behandelt werden, so dass seine Auswirkungen lokal begrenzt sind.

Darüberhinaus kommt den folgenden Architekturprinzipien eine wesentliche Bedeutung zu:

3.1 Komponentenbasierte Architektur

Eine *(Software-)Komponente* (engl. Component) definieren wir nach Szyperski [BDH+98] als eine Einheit der Komposition mit durch Kontrakt spezifizierten Schnittstellen und ausschließlich expliziten Kontextabhängigkeiten. Eine Software-Komponente kann unabhängig verteilt und zur Komposition durch Dritte verwendet werden. In DAIS finden Software-Komponenten in hohem Maße Anwendung, da sie eine klare Trennung von Funktionalitäten ermöglichen und Abhängigkeiten zwischen Systembestandteilen minimieren. Diese Eigenschaften sind Voraussetzung für eine Austauschbarkeit von Implementierungen, die im Folgenden u.a. bei redundanzbasierten Architekturen eine wesentliche Rolle spielt.

Um (sinnvolle) Systemfunktionalitäten bereitzustellen, müssen Komponenten miteinander interagieren, was einen Konflikt mit dem Ziel der Minimierung von Abhängigkeiten darstellt. Um dennoch eine *lose Kopplung* (Interaktion bei gleichzeitiger Minimierung von Abhängigkeiten) zu erreichen, kommen i.d.R. *Konnektoren* (engl. Connectors) zum Einsatz. Ein Konnektor ist ein Architekturelement, das die Interaktionen zwischen Komponenten kapselt. Er hat nach [BaPl01] die Aufgabe, alle Interaktionen sowohl von Komponenten mit der Umgebung als auch von Komponenten untereinander abzufangen, zu kontrollieren und gezielt zu delegieren. Darüberhinaus fungiert er als Adapter von Schnittstellen und ausgetauschten Datenstrukturen. Die Abläufe innerhalb des Konnektors erfolgen dabei für die beteiligten Komponenten transparent, d.h. diese müssen insbesondere keine Unterstützung für Abläufe innerhalb eines Konnektors implementieren. Aufgrund dieser Merkmale eignen sich Konnektoren gut zur Verwendung in DAIS, wobei sie Teile der im folgenden Abschnitt beschriebenen Kontrolllogik realisieren.

3.2 Separation of Concerns

Das Prinzip der Serparation of Concerns (kurz SoC) erleichert eine klare funktionale Trennung zwischen *logischen* Systembestandteilen und hilft damit, die

Komplexität des Gesamtsystems beherrschbar zu halten. Dies bedeutet allerdings nicht notwendigerweise auch eine Trennung auf der Implementierungsebene, die durchaus verschiedene logische Bestandteile in einer Implementierungseinheit verbinden kann. Durch diese Komplexitätsreduktion steigt auch die Wahrscheinlichkeit einer korrekten Implementierung der für Verlässlichkeit und Adaptivität verantwortlichen Systemteile, welche in DAIS – ähnlich z.B. zu Datenbanksystemen – eine Grundvoraussetzung für die angestrebten Eigenschaften darstellt [GuRL03], [Sha01].

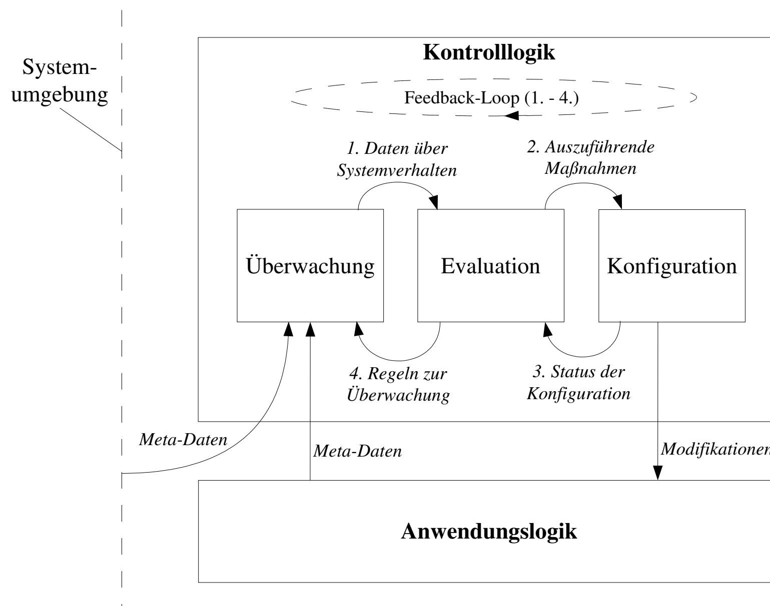

Abbildung 1. Separation of Concerns in verlässlichen, adaptiven Systemen

Abbildung 1 zeigt die Anwendung des SoC-Prinzips in verlässlichen und adaptiven Systemen, die sich vor allem in zwei Bereichen beobachten lässt: zwischen Anwendungs- und Kontrolllogik und innerhalb der Kontrolllogik [RLS00]. Mit *Anwendungslogik* bezeichnen wir im Folgenden die eigentliche Funktionalität einer Anwendung oder Komponente, d.h. die Dienste, die sie zur Verfügung stellt. DAIS unterstellen – bis auf wenige Ausnahmen[6] –, dass die Anwendungslogik weder verlässlich noch adaptiv ist bzw., dass diese Eigenschaften nicht garantiert werden.

[6] Die in [Sha01] beschriebene Simplex-Architektur setzt z.B. explizit die Korrektheit des verwendeten High Assurance Subsystems voraus, zielt damit allerdings gerade auf die Möglichkeit der Verwendung eines nicht verlässlichen High Performance Subsystems ab.

Diese Philosophie ermöglicht, dass sich die (Eigen-)Entwicklung von Komponenten auf Anwendungsfunktionalitäten konzentrieren kann und im Idealfall keine Rücksicht auf nicht direkt mit der Funktionalität der Komponente verbundene Aspekte (wie z.B. Fault Tolerance) nimmt. Dies verringert die Komplexität der Komponenten und damit auch den Aufwand für deren Erstellung und steigert gleichzeitig die Qualität [Sha01]. Auch den Einsatz vorgefertigter Komponenten (z.B. COTS-Komponenten) erleichert diese Reduktion der Annahmen erheblich, da dort i.d.R. nicht (bzw. nur mit hohem Aufwand) die Möglichkeit besteht, Angaben des Herstellers verlässlich zu prüfen oder Anpassungen vorzunehmen.

Den Teil eines Informationssystems, der die für Verlässlichkeit und Adaptivität notwendigen Mechanismen enthält, bezeichnen wir als *Kontrolllogik*. Diese unterteilt sich im Sinne einer Separation of Concerncs in die drei Funktionsbereiche *Überwachung*, *Evaluation* und *Konfiguration*, welche wir im Folgenden *Phasen* nennen. Diese Phasen bilden einen *Feedback-Loop* (z.B. [DiMu03]), der kontinuierlich durchlaufen wird und die Einhaltung der für Verlässlichkeit und Adaptivität wesentlichen Eigenschaften garantiert.

Die Überwachungsphase sammelt Informationen über das System und seine Umgebung, um sie den nachfolgenden Phasen zur Verfügung zu stellen[7]. Die Evaluationsphase[8] wertet die Daten aus und vergleicht sie mit vorhandenen Referenzdaten (z.B. aus Logfiles oder manuell hinterlegten Spezifikationen). Dann prüft sie anhand eines Regelwerks, ob eine Veränderung des Systemzustandes notwendig (etwa im Falle eines Komponenten-Failures) bzw. wünschenswert (z.B. zur Optimierung der Performanz) ist und bestimmt eine zur Durchführung der Änderung geeignete Aktion.

Die Konfigurationsphase verantwortet schließlich die Durchführung der festgelegten Aktion, sofern die Evaluation die Notwendigkeit einer Zustandsänderung erkannt hat. In verlässlichen Systemen lässt sich diese Phase weiter differenzieren in *Recovery*, *Prevention* und *Removal*. (Prevention und Removal entsprechen der Anwendung der in Abschnitt 2.2 vorgestellten Möglichkeiten zum Umgang mit Faults, Recovery ist der Fault Tolerance zuzurechen.)

Recovery bezeichnet die Rückkehr aus einem Fehlerzustand (d.h. dem Auftreten eines Errors) in einen fehlerfreien Zustand, wobei die Vermeidung eines Service Failures höchste Priorität hat. Eine Behebung des zugrundeliegenden Faults erfolgt hingegen *nicht*, d.h. bei erneuter Aktivierung desselben Faults kann der gleiche Error erneut auftreten.

Dabei unterscheiden wir zwei grundsätzliche Arten der Recovery: Forward Recovery und Backward Recovery. Beim *Forward Recovery* versucht das System in einen fehlerfreien Zustand zu gelangen, indem es ausgehend von dem

[7] Dies können z.B. Messdaten über die Ausführungszeit von Methoden, das Auftreten von Failures (Komponenten) oder Daten wie die aktuelle CPU- oder Netzwerklast (Systemumgebung) sein.

[8] In verlässlichen Systemen wird die Evaluation-Phase oft auch als *Detection* bezeichnet, in adaptiven Systemen meist als *Reasoning*. Die dieser Phase zugeordneten Aufgaben unterscheiden sich allerdings nicht.

fehlerhaften Zustand korrigierende Maßnahmen ergreift. Hauptziel ist dabei die Erhaltung bereits verrichteter Arbeit. Eine prominente Implementierung dieses Ansatzes ist Exception Handling, welches wir in Abschnitt 4.7 diskutieren. Beim *Backward Recovery* verwirft es hingegen den fehlerhaften Zustand und setzt sich auf einen früheren, fehlerfreien Zustand zurück [FiGR03].

Die Prevention adressiert das Problem der wiederholten Aktivierung von Faults, indem derjenige Systembestandteil (je nach Granulat der Implementierung z.B. Komponente, Klasse, Methode) lokalisiert wird, der den betreffenden Fault enthält, und seine erneute Ausführung verhindert wird. Dies geschieht bspw. durch das Markieren des Systembestandteils als fehlerhaft oder seinen Austausch durch eine (vermutlich) fehlerfreie Ersatzimplementierung.

Nachteilig an der Prevention ist vor allem, dass sie nur bis zu einer gewissen Anzahl an Faults angewendet werden kann und die Flexibilität des Systems durch das effektive Entfernen von Implementierungsteilen immer weiter reduziert[9]. Noch weiter geht daher das Fault Removal, das nicht nur den Fault lokalisiert, sondern diesen durch Modifikation des betreffenden Systemteils behebt.

Removal erfordert selbstmodifizierende Systeme (vgl. Abschnitt 2.3) und wird daher in existierenden Architekturen nicht unterstützt. Stattdessen stellen einige DAIS-Architekturen Prevention-Mechanismen bereit, die optional um ein Fault-Logging und ein darauf basierendes manuelles Removal außerhalb des Systems erweitert sind.

In nahezu allen Architekturen findet sich die beschriebene Umsetzung des SoC-Prinzips anhand der Trennung zwischen Anwendungs- und Kontrolllogik und der Aufspaltung der Kontrolllogik in mehrere Phasen. (In der Regel ist die Umsetzung leicht abgewandelt und/oder mit abweichenden Benennungen gekennzeichnet. Vgl. z.B. [DiMu03], [AvLR01], [RLS00]) Daneben verwenden redundanzbasierte Systeme meist eine *Proxy-Komponente*, welche die Delegation von Systemaufrufen an Komponenten kapselt und ggf. die Kommunikation von Komponenten mit anderen Systemteilen oder externen Systemen kontrolliert.

3.3 Architectural Reflection

Zur Realisierung adaptiver Systeme ist es – wie in Abschnitt 2.3 beschrieben – notwendig, dass ein Informationssystem über ein Modell seiner selbst verfügt und dieses zur Laufzeit verändern kann. Die grundsätzliche Fähigkeit eines Systems, sein Verhalten zu überwachen und zu ändern ist bekannt als *Computational Reflection* [GaRu01]. Entsprechend definieren wir die Fähigkeit eines Systems, seine Architektur zu überwachen und zu verändern als *Architectural Reflection* [TSCS00].

Die Architektur eines reflektiven Systems lässt sich in zwei Bereiche untergliedern: die *Base-Level-Architektur* und die *Meta-Level-Architektur* [GaRu01].

[9] Wird bspw. die weitere Verwendung einer Komponente aufgrund eines Faults durch den Prevention-Mechanismus unterbunden, verliert das System gleichzeitig auch alle durch diese Komponente eingebrachten Vorteile (z.B. hohe Performanz in bestimmten Szenarien).

Die Base-Level-Architektur enthält Objekte, welche die Funktionalität des Systems implementieren. Auf dieser Ebene ist kein Wissen über die Existenz einer Meta-Ebene vorhanden und daher auch keine Unterstützung der Reflection-Architektur in den Implementierungen dieser Ebene erforderlich.

Die Meta-Level-Architektur besteht aus Objekten, welche die Base-Level-Architektur überwachen und ggf. verändern. Dabei erfolgt eine Abbildung von Eigenschaften aus der Base- in die Meta-Ebene, genannt *Reification*, und die umgekehrte Abbildung, genannt *Reflection*. Die Schnittstelle zwischen Base- und Meta-Level-Architektur bezeichnet man als *Meta-Object Protocol*. Sie gewährt in generischer Weise Zugriff auf Implementierungskonstrukte und ist von der jeweils benutzten Plattform abhängig. Abbildung 2 veranschaulicht das Konzept.

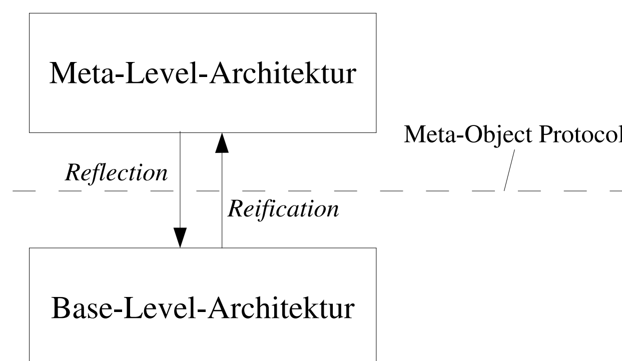

Abbildung 2. Reflection auf Architektur-Ebene

Bezogen auf DAIS-Architekturen lässt sich Reflection leicht auf die in Abschnitt 3.2 beschriebenen logischen Bereiche abbilden: Die Base-Level-Architektur enthält die Anwendungslogik, welche ohne Kenntnis etwaiger Kontrollkonstrukte arbeitet. Die Kontrolllogik residiert in der Meta-Level-Architektur und steuert Vorgänge in der Meta-Ebene gemäß festgelegter Regeln. Beispielsweise stellt die Kontrolllogik eine suboptimale Performanz in der Anwendungslogik fest und verändert daraufhin die zur Anwendung kommende Strategie, indem sie eine bestimmte Implementierung in der Base-Ebene durch eine andere ersetzt.

3.4 Redundanz

Als *Redundanz* bezeichnen wir im Folgenden das mehrfache Vorhandensein funktionsgleicher Systembestandteile. Dabei lassen sich zwei Arten der Redundanz unterscheiden: identische Redundanz und diversifizierende Redundanz.

In ersterem Fall sind nicht nur die Funktionen redundanter Systembestandteile, sondern auch deren Implementierungen identisch. In Softwaresystemen wird

diese Art der Reundanz zur Steigerung der Verlässlichkeit nur sehr begrenzt eingesetzt, da sie offensichtlich nur bei nicht nicht dauerhaften und nicht implementierungsbedingten Faults Wirkung zeigt [Aviz95], [AvLR01]. Adaptive Systeme können mit diesem Ansatz z.B. auf eine schwankende Zahl von Benutzeranfragen reagieren (etwa durch Komponenten-Pooling). Erweiterte Anpassungen (z.B. an eine veränderte *Art* von Anfragen) sind jedoch nicht möglich.

Die zweite Art der Redundanz ist hingegen weit verbreitet und vor allem bekannt unter der Bezeichnung *Design Diversity*. Hierbei werden bewusst unterschiedliche Implementierungen der gleichen Funktionalität herangezogen, um auch Faults, die aus dem Design oder der Umsetzung eines Systembestandteiles resultieren (d.h. unter gleichen Umweltbedingungen mit hoher Wahrscheinlichkeit erneut auftreten), vorzubeugen [Aviz95].

Aufgrund der hohen Effizienz und der Mächtigkeit des Ansatzes kommt der Redundanz in DAIS eine hohe Bedeutung zu. Allerdings kam sie über lange Zeit nur bei sehr hohen Verlässlichkeitsanforderungen zum Einsatz, da die Entwicklung von zwei oder mehr funktionsgleichen Implementierungen eines Systembestandteiles mit hohen finanziellen Aufwendungen einhergeht. Durch die steigende Verbreitung von COTS-Komponenten findet diversifizierende Redundanz in jüngerer Zeit jedoch zunehmend Verbreitung, da das Kosten/Nutzen-Verhältnis beim Kauf mehrerer, funktionsgleicher Komponenten verglichen mit dem einer Eigenentwicklung einer einzelnen, vergleichbar verlässlichen Komponente sehr positiv ist, wie bspw. Popov et. al. in [PoSR00] beschreiben.

Auch zur Realisierung adaptiver Systeme lässt sich Design Diversity einsetzen. So ist es nach [DiMu03] bspw. sehr komplex, einzelne Komponenten zu bauen, welche unter allen Umständen die optimale Performanz aufweisen; hingegen lassen sich mit deutlich geringerem Aufwand Mengen von Komponenten entwicklen, innerhalb derer jede Komponente auf den Einsatz in einem klar abgegrenzten Szenario optimiert ist und welche – den korrekten Austausch der Komponenten zur Laufzeit vorausgesetzt – die gleiche Gesamtperformanz aufweisen.

3.5 Realisierungsprobleme

Bei der Anwendung der in diesem Kapitel beschriebenen Prinzipien ergeben sich eine ganze Reihe von Problemen, auf die wir an dieser Stelle eingehen wollen. Ansätze zu deren Lösung, wie sie in den im Folgenden vorgestellten Architekturen zum Einsatz kommen, liegen außerhalb des Umfangs dieser Arbeit, so dass wir sie jeweils nur kurz skizzieren.

Ein Problembereich ist die Analyse von Ergebnissen redundanter Komponenten: entweder muss ein Ergebnis auf die Erfüllung einer Spezifikation getestet oder mit den Ergebnissen anderer Komponenten verglichen werden. In ersterem Fall erfordert dies sowohl das Vorhandensein einer solchen Spezifikation als auch effiziente Algorithmen, um einen Ergebniswert zur Laufzeit gegen die Spezifikation zu testen (Vgl. dazu auch Abschnitt 4.2). In letzterem Fall bedingt es insbesondere die Existenz einer „equals"-Operation auf allen in einer Anwendung eingesetzten Datentypen [RaMe01]. Dabei ist inbesondere die Verwendung

komplexer Datentypen (etwa Listen, Mengen oder auch Gleitkommazahlen) problematisch, da der Vergleich schwierig ist bzw. hohen Rechenaufwand erfordert.

Bei dem Vergleich der Ergebnisse mehrerer Komponenten stellt sich zudem das Problem, *einen* Ergebniswert zu bestimmen, der (mit ausreichend hoher Wahrscheinlichkeit) korrekt ist und an die Anwendung zurückgegeben wird. Hierbei lassen sich zwei Arten von Lösungen unterscheiden: *autoritative Verfahren* und *abstimmungsbasierte Verfahren*. Bei autoritativen Verfahren wird eine Komponente als maßgebend (engl. authoritative) gekennzeichnet und ihre Ergebnisse als korrekt angenommen. Die Auswahl der maßgebenden Komponente geschieht i.d.R. aufgrund von Auswertungen über die Qualitiät der Ergebnisse aller vorhandenen Komponenten in der Vergangenheit. [RaMe01]

Bei abstimmungsbasierten Verfahren (engl. Majority Voting) wird derjenige Ergebniswert verwendet, den die Mehrheit der vorhandenen Komponenten zurückliefert. Die Ergebnisse einzelner Komponenten können optional zusätzlich gewichtet werden, um die Qualität der Auswertung zu verbessern[10]. Ist die Annahme, dass die Mehrheit der vorhandenen Komponenten korrekte Ergebnisse liefert, in einem bestimmten Anwendungskontext falsch, liefert dieses Verfahren Ergebniswerte, die in hohem Maße unzuverlässig sind. Darüberhinaus muss die oben beschriebene algorithmische Vergleichbarkeit der Ergebnisse gewährleistet sein. [DuSN05], [Sha01]

Ein weiteres Problem ist die Isolation von Komponenten: um inkonsistente Anwendungszustände zu vermeiden, müssen neben der beschriebenen Filterung der Ergebniswerte auch alle anderen Interaktionen von Komponenten mit ihrer Umgebung kontrolliert werden [RaMe01]. In der Regel darf nur jeweils eine Komponente (z.B. eine als maßgeblich gekennzeichnete) tatsächlich mit ihrer Umgebung kommunizieren, während die Aufrufe anderer Komponenten etwa von einem Konnektor abgefangen werden[11]. Allerdings ist in vielen Fällen eine Kommunikation von Komponenten mit anderen Systemteilen erforderlich, damit diese überhaupt Ergebnisse produzieren können. Zum Vergleich verschiedener Implementierungen ist daher nur eine eingeschränkte Regulierung der Kommunikation möglich. Eine mögliche Lösung könnte in diesem Zusammenhang die Simulation der jeweiligen Kommunikationspartner durch einen Konnektor sein, was aber wiederum eine hohe Komplexität desselben zur Folge hat.

Auch die eigentliche Verbindung unterschiedlicher Implementierungen durch Konnektoren birgt Probleme in sich. Insbesondere ist eine wesentliche Grundvoraussetzung für die Anwendbarkeit von diversifizierender Redundanz, dass die Schnittstellen der einzelnen funktionsgleichen Komponenten eine ausreichende Ähnlichkeit aufweisen, um sie auf eine einheitliche, externe Schnittstelle (die des

[10] Beispielsweise kann für jede Funktionalität aufgezeichnet werden, welche Komponente mit ihren Ergebnissen wie oft von dem als korrekt ermittelten Wert abweicht, und die Ergebnisse von Komponenten mit hoher Fehlerfrequenz schwächer gewichtet werden.

[11] Betrachten wir z.B. redundante Komponenten, welche eine Hotelreservierung über Web Services vornehmen, so darf die eigentliche Reservierung nur ein einziges Mal (nämlich von der aktuell als korrekt angenommenen Komponente) durchgeführt werden.

Konnektors zur Anwendung) abzubilden. Dies betrifft sowohl die jeweils bereitgestellten Methoden als auch die ausgetauschten Datentypen. Es ergibt sich daher das Dilemma, dass einerseits möglichst große Abweichungen in den einzelnen Komponenten angestrebt werden, um die Verlässlichkeit zu erhöhen, diese aber andererseits möglichst ähnlich implementiert sein müssen, um eine Austauschbarkeit zu gewährleisten. Die praktische Umsetzung erfordert daher sorgfältige Abwägung zwischen beiden Zielen und das Finden geeigneter Kompromisse.

Eine weitere kritisch zu betrachtende Annahme im Konzept der diversifizierenden Redunanz ist die Unabhängigkeit von Faults [Sha01]. Konkret bedeutet diese Annahme, dass wenn eine Implementierung einen bestimmten Fault enthält, andere Implementierungen diesen Fault mit hoher Warscheinlichkeit nicht enthalten. Dies ist insofern fraglich, als sich bei vielen Problemstellungen Muster (sowohl auf Design- als auch auf Implementierungsebene) herausgebildet haben, die oft sogar gezielt Anwendung finden[12]. Es besteht daher die Gefahr, dass mehrere funktionsgleiche Implementierungen zwar unabhängig voneinander entwickelt werden, aber trotzdem zumindest in Teilen Ähnlichkeiten aufweisen, die zum Vorhandenseins des gleichen Faults in (scheinbar) unterschiedlichen Implementierungen führen. Vorschläge zur Lösung dieses Problems beinhalten z.B. die Anwendung explizit unterschiedlicher Vorgehensmodelle und Entwicklungsmethoden, um eine Diversifizierung von Implementierungen zu erzwingen.

Schließlich stellt sich das Problem des Transfers von Zuständen zwischen redundanten Komponenten: wird eine Komponente durch eine andere ersetzt, so muss ihr Zustand auf die neue Komponente übertragen werden, um den Vorgang gegenüber der Anwendung zu kapseln[13]. Durch die Diversifikation weicht die Speicherung von Zuständen zwischen verschiedenen Implementierungen notwendigerweise ab. Gleichzeitig ist die Zustandsverwaltung i.d.R. komplett innerhalb einer Komponente gekapselt und somit nicht für das umgebende System sichtbar. Zur Lösung dieses Problems existieren zahlreiche Ansätze, die allerdings allesamt gravierende Nachteile aufweisen. Im Folgenden skizzieren wir beispielhaft einige prominente Varianten.

Ein möglicher Ansatz ist das Logging von Anwendungsaufrufen auf der Ebene der generischen Konnektorschnittstelle[14]. Da ohnehin eine Abbildung dieser Schnittstelle auf die der einzelnen Komponenten existiert, ist die Zustandsverfolgung damit unabhängig von den einzelnen Implementierungen. Dies setzt allerdings voraus, dass die Aufrufe in sehr kurzer Zeit (nämlich bei der Instantierung neuer Komponenten) abgearbeitet werden können und keine Abhängigkeiten zu

[12] Beispielsweise Entwurfsmuster und Idiome, deren Anwendung i.d.R. als „guter Stil" betrachtet wird.

[13] Bei zustandslosen Komponenten stellt sich dieses Problem offensichtlich nicht. In realen Systemen lässt sich die Verwendung zustandsbehafteter Komponenten jedoch nur in wenigen Fällen vermeiden, so dass dem Problem eine hohe Bedeutung zukommt.

[14] Dieses Verfahren gleicht dem Logging von Datenbankoperationen in Datenbanksystemen, die ggf. wiederholt ausgeführt werden können, um das Erreichen eines bestimmten Zustandes sicherzustellen.

anderen Teilen des Systems aufweisen. Entfällt z.B. ein Großteil des Zeitaufwandes innerhalb einer Anwendung auf die Bearbeitung von Anfragen in den einzelnen Komponenten (Rechenzeit), funktioniert dieses Verfahren nicht.

Eine weitere Möglichkeit, bekannt als *aktive Replikation* (engl. Active Replication), ist die Vermeidung von Zustandstransfers durch den parallelen Einsatz aller verfügbaren Komponenten [DuSN05]. In diesem Fall existiert ein Proxy (z.B. ein Konnektor), der eingehende Aufrufe an alle Komponenten (innerhalb einer Gruppe funktionsgleicher Komponenten), weiterleitet und aus den Ergebnissen eines selektiert. Dies hat insbesondere den Nachteil, dass keine dynamische Instantierung neuer Komponenten möglich ist. Damit unterstützt dieser Ansatz zwar den verlässlichen Austausch einer Komponente (z.B. wenn diese sich in einem Fehlerzustand befindet), nicht jedoch die Adaption des Systems an veränderte Lastbedingungen.

Daneben existieren Ansätze, die Zustände einer Komponente und die Übergänge zwischen diesen durch Adaption des State Patterns explizit zu machen [FeRu98]. Dabei erfolgt die Speicherung des Zustandes nicht mehr innerhalb von Komponenten (also z.B. in Objektvariablen), sondern auf der Meta-Ebene. Die einzelnen Komponenten unterteilen sich dabei in mehrere Subkomponenten, von denen jede genau einen Zustand repräsentiert, die aber selbst keinen Zustand haben. Zustandsübergänge erfolgen dann ebenfalls auf der Meta-Ebene, wobei nach jedem Zustandswechsel die den neuen Zustand repräsentierende Komponente genutzt wird.

Der Vorteil dieses Verfahren ist, dass die Zustandsübergänge sehr einfach und generisch zu bewerkstelligen sind. Nachteilig ist aber, dass alle eingesetzten Komponenten das Muster bei ihrer Implementierung berücksichtigen müssen, so dass die Verwendung von COTS-Komponenten nahezu unmöglich ist. Darüberhinaus lässt sich der Ansatz nur bei einer relativ kleinen Anzahl an Zuständen sinnvoll umsetzen, da für jeden Komponentenzustand eine eigene Implementierung erforderlich ist.

4 Architekturen verlässlicher Systeme

Zur Berücksichtigung von Verlässlichkeit auf der Architekturebene finden sich in der Literatur eine Vielzahl von Ansätzen, welche auf den im vorangegangenen Kapitel beschriebenen Prinzipien beruhen. Im Folgenden werden wir diese Ansätze kurz erläutern und ihre Unterschiede diskutieren.

4.1 N-Version-Software

Die Architektur sog. N-Version-Software, kurz NVS, beruht auf Ideen aus redundanten Hardware-Systemen. Hierbei werden laut [Aviz95] zwei oder mehr funktionsgleiche, nicht fehlertolerante Software-Einheiten (auch Simplex-Einheiten oder Versionen genannt) parallel über sog. Computing Lanes angesteuert und ihre Ergebnisse von einem Controller abstimmungsbasiert (im Majority-Voting-Verfahren) verglichen (Vgl. Abschnitt 3.5). An die Umgebung wird nur das als korrekt eingestufte Ergebnis weitergereicht, wie Abbildung 3 veranschaulicht.

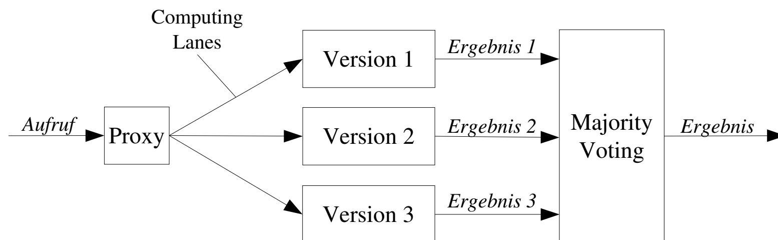

Abbildung 3. NVS-Architektur mit 3 Versionen

Avizienis betont explizit, dass es sich bei den Simplex-Einheiten (im Gegensatz zu Hardware-Einheiten) nicht um identische Implementierungen handeln darf, da andernfalls ein Vergleich der Ergebniswerte keinen Hinweis auf deren Korrektheit liefern kann (Vgl. Abschnitt 3.4). Problematisch ist zudem, dass mindestens drei unterschiedliche Implementierungen jeder Software-Einheit vorhanden sein müssen, um im Fall nicht-übereinstimmender Ergebniswerte den (wahrscheinlich) korrekten Wert bestimmen zu können. Da das Verfahren zudem keinerlei Lösungsansätze zu den in Abschnitt 3.5 erläuterten Problemen beinhaltet, ist es für praktische Anwendungen nicht relevant. Es stellt jedoch ein gutes Beispiel für die Grundideen verlässlicher Systeme dar.

4.2 Recovery Blocks

Recovery Blocks verfolgen nach [Aviz95] einen der N-Version-Software sehr ähnlichen Ansatz. Der wesentliche Unterschied liegt in der Durchführung der Ergebnisüberprüfung, die im Gegensatz zur NVS nicht in einem Vergleich mehrerer Ergebnisse besteht, sondern lediglich in einen Akzeptanztest. Dabei wird eine von N verfügbaren Software-Einheiten zur Berechnung eines Wertes herangezogen und im Anschluss getestet, ob sich der Ergebniswert in einem als zulässig definierten Bereich bewegt. Ist dies nicht der Fall, wird eine andere Software-Einheit zur Berechnung herangezogen und der Wert wiederum getestet. Dies geschieht so lange, bis entweder ein als gültig eingestufter Wert zurückgeliefert wird oder keine alternative Einheit mehr zur Verfügung steht. (In letzterem Fall wird eine anwendungsspezifische Ausnahmebehandlung ausgelöst.)

Im Gegensatz zur N-Version-Software setzen Recovery Blocks noch zwei kritische Annahmen voraus. Der Ansatz verlangt, dass zur Entwicklungszeit sowohl eine exakte Spezifikation des korrekten Systemverhaltens vorliegt als auch Algorithmen, um einen beliebigen Ergebniswert zur Laufzeit auf deren Einhaltung zu testen [DiMu03]. Erstere Annahme ist nur in sehr einfachen Systemen haltbar und lässt den Ansatz daher für aktuelle Softwaresysteme unrealisitisch erscheinen. Die zweite Annahme ist ebenfalls problematisch, da sie – wie bereits in Abschnitt 3.5 erläutert – nur schlecht auf komplexe Datentypen anwendbar ist bzw. hohen Rechenaufwand erfordert.

Darüberhinaus setzen Recovery Block die Korrektheit des Test-Algorithmus voraus, welche in praktischen Anwendungen i.d.R. jedoch nicht verifizierbar ist. In bestimmten Anwendungen ist zudem die Korrektheit von einzelnen Ergebniswerten nachweislich unentscheidbar [Sha01][15]. Daraus resultiert die Gefahr, dass der Test-Algorithmus Unschärfen aufweist, welche u.U. bedingen, dass fehlerhafte Werte nicht als solche erkannt werden und somit zu einer Abweichung des Folgezustandes (d.h. einem Error) führen [Aviz95].

4.3 Multi-Versioning Connectors

Einen Spezialfall von N-Version-Software stellen *Multi-Versioning Connectors*, kurz MVC[16], dar. Hierbei liegt das Hauptaugenmerk darauf, die Einführung neuer Versionen *einer* Komponente auf der Architekturebene zu unterstützen, um so die Verlässlichkeit eines sich (kontinuierlich) verändernden Systems zu gewährleisten.

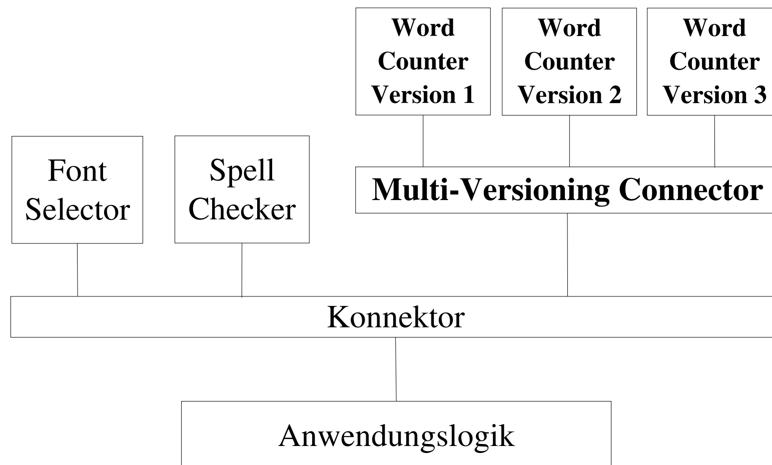

Abbildung 4. Beispiel eines MVC in einem Text-Editor (In Anlehnung an [RaMe01])

Die Umsetzung des MVC geschieht in Form eines Wrappers, welcher eine oder mehrere Versionen einer Komponente kapselt und deren Funktionalität einer Anwendung zur Verfügung stellt. (Abbildung 4 zeigt diesen Aufbau beispielhaft anhand einer Komponente, die innerhalb eines Text-Editors das Zählen der Wörter übernimmt.) Aufrufe der Anwendung an den MVC leitet dieser an alle vorhandenen Versionen weiter und wertet deren Ergebnisse aus. Rakic und

[15] Sha [Sha01] führt als einfaches Beispiel die Erzeugung gleichverteilter Zufallszahlen an, bei der offensichtlich nur bei Betrachtung mehrerer Ergebniswerte Aussagen über die Qualität einzelner Ergebniswerte möglich sind.

[16] Nicht zu verwechseln mit dem Model-View-Controller-Entwurfsmuster.

Medvidovic [RaMe01] schlagen eine autoritätsbasierte Selektion der Ergebnisse auf Komponenten- oder Methodenebene vor (Vgl. Abschnitt 3.5). Dabei wird genau eine Version in einem MVC (bzw. pro Methode innerhalb eines MVC) als maßgebend gekennzeichnet und ihre Ergebnisse als korrekt angenommen. Um inkonsistente Anwendungszustände zu vermeiden, sind zudem alle weiteren Interaktionen mit der Umgebung auf die jeweils maßgebende Komponente beschränkt.

Die Zuordnung der Authorität an eine Komponente wird manuell vorgenommen[17]. Als Grundlage dafür dienen Log-Daten, welche der MVC generiert. Diese geben bspw. Aufschluss über die von den einzelnen Versionen gelieferten Ergebnisse zu jedem Methodenaufruf, die Performanz und die Häufigkeit von Failures einzelner Versionen. Erweist sich eine neue Version aufgrund dieser Informationen als stabil und qualitativ überlegen (d.h. sie führt insbesondere keine neuen Errors ein), kann sie als maßgebend gekennzeichnet und ihre Vorgänger aus dem System entfernt werden.

Damit ist der MVC-Ansatz – im Gegensatz zu N-Version-Software im Allgemeinen und Recovery Blocks – klar auf eine *temporäre* Parallelität verschiedener Komponentenversionen ausgelegt. Das heißt, das Ziel besteht zu jedem Zeitpunkt darin, eine Version einer Komponente durch eine bessere Version zu ersetzen und damit ultimativ eine einzige Komponentenversion zu erreichen, welche unter allen Umgebungsbedinungen optimal funktioniert. Eine mögliche Optimierung und Komplexitätsreduktion durch den gezielten parallelen Einsatz mehrerer Versionen bleibt damit unberücksichtigt.

Neben den bei redundanzbasierten Ansätzen inhärenten Problemen (wie Isolation von Zustandsänderungen, Ähnlichkeit der Schnittstellen, usw.) gelten auch bei MVC die gleichen kritischen Annahmen bzgl. der Spezifikation des Komponentenverhaltens und der Vergleichbarkeit von Ergebniswerten. Damit ergibt sich insgesamt eher eine Methodik zur Unterstützung des Entwicklungsprozesses von Komponenten, als ein eigenständiger Ansatz für DAIS-Architekturen.

4.4 RAIC

Liu et al. schlagen vor, die Redundanz von Komponenten in *Redundant Arrays of Independent Components* (kurz RAIC) zu kapseln. In [LiRi02] definieren sie ein RAIC als eine Gruppe ähnlicher oder identischer Komponenten, welche der Anwendung Dienste von Komponenten innerhalb der Gruppe zur Verfügung stellt. Für die Anwendung stellt sich ein RAIC dabei als einzelne Komponente dar, welche die Komplexität der Delegation an Subkomponenten verbirgt (Vgl. Abschnitt 4.3).

Ein Controller innerhalb des RAICs übernimmt die Aufgabe, Aufrufe der Anwendung an eine oder mehrere Komponenten zu delegieren und deren Rückgaben

[17] Rakic und Medvidovic [RaMe01] regen eine Automatisierung der Komponentenbewertung und des Versionenaustauschs in Form von Heuristiken an. Da der Entwicklungsprozess der Komponenten nach wie vor zwingend manuell verläuft, erhöht diese Erweiterung allerdings nicht die Mächtigkeit des Ansatzes insgesamt.

zu bewerten und transformieren. Dabei treten die gleichen Probleme auf, die wir sie bereits im vorangegangen Abschnitt diskutiert haben. Des Weiteren unterscheiden Liu et al. in Abhängigkeit von der Art der Komponentenredundanz verschiedene RAIC-Level und in Abhängigkeit der Komponentenparallelität verschiedene Aufrufmodelle (Sequentiell, Synchron parallel, Asynchron parallel).

Das Hinzufügen (bzw. Entfernen) von Komponenten in einem RAIC kann entweder statisch, d.h. als Teil des Deployments, oder dynamisch, d.h. während der Laufzeit des Systems vorgenommen werden. Im dynamischen Fall benutzt der RAIC-Controller Verzeichnisdienste, um verfügbare Komponenten zu ermitteln und einzubinden. Die Besonderheit des RAIC-Konzepts liegt darin, dass sowohl beim Hinzufügen neuer Komponenten als auch während des laufenden Betriebs mit vorhanden Komponenten *Just-in-Time Component Testing* (kurz JIT) zum Einsatz kommt. Dabei handelt es sich entweder um vordefinierte Unit-Tests, die bei dem Einbinden der Komponente durchgeführt und ausgewertet werden, oder um Tests mit Live-Daten parallel zu einer anderen, bereits eingebundenen Komponente.

Der RAIC-Controller wertet aufgrund der Testfälle aus, ob eine Komponente voraussichtlich verlässlich funktioniert (d.h. keine Errors auftreten) und überprüft gleichzeitig, wie effizient sie arbeitet. Zudem persistiert der Controller die Testergebnisse in Form von Meta-Daten zu jeder Komponente. Dadurch entstehen zwischen den verschiedenen Komponenten Beziehungen (etwa in Bezug auf Fehleranfälligkeit, Aufrufkosten, Performanz, etc.), welche er beim Austausch von Komponenten berücksichtigt. Ziel ist es, dass nur die beste (bezogen auf die jeweils definierten Kriterien wie z.B. Zahl von Errors oder Ausführungszeit von Methoden) Komponente Anwendungsaufrufe bearbeitet und so optimale Verlässlichkeit und Performanz gewährleistet ist.

Problematisch bei der kontinuierlichen Ausführung von Tests im Live-Betrieb ist der Einfluss auf die Performanz des Gesamtsystems. Zwar kommt im Idealfall nur die jeweils effizienteste Komponente zum Einsatz, das System muss aber darüberhinaus den Overhead durch das Testen neuer Komponenten und die permanente Erhebung und Kontrolle von Daten zur aktuellen Komponente bewältigen. Im Fall vordefinierter Testfälle ist zudem die Aussagekraft der Testergebnisse fraglich.

Darüberhinaus nimmt das RAIC-Konzept keinerlei Separation of Concerns innerhalb der Kontrolllogik vor: alle in Abschnitt 3.2 beschriebenen Phasen sind im RAIC-Controller gekapselt und eng verwoben. Dieser weist daher eine sehr hohe Komplexität auf und ein Austausch von Implementierungsteilen oder sogar die Verwendung verschiedener Strategien (z.B. zum Auffinden oder Austausch von Komponenten) ist mit hohem Änderungsaufwand und Unsicherheit verbunden. Zudem erfordert jede Änderung an der Kontrolllogik zwangsläufig ein neues Deployment des Systems.

Des Weiteren ist die Evaluationslogik sehr rudimentär. Der RAIC-Controller unterstellt, dass genau eine Komponente in allen Szenarien optimal funktioniert. Eine Unterscheidung nach verschiedenen Umgebungsbedingungen findet nicht statt. Auch beschränkt sich das Granulat der Überwachung auf Komponenten,

nicht auf z.B. Methoden. Damit unterstellt das RAIC-Konzept, dass in einer Komponente jeweils *alle* Methoden homogene Charakteristiken in Bezug auf die zugrundegelegten Kriterien aufweisen. Eine Konfiguration von Kriterien und deren Gewichtungen zum Austausch von Komponenten durch den Systemverwalter (außer durch Modifikation des Controller-Codes) unterstützt die Architektur ebenfalls nicht.

Ingesamt ist der Ansatz zwar eine deutliche Erweiterung gegenüber den NVS-artigen Architekturen, er weist jedoch eine Vielzahl von Einschränkungen und Nachteilen auf, die aus der simplistischen Architektur resultieren. Insbesondere die geringe Flexibilität der Kontrolllogik wiegt dabei schwer. Für den praktischen Einsatz in realen Systemen scheint die RAIC-Architektur daher nicht geeignet.

4.5 Component Redundancy

Das Konzept der *Component Redundancy*, vorgestellt in [DiMu03], erweitert die Ansätze von RAICs und N-Version-Software in mehrfacher Hinsicht. Im Gegensatz zu den bisher diskutierten Architekturen unterstellt es explizit *nicht*, dass das Systemverhalten in allen zur Laufzeit möglichen Szenarien zur Entwicklungszeit verlässlich spezifiziert werden kann und berücksichtigt darüberhinaus die Evolution von Informationssystemen nach dem ursprünglichen Deployment. Zudem verfolgt Component Redundancy die Strategie der Design Diversity (Vgl. Abschnitt 3.4), indem unterschiedliche Komponenten gleicher oder ähnlicher Funktionalität in Abhängigkeit vom jeweiligen Systemkontext eingesetzt werden.

Hierzu werden eine oder mehrere (funktional äquivalente aber unterschiedlich implementierte) Komponenten[18] in einer sog. *Redundancy Group* zusammengeschlossen, welche gegenüber der Anwendung als Wrapper fungiert, d.h. für diese als eine einzelne Komponente sichtbar ist. Dabei delegiert der Wrapper zu jedem Zeitpunkt Anfragen der Anwendung an genau eine der verfügbaren Komponenten (*Aktive Komponente)*, während die anderen Komponenten keine Aktionen ausführen (*Passive Komponenten*).

Abbildung 5 zeigt den Aufbau der Architektur. Der Wrapper besteht aus drei logischen Tiers, dem Monitoring Tier, dem Evaluation Tier und dem Action Tier, welche den diskutierten Phasen (Überwachung, Evaluation, Konfiguration) entsprechen, und einem Proxy, der die Delegation von Anwendungsaufrufen an die aktive Komponente kapselt. Der Monitoring Tier zeichnet zur Laufzeit Log-Daten über die gerade aktive Komponente und die Systemumgebung auf, welche als Grundlage für den Austausch von Komponenten dienen und in Form von Meta-Daten zu jeder Komponente hinterlegt werden. Optional können bei der Einbindung neuer Komponenten auch manuell Meta-Daten hinterlegt werden, welche als Entscheidungsgrundlage dienen, solange noch keine ausreichenden Messdaten verfügbar sind.

[18] Diaconescu und Murphy [DiMu03] berücksichtigen hierbei explizit den Einsatz von COTS-Komponenten, indem kein Wissen über die Verwendung innerhalb einer Redundany Group in den eingebundenen Komponenten vorhanden sein muss.

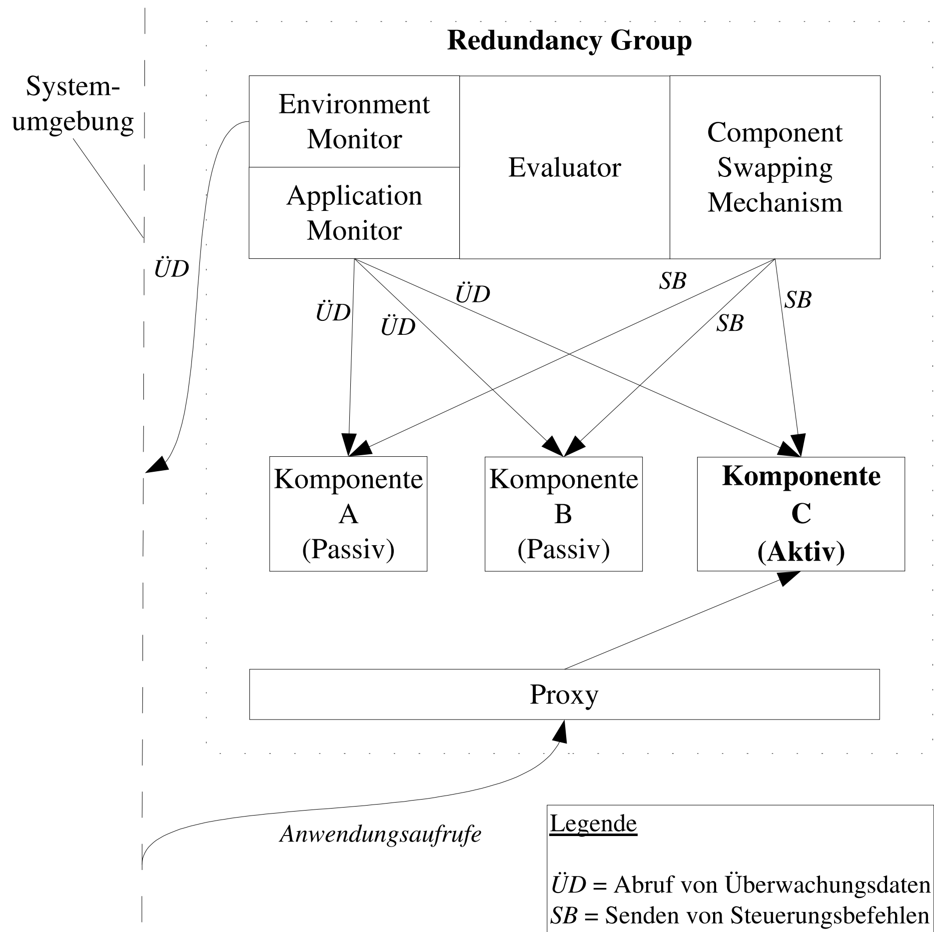

Abbildung 5. Aufbau der Component-Redundancy-Architektur

Der Evaluation Tier implementiert ein Regelwerk, das vorschreibt, beim Eintritt welcher Bedingungen (bezogen auf die vom Monitoring Tier bereitgestellten Informationen) welche Aktion (d.h. insbes. Austausch von Komponenten) zu erfolgen hat. [DiMu03] unterteilen den Evaluation Tier dabei nochmals in zwei Schichten, den *Top Layer* und den *Bottom Layer*. Der Top Layer verantwortet ausschließlich die Behandlung von Errors und Failures der aktiven Komponente, während der Bottom Layer versucht, die Performanz der Redundancy Group aufgrund der vorhandenen Meta-Daten zu den Komponenten zu optimieren

Im Action Tier erfolgt schließlich die eigentliche Verwaltung der verfügbaren Komponenten sowie insbesondere deren dynamischer Austausch zur Laufzeit. Dieser Tier implementiert auch die in Abschnitt 3.2 beschriebenen Aufgaben in Bezug auf die Verwaltung redundanter Komponenten wie Zustandstransfers etc.

Problematisch bei diesem Ansatz ist vor allem die Skalierbarkeit, insbesondere bei der hierarchischen Anordnung von Redundancy Groups in größeren Systemen. Zum einen entsteht ein hoher Overhead, da auf jeder Hierarchiestufe eine permanente Evaluierung der Komponentenfunktion und deren Performanz erfolgt und ggf. Aktionen zu deren Optimierung durchgeführt werden, so dass eine hochgradige Überlappung der entsprechenden Überwachungsmechanismen vorliegt. (Dies stellt letztlich einen Trade-Off zwischen Verlässlichkeit – im Sinne möglichst vieler Überprüfungen – und Performanz dar.) Zum anderen besteht (wie in [DiMu03] beschrieben) die Gefahr, dass eine lokale Optimierung (d.h. eine Optimierung auf der Ebene einzelner Komponenten im System) keine Optimierung für das Gesamtsystem darstellt oder dessen Leistung sogar verschlechtert[19]. Wechselwirkungen zwischen verschiedenen Optimierungsmaßnahmen finden ebenfalls keine Berücksichtigung.

Ebenfalls kritisch zu betrachten ist die implizite Annahme, dass alle Komponenten innerhalb einer Redundancy Group über ausreichend ähnliche Schnittstellen verfügen, um diese in generischer Weise anzusteuern bzw. gegenüber der Anwendung zu kapseln (vgl. Abschnitt 3.5). Insbesondere da die Component-Redundancy-Architektur explizit die Verwendung von COTS-Komponenten (also i.d.R. auch funktionsgleiche Komponenten unterschiedlicher Hersteller) berücksichtigt, erscheint diese Einschränkung fragwürdig.

4.6 Simplex-Architektur

Sha [Sha01] konzentriert sich auf die Steigerung *nachweislicher* Verlässlichkeit auf der Architekturebene. Er kritisiert, dass die Annahme der Unabhängigkeit von Faults (vgl. Abschnitt 3.5), welche bei den bisher diskutierten Verfahren zugrundeliegt, sich i.d.R. nicht nachweisen lässt und eine Nicht-Erfüllung dieser

[19] [DiMu03] schlagen zur Lösung des Problems der globalen Optimierung eine Verteilung des Überwachungsmechanismus ähnlich dem Exception Handling in Programmiersprachen vor. Dabei werden lokal nur solche Aktionen ausgeführt, die kritische oder mit hoher Wahrscheinlichkeit suboptimale Leistungen der Redundancy Group beheben, während alle sonstigen Entscheidungen an hierarchisch übergeordnete Stellen delegiert werden.

Voraussetzung gravierende und nur schwer abzuschätzende Auswirkungen haben kann, welche bei kritischen Systemen nicht tolerierbar sind.

In [Sha01] schlägt er daher einen anderen Ansatz – bekannt als *Analytische Redundanz* oder *Simplex-Architektur* – vor, der vorrangig die Komplexität von Software-Komponenten als kritisch für deren Verlässlichkeit betrachtet[20]. Dabei liegen zwei Annahmen zugrunde: Erstens, dass Design Diversity nicht automatisch zu nachweislich höherer Verlässlichkeit führt, sondern nur unter o.g. Voraussetzungen, und zweitens, dass die Komplexität von Software-Komponenten zu großen Teilen aus Features resultiert, die nicht zur Kernfunktionalität der jeweiligen Komponente zählen, aber wünschenswert sind.

Basierend auf diesen Annahmen besteht die Architektur eines analytisch redundanten Systems aus einem *High-Performance Subsystem* (HPS) und einem *High-Assurance Subsystem* (HAS) (d.h. jeweils einer Komponente), welche durch einen Controller angesteuert werden. Das High-Performance Subsystem ist ohne Rücksicht auf Verlässlichkeit entworfen und auf hohe Performanz und eine große Zahl an über die Kernfunktionalität hinausgehenden Features fokussiert. Das High-Assurance Subsystem hingegen implementiert nur die essentiellen Systemfunktionalitäten und ist entweder verifizierbar oder aufgrund von Erfahrungswerten aus dem Einsatz in anderen Kontexten (i.d.R. über lange Zeit) als verlässlich eingestuft.

Der Controller steuert beide Subsysteme parallel an, wobei er im Normalfall die Ergebnisse der High-Performance-Komponente an das umgebende System propagiert. Dabei überprüft er kontinuierlich, dass die Ergebniswerte innerhalb von der High-Assurance-Komponente vorgegebener Grenzwerte (dem sog. *Stability Envelope*) liegen. Überschreiten die Ergebnisse diese Grenzwerte, so werden temporär die High-Assurance-Ergebnisse verwendet, bis sich die normalen Ergebnisse wieder im zulässigen Bereich bewegen.

Voraussetzung für die Anwendbarkeit dieser Architektur ist die Funktion der beiden Komponenten innerhalb klar definierter Zeitgrenzen, da sonst eine kontinuierliche Verfügbarkeit des Systems trotz permanenter Kontrolle aller HPS-Ergebnisse durch HAS-Ergebnisse nicht möglich ist. Damit ist eine Steigerung der Performanz des Gesamtsystems (trotz der diesbezüglich irreführenden Benennung der Komponenten) nur beschränkt möglich, da die Ausführungszeit jedes Aufrufs des Systems immer im *Maximum* der Ausführungszeiten von HAS und HPS besteht (d.h. die High-Assurance-Komponente stellt i.d.R. einen Flaschenhals dar).

Ebenfalls problematisch ist die Verfügbarkeit von ausreichend verlässlichen HAS-Komponenten, da selbst bei minimaler Funktionalität eine formale Verifikation moderner Systeme kaum möglich ist. Auch das Heranziehen in ausreichen-

[20] Die zugrundeliegende Idee ist das traditionelle KISS-Prinzip („Keep it small and simple“) der Ingenieurswissenschaften – auch als RISC Style bekannt –, also das Erreichen hoher Qualität durch einfaches Design.

der Weise getesteter Komponenten[21] bedingt die Verwendung deutlich älterer Komponenten, die sich wiederum negativ auf die Performanz, Kompatibiliät, Wartbarkeit, usw. des Systems auswirkt. Verringert man das als ausreichend betrachtete Zeitintervall für Tests, so konvergiert die Simplex-Architektur gegen N-Version-Software und büßt damit ihre Vorteilhaftigkeit ein.

Neben den Anforderungen an das HAS weist die Architektur einen weiteren Nachteil auf, der sie primär auf den Einsatz bei kleinen (z.B. eingebetteten) Systemen beschränkt [GRRL03]. Dieser besteht in der nicht vorgesehenen Schachtelung von Komponenten, die u.a. dem Konzept der Komplexititätsreduktion der HAS-Komponente und den Zeitschranken bei der Ausführung widersprächen. Stattdessen überwacht die Architektur das Verhalten des Gesamtsystems, was (wie bei N-Version-Software und ähnlichen Ansätzen) dessen vollständige Spezifikation mit den damit verbundenen Problemen erfordert.

4.7 Exception Handling auf Architekturebene

Das Exception Handling ist ein weiterer Ansatz, Fault Tolerance umzusetzen. Er zielt auf eine Separation of Concerns zwischen normalem Verhalten eines Systemteils und der Behandlung von Fehlerzuständen in diesem Systemteil, wie wir sie in Abschnitt 3.2 vorgestellt haben, ab. Dadurch wird vor allem eine Komplexitätsreduktion in der Anwendungslogik erreicht, da diese weitgehend ohne Rücksicht auf mögliche Fehlerzustände entworfen werden kann[22] [GuRL03], [RLS00].

Auf der Design-Ebene ist dieser Ansatz weit verbreitet und wird von modernen Programmiersprachen unterstützt. Die Granularität der Implementierung sind dabei Methoden oder sogar einzelne Programminstruktionen. Filho et. al. [FiGR03] kritisieren allerdings, dass in komponentenbasierten Systemen diese traditionelle Form des Exception Handling nicht ausreicht: zum einen dienen in diesen Systemen Komponenten (im Gegensatz zu Methoden) als Granulat; zum anderen sind Komponenten i.d.R. „Black Boxes“, d.h. weder eine Überwachung komponenteninterner Abläufe noch deren Erweiterung um für die Umgebung sichtbare Exceptions ist möglich.

In [GuRL03] schlagen sie daher eine Architektur vor, in der jede Komponente unterteilt wird in eine *Normal-Activity-Komponente* (NAK) und eine *Abnormal-Activity-Komponente* (AAK). Erstere implementiert die Anwendungslogik und kann z.B. aus einer COTS-Komponente bestehen. Letztere enthält einen auf die

[21] [Sha01] beschreibt den Einsatz der Simplex-Architektur im Flugkontrollsystem der Boing 777, bei welcher eine über 25 Jahre bewährte Software-Einheit als High-Assurance-Komponente herangezogen wurde.

[22] Robertson et. al. sagen in [RLS00] in Bezug auf Exception Handling:

> „It is often easier to make a program that monitors its performance and recovers from errors than it is to make a program that goes to great lengths to avoid making any errors.“

Komponente abgestimmten Recovery-Mechanismus. Die Komponenten sind untereinander über Konnektoren verbunden und kommunizieren über asynchrone Nachrichten. Die Anbindung der Komponente an die Anwendung ist ebenfalls durch Konnektoren realisiert, die gegenüber dieser als Wrapper fungieren.

Das Nachrichtenmodell besteht im Wesentlichen aus zwei einfachen Nachrichten: einem *Service Request* und einer *Normal Response*. Ein Service Request signalisiert der Komponente, dass die Anwendung eine bestimmte Funktionalität benötigt und beschreibt deren Details. Dieser wird von einem Konnektor an die Normal-Activity-Komponente geleitet, die den Request auszuführen versucht, und – sofern die Ausführung erfolgreich verläuft – eine Normal Response zurück an die Anwendung sendet.

Exceptions sind ebenfalls als Nachrichten modelliert und erweitern das Nachrichtenmodell um *Interface Exceptions*, *Failure Exceptions* und *Internal Exceptions*. Mit einer Interface Exception symbolisiert die NAK der Anwendung, dass der empfangene Service Request ungültig war. Internal Exceptions sendet sie an die AAK, um einen Error während der Behandlung eines Service Requests zu kommunizieren. Diese versucht daraufhin ein Recovery und gibt den Kontrollfluss im Erfolgsfall wieder zurück an die NAK. Schlägt das Recovery fehl, sendet die AAK eine Failure Exception an die Anwendung, womit sie dieser einen Service Failure bei der Bearbeitung eines (gültigen) Requests mitteilt [GuRL03]. Abbildung 6 veranschaulicht diese Abläufe.

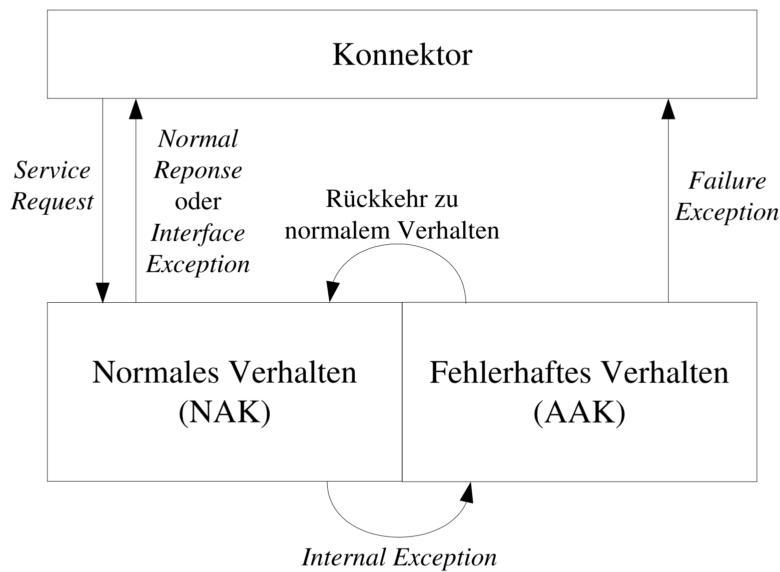

Abbildung 6. Ablauf des Exception Handling (Vereinfacht nach [GuRL03])

Die Behandlung von Exceptions erfolgt – analog zu Exception-Modellen auf der Ebene von Programmiersprachen – in dafür definierten *Exception Handlers*, auf die der Kontrollfluss von der auslösenden Komponente übergeht. Dieser kann entweder einen Recovery-Prozess anstoßen oder seinerseits eine Exception auslösen. Implementiert eine aufrufende Komponente keinen Handler für eine Exception die eine von ihr aufgerufene Komponente generiert, wird diese an die jeweils übergeordnete Komponente propagiert. Findet sich in der gesamten Komponenten-Hierarchie kein geeigneter Handler, wird ein Standard-Handler angestoßen [FiGR03].

Gravierender Nachteil des Exception-Handling-Ansatzes ist allerdings die implizite Annahme, dass alle Ausnahmesituationen zur Entwicklungszeit vorhersehbar und verlässlich definiert sind und, dass entsprechende Exceptions an den jeweiligen Stellen in der NAK generiert werden. (In der beschriebenen Architektur betrifft dies insbesondere die Interface Exceptions und Internal Exceptions, welche die NAK generieren muss.) Dies bedingt sowohl eine Unterstützung des Mechanismus in den verwendeten Komponenten als auch die Verfügbarkeit einer verlässliche Spezifikation aller generierten Exceptions als Teil der Komponentenschnittstelle. Damit steigt die Zahl der Annahmen über die verwendeten Komponenten, was sich negativ auf die Fähigkeit der Architektur auswirkt, möglichst beliebige (COTS-)Komponenten einzubinden.

Ein weiterer Nachteil ist die inkonsequente Separation of Concerns der Kontrolllogik: Die Überwachung findet teilweise innerhalb der Anwendungslogik statt (in Form von Konstrukten, die bestimmte Bedingungen überwachen und ggf. Exceptions generieren) und die Evaluation (hier die Zuordnung von Handlern zu Exceptions) ist nur implizit über (ggf. hierarchisierte) Abbildungen vorhanden. Zudem sind alle Bestandteile der Kontrolllogik in hohem Maße dezentralisiert (z.B. Exception Handler), was eine Wiederverwendung von Implementierungen und gezielte Änderungen des Verhaltens erschwert.

4.8 Versatile Dependability

Ein Aspekt, den die bisher beschriebenen Architekturen weitgehend vernachlässigen, ist der Einfluss, den die Verlässlichkeit eines Systems auf andere wesentliche Systemmerkmale wie z.B. Performanz hat. So führt bspw. die redundante Auslegung eines Systemteils zu einem gewissen Verwaltungsoverhead (etwa für den Transfer von Zuständen zwischen Komponenten), erhöht dafür aber die Verlässlichkeit des Systems.

Dumitras et. al. regen daher in [DuSN05] an, die verschiedenen Ziele einer Architektur als Dimensionen eines Raumes – des *Dependability Design Spaces* – zu betrachten. Als Dimensionen führen sie Fault Tolerance, Ressourcenbedarf und Performanz an. Diese Dimensionen stehen in einer Konfliktbeziehung (*Trade-Off*) zueinander, d.h. eine Verbesserung in einer Dimension wirkt sich u.U. nachteilig auf andere Dimensionen aus. Jedes DAIS befindet sich (zu einem bestimmten Zeitpunkt) an einem bestimmten Punkt innerhalb des Design Spaces, verfügt also über eine definierte Konfiguration der Trade-Offs.

[DuSN05] schlagen vor diesem Hintergrund das Konzept der *Versatile Dependability* vor, das zur Laufzeit des Systems eine Anpassung der Trade-Off-Konfiguration ermöglicht, um auf veränderte Umgebungsbedingungen zu reagieren. Die Anpassung der Konfiguration erfolgt durch die Manipulation von Merkmalen eines Systems, genannt *Regler* (engl. Knobs), die eine bestimmte Dimension beeinflussen.

Um die Komplexität der Systemkonfiguration beherrschbar zu halten, unterscheidet Versatile Dependability zwischen *High-Level Knobs* und *Low-Level Knobs*. High-Level Knobs repräsentieren logische Eigenschaften eines Systems wie bspw. Skalierbarkeit. Diese werden system-intern, d.h. für den (menschlichen) Systemverwalter unsichtbar, auf Low-Level Knobs abgebildet, welche die tatsächlichen Konfigurationsparameter des jeweiligen Systems repräsentieren.

Der Systemverwalter definiert mit Hilfe der High-Level Knobs *Adaptionsregeln* (engl. Adaption Policies), die dem System vorgeben, unter welchen Bedingungen es welche Konfigurationsänderungen vornehmen soll. Die Bedingungen sind anhand von Grenzwerten für beliebige Systemeigenschaften definiert, die der Verwalter festlegt und werden vom System zur Laufzeit überwacht. Stellt es eine Abweichung fest, leitet es die entsprechende Konfigurationsänderung aufgrund des Regelwerks ein.

Die Versatile-Dependability-Architektur ist auf den Einsatz in verteilten Umgebunden in Form einer Middleware-Lösung ausgelegt. Die gesamte Funktionalität ist in einer einzigen Komponente, dem *Replikator*, konzentriert, die als Kontrollschicht zwischen Betriebssystem und Anwendung (bzw. dem Middleware-Framework, auf das die Anwendung aufsetzt,) fungiert. Zur Einbindung in das System kommt die Technik der *Library Interposition* zum Einsatz, d.h. eine Veränderung von Bibliotheksaufrufen des Betriebssystems, welche Aufrufe an die Zwischenschicht vor den eigentlichen Aufrufen an Anwendung und System „einschiebt“. Damit gewinnt der Replikator – in gegenüber der Anwendung vollkommen transparenter Weise – die Möglichkeit, Anwendungsaufrufe zu überwachen und ggf. Adaptionsmaßnahmen zu ergreifen.

Vorrangiger Zweck des Replikators ist es – wie der Name bereits andeutet –, den Zustandstransfer zwischen Anwendungsknoten zu verwalten. Als eine Art „Zusatzdienst“ bietet er die die Veränderung der Systemkonfiguration sowie die dafür notwendige Überwachung der Anwendungslogik, das Auswerten des Regelwerks und die Durchführung von Konfigurationsanpassungen. Eine logische Trennung der einzelnen Aufgaben im Sinne einer Separation of Concerns erfolgt dabei nicht. Auch sieht die Architektur keinerlei Anpassungen an der Kontrolllogik vor, die über eine Veränderung des Regelwerks hinausgehen, sondern ist vielmehr auf einen dauerhaften, unveränderten Einsatz des Replikators ausgelegt.

Diese Struktur führt zu einer sehr hohe Komplexität, die sich in einer einzigen logischen wie physischen Komponente konzentriert, und gleichzeitig zu einer geringen Flexibilität und Leistungsfähigkeit. Damit ist die eigentliche Architektur für praktische Anwendungen eher uninteressant. Die Grundidee des Versatile-Dependability-Konzeptes hingegen, Architekturen auf einem hohen Abstrakti-

onsniveau konfigurierbar zu machen, um so auf Veränderungen der Anforderungen eines Systems flexibel zu reagieren, ist sehr interessant und sollte weiter verfolgt werden. Inbesondere eine Anwendung der Idee auf andere (besser strukturierte) verlässliche Architekturen könnte vielversprechende Ergebnisse produzieren.

5 Architekturen adaptiver Systeme

In diesem Kapitel diskutieren wir Architekturen adaptiver Systeme. Dem sei zunächst vorangestellt, dass die hier vorgenommene Trennung in vorrangig auf Verlässlichkeit und vorrangig auf Adaptivität ausgelegte Architekturen unscharf ist: Einerseits setzen alle in Kapitel 4 beschriebenen Architekturen in mehr oder weniger großem Umfang adaptive Mechanismen ein, um Verlässlichkeit zu gewährleisten. (Beispielhaft seien hier genannt der dynamische Austausch von Implementierungen in Abschnitt 4.5 und die Konfiguration von Systemeigenschaften in Abschnitt 4.8). Andererseits argumentieren z.B. [Koop03], dass zumindest alle auf Selbstheilung ausgelegten adaptiven Systeme letztlich verlässliche Systeme sind. Die Trennung der verschiedenen Architekturansätze in Kategorien nehmen wir daher ausschließlich unter den Gesichtspunkten der Übersichtlichkeit und Verständlichkeit vor, nicht jedoch mit dem Ziel einer eindeutigen Zuordnung.

Grundsätzlich stellen adaptive Systeme im Vergleich zu verlässlichen Systemen ein relatives junges Forschungsgebiet dar. Die Ansätze zur Berücksichtigung der Adaptivität von Software auf der Architekturebene sind daher weitaus weniger konkret und bewegen sich eher auf der Ebene grundsätzlicher Konzepte[23]. Im Folgenden werden wir zunächst einen Überblick über die in der Literatur diskutierten Architekturstile für adaptive Systeme geben und anschließend beispielhaft einige konkrete Konzepte zur Umsetzung von Adaptivität auf der Architekturebene vorstellen. Insbesondere erheben wir damit jedoch keinen Anspruch auf Vollständigkeit der diskutierten Konzepte.

5.1 Architekturstile

Hawthorne und Perry identifizieren in [HaPe05] vier grundlegende Architekturstile adaptiver Systeme: Aspect Peer To Peer, Aggregator Escalator Peer, Chain of Configurators und Configuration Manager. Diese basieren auf der Annahme, dass sich das System in verschiedene, für seine Anpassung relevante Aspekte zerlegen lässt, die sich unabhängig voneinander überwachen und steuern lassen. Im folgenden erläutern wir die einzelnen Stile und diskutieren sie im Hinblick auf die in Kapitel 3 vorgestellten Grundprinzipien für DAIS-Architekturen.

Der *Aspect-Peer-to-Peer*-Stil ordnet jedem der identifizierten Aspekte genau einen *Monitor* zu, der Informationen über den Zustand des Aspektes liefert.

[23] Auch in dieser Hinsicht unterscheidet sich Adaptivität von Adaptierbarkeit, die seit langem ein Ziel von Softwarearchitekturen dartstellt.

Jedem Monitor ist wiederum genau ein *Konfigurator* zugordnet – daher die Bezeichnung Peer-To-Peer –, der die Evaluation und Konfiguration (im Sinne der in Abschnitt 3.2 beschriebenen Phasen) übernimmt. Die einzelnen Monitore und die zugeordneten Konfigurators weisen aufgrund der hohen Spezialisierung nur eine geringe Komplexität auf und bewahren in hohem Maße die Unabhängigkeit der einzelnen Komponenten. Allerdings besteht die Gefahr suboptimaler Entscheidungen der einzelnen Konfiguratoren (z.B. aufgrund der ausschließlich lokalen Optimierung wie in Abschnitt 4.5 diskutiert), da nur ein einzelner Aspekt anstatt eines zusammenhängenden Systems von Aspekten betrachtet wird.

Diese Nachteile adressiert der *Aggregator-Escalator-Peer*-Stil, der jeweils mehrere Monitore und mehrere Konfiguratoren zu umfassenderen Komponenten aggregiert, um so die Entscheidungsgrundlage für Veränderungen zu verbessern und die Effizienz zu steigern. Gleichzeitig bietet dieser Stil die Möglichkeit der Delegation von Entscheidungen an andere Konfiguratoren, so dass sich ggf. eine Hierarchie von Zuständigkeiten ergibt. Die Peer-To-Peer-Zuordnung von Monitoren und Konfiguratoren sich entsprechender Mengen von Aspekten bleibt allerdings erhalten, so dass insbesondere unterstellt wird, dass entweder jeder Aspekt nur an einer Stelle innerhalb des gesamten Systems relevant ist oder zentral an dieser Stelle alle ihn beeinflussenden Faktoren gesteuert werden können. Zumindest bei sehr generischen Aspekten wie der Prozessorlast erscheint diese Annahme fraglich, da diese von nahezu allen Komponenten eines Systems beeinflusst werden und diese wiederum sehr unterschiedliche Strategien zur Änderung eines bestimmten Aspektes erfordern können. Auch eine Betrachtung mehrerer Aspekte in Form von Trade-Offs wie in Abschnitt 4.8 beschrieben bleibt bei diesem Ansatz unberücksichtigt.

Der von [HaPe05] beschriebene *Chain-of-Configurators*-Stil erweitert das Konzept der Delegation von Entscheidungen durch Anwendung des Chain-of-Responsibility-Musters. Dabei werden mehrere Konfiguratoren in einer explizit festgelegten sequentiellen Reihenfolge durchlaufen. Jeder Konfigurator entscheidet sich entweder, selbst auf das vorliegende Szenario zu reagieren (womit er die Kette beendet), oder er eskaliert die Behandlung an den nächsten Configurator. Dieser Stil ermöglicht eine explizite Trennung der verschiedenen Strategien zur Reaktion auf ein Szenario und erleichert das dynamische Hinzufügen neuer Strategien.

Ein gravierender Nachteil, der bei allen drei diskutierten Architektur-Stilen inhärent ist, besteht in der Vermischung der logischen Aspekte Evaluation und Konfiguration (Vgl. Abschnitt 3.2). Dies bedingt eine enge Verknüpfung der Strategien zur Änderung des Systemzustandes mit der Entscheidung darüber, ob eine solche Zustandsänderung in einem bestimmten Szenario sinnvoll ist und welche Strategie sich dafür optimal eignet. Inbesondere besteht dabei die Gefahr, dass ein dynamisches Hinzufügen von Evaluationsstrategien oder eine Änderung des Regelwerks nicht geeignet unterstützt wird und daher hohen Aufwand verursacht.

Auf die Berücksichtigung des SoC-Prinzips zielt der *Configuration-Manager*-Stil ab. Die einzelnen Monitore kommunizieren Änderungen der überwachten

Aspekte an eine zentrale Entscheidungsinstanz, den *Configuration Manager*. Dieser implementiert die Logik der Evaluationsphase, bestimmt also, ob eine Zustandsänderung erforderlich ist und welche Strategie zu deren Ausführung gewählt wird. Anschließend ruft er den entsprechenden Konfigurator auf, um die gewählte Strategie auszuführen.

Eine 1-zu-1-Zuordnung von Aspekten des Systems und Strategien zur Beeinflussung dieser Aspekte ist bei Anwendung des Configuration-Manager-Stils nicht mehr erforderlich, obwohl diese in [HaPe05] nach wie vor implizit unterstellt wird. In Verbindung mit den in Kapitel 3 vorgestellten grundlegenden Architekturprinzipien von DAIS (Komponenten-Basierung, Lose Kopplung, Konnektoren, etc.) stellt dieser Stil daher einen mächtigen Ansatz zur Realisierung adaptiver Systeme dar.

5.2 Contract-based Architecture

CASA (*Contract-based Adaptive Software Architecture*) [MuGl03] ist ein Beispiel für eine adaptive Architektur, die das Zusammenspiel mehrerer, verteilter Applikationen berücksichtigt. Dabei steht insbesondere der Aspekt gemeinsam genutzter (begrenzter) Ressourcen im Vordergrund, deren Verfügbarkeit sich kontinuierlich ändert (etwa CPU-Zeit, Hauptspeicher oder Netzwerkbandbreite).

Zentrales Element in CASA sind *Contracts*, welche den Ressourcenbedarf jeder Anwendung spezifizieren. Diese Spezifikation differenziert verschiedene *Operating Zones*, innerhalb derer die Anwendung jeweils bestimmte Funktionalitäten in einer definierten Qualität bereitstellt und dafür bestimmte Ressourcen benötigt. Die Beschreibung von Ressourcen erfolgt auf zwei logischen Ebenen anhand von *Service-Parametern* und *Resource-Level Requirements*. Diese Unterscheidung ist vergleichbar mit den in Abschnitt 4.8 beschriebenen High-Level Knobs (∼ Service-Parameter) und Low-Level Knobs (∼ Resource-Level Requirements) des Versatile-Dependability-Ansatzes. Erstere verwenden Anwendungen, um mit anderen Anwendungen *Service Levels* (gekennzeichnet durch garantierte Qualitätsmerkmale) auszuhandeln, letztere, um daraus resultierende Anforderungen an Ressourcen zu beschreiben.

Jede Operating Zone in einem Application Contract enthält eine oder mehrere Spezifikationen von für diese Zone zulässigen *Komponenten-Konfigurationen*. Eine solche Komponenten-Konfiguration beschreibt die Komposition der Anwendung aus einer (Teil-)Menge der verfügbaren Komponenten. Das heißt, je nach Operating Zone benutzt die Anwendung unterschiedliche Komponenten, um eine bestimmte Funktionalität zu erbringen. Zusätzlich enthält jede Konfiguration eine Angabe über die zu ihrer Verwendung benötigten Ressourcen. Verändert sich die Verfügbarkeit von Ressourcen über die festgelegten Grenzwerte hinaus (z.B. aufgrund höherer Auslastung oder eines Hardwaredefekts), vereinbart die Anwendung mit den anderen über CASA verwalteten Anwendungen ein anderes Service Level und wechselt in eine andere Operating Zone.

Abbildung 7 zeigt die Bestandteile der CASA-Architektur und deren Zusammenwirken. Jede Anwendung besteht aus der Anwendungslogik und einem *Contract-based Adaption System* (CAS) auf der Meta-Ebene (Vgl. Kapitel 3.3).

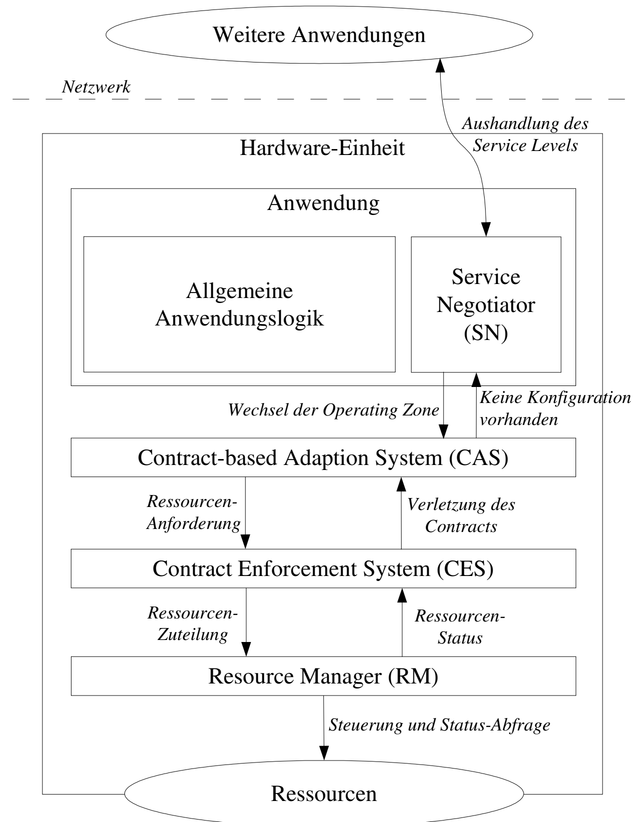

Abbildung 7. CASA-Architektur (In Anlehnung an [MuGl03])

Das CAS vereinigt Konfiguration und Teile der Evaluation: es entscheidet über die im aktuellen Systemkontext angemessene Komponenten-Konfiguration und führt den Wechsel zwischen den Konfigurationen durch. Die Auswahl von Konfigurationen erfolgt auf der Grundlage der im Application Contract für die aktuelle Operating Zone hinterlegten Alternativen.

Das *Contract Enforcement System* (CES) überwacht die Erfüllung der im Application Contract für die momentane Konfiguration hinterlegten Resource-Level Requirements. Es implementiert damit den zum CAS komplementären Teil der Evalutionsphase. Eine Nicht-Erfüllung der Requirements wird an das CAS kommuniziert. Die eigentliche Überwachung der Ressourcen-Eigenschaften und die Zuteilung der verfügbaren Ressourcen erfolgt im *Resource Manager* (RM)[24]. Sowohl CES als auch RM können ebenfalls als verteilte Systeme implementiert sein, um bspw. die Zuteilung solcher Ressourcen zu verwalten, die nicht von einem einzigen Hardwaresystem abhängen (z.B. die Netzwerkauslastung).

Neben diesen Bestandteilen existiert ein *Service Negotiator* (SN), den [MuGl03] als Teil der Anwendungslogik betrachten. Er hat die Aufgabe, mit allen über CASA verbunden Anwendungen ein für alle ausreichendes Service Level auszuhandeln und dieses (anhand der einzelnen Service-Parameter) auf eine Operating Zone abzubilden. Die jeweils gewählte Operating Zone kommuniziert der SN an das CAS, das dessen Umsetzung verantwortet.

Aufgrund der Service-Orientierung und der expliziten Berücksichtigung nicht-lokaler Aspekte ist CASA ein flexibler Ansatz für anwendungsübergreifende Adaptivität. Allerdings weist die Architektur auch eine Reihe von Nachteilen auf. Ein gravierendes Problem ist, dass die Definition der Operating Zones, der Komponenten-Konfigurationen (inklusive deren Ressourcenbedarf) und die Abbildung von Service-Parametern auf Operating Zones in CASA manuell durch die Entwickler der jeweiligen Applikation erfolgt. Dies verursacht in größeren Systemen einen u.U. sehr hohen Aufwand und eine hohe Komplexität, welche die Wartbarkeit der Spezifikationen erschwert. Dies lässt die praktische Verwendbarkeit des Ansatzes in realen Systemen fraglich erscheinen.

Darüberhinaus nimmt die Architektur eine unsaubere Aufteilung der in Abschnitt 3.2 beschriebenen Concerns auf logische Tiers vor: Sowohl die Entscheidung darüber, welche Konfiguration aktuell angemessen ist als auch die Umsetzung dieser Konfiguration sind im CAS implementiert und kaum voneinander abgegrenzt. Dies erschwert Anpassungen der Auswahlstrategie und Änderungen der Konfigurationslogik. Ein weiterer Nachteil ist die fehlende Unterstützung für eine dynamische Optimierung der Konfigurationsauswahl (innerhalb einer Operating Zone). Die Auswahl erfolgt lediglich aufgrund einer manuell spezifizierten Reihenfolge im Application Contract anstatt aufgrund von zur Laufzeit ermittelten Messwerten.

Die Bereitstellung von Logging-Informationen über das Systemverhalten ist in der Architektur ebenfalls nicht vorgesehen, so dass selbst die manuelle Optimierung des Application Contracts kaum objektiv möglich ist. Zudem erschweren

[24] Nicht zu verwechseln mit dem gleichnamigen Systemelement im Bereich von Datenbanksystemen.

das fehlende Logging und die geringe Flexibilität bei der Konfigurationsauswahl den Einsatz von CASA in verlässlichen Systemen, da bspw. fehlerhafte Konfigurationen nicht erkannt oder vermieden werden können. Ingesamt scheint die Architektur damit eher als Denkanstoß für Adaptivität in verteilten Umgebungen, als für die Realisierung realer Systeme geeignet zu sein.

5.3 Distributed Configuration Routing

Das *Distributed Configuration Routing*, kurz DCR, stellt einen weiteren Architekturansatz dar, den Hawthorne und Perry in [HaPe04] beschreiben. Es versucht die Flexibilität der Konfigurationsphase zu maximieren, indem das Wissen über mögliche Strategien zur Durchführung von Zustandsänderungen nicht mehr an einer Stelle zentral, sondern stattdessen nur noch in einem Netzwerk aller Komponenten vorhanden ist. Gleichzeitig sollen Strategien sich aus beliebig vielen Teilstrategien zusammensetzen können, die unabhängig voneinander existieren.

Alle im System verfügbaren Konfigurationsdienste werden als Zustandsübergänge betrachtet. Eine Komponente K_O, die ihre Konfiguration (aufgrund einer vorherigen Evaluation) verändern möchte, erfordert in dieser Betrachungsweise also einen Übergang von einem Startzustand S in einen Zielzustand T. Um den Übergang vorzunehmen, sendet sie einen *Configuration Route Request* (kurz CRR) an alle Konfigurationskomponenten, der Angaben zum aktuellen Zustand und dem gewünschten Zielzustand enthält. Alle Komponenten, die einen Übergang von einem beliebigen Zustand $X^1(\neq S)$ in den gewünschten Zielzustand T vornehmen können, fügen diese Angaben dem CRR hinzu und senden diesen erneut in das Netzwerk[25].

Dieser Vorgang wird rekursiv so lange wiederholt, bis eine Komponente K den Übergang vom Startzustand S in einen aktuelle benötigten Zwischenzustand X^n vornehmen kann. (Die vorzunehmenden Übergänge wären in diesem Beispiel also $S \rightarrow X^n \rightarrow \ldots \rightarrow X^1 \rightarrow T$.) Die betreffende Komponente K sendet eine Bestätigung an die Komponente K_O, welche die Zustandsänderung benötigt. Diese kann daraufhin durch das Senden einer *Configuration Route Notification* die Sequenz von Zustandsänderungen anstoßen. Abbildung 8 zeigt beispielhaft das Entstehen eines CRR.

Existieren mehrere, verschiedene Sequenzen von Zustandsübergängen, um die Komponente K_O von Zustand S nach T zu transformieren, muss exklusiv eine dieser Sequenzen ausgewählt werden. [HaPe04] schlagen hierfür die Verwendung eines Kostenmodells vor: jeder mögliche Zustandsübergang in einem CRR wird mit Meta-Informationen versehen, die seine Kosten (z.B. Bedarf an physischen Ressourcen oder Dauer der Durchführung) repräsentieren. Anhand dieser Informationen kann K_O die kostenminimale Strategie wählen.

Die Umsetzung dieser Architektur erfolgt durch die in Abschnitt 3.2 beschriebene Zweiteilung in Anwendungs- und Kontrolllogik und einer Aufspaltung der

[25] Um Zyklen zu vermeiden gelten für das Hinzufügen von Übergängen zu einem CRR noch zusätzliche Einschränkungen. Beispielsweise darf jede Komponente i.d.R. nur ein einziges Mal an einem CRR beteiligt sein.

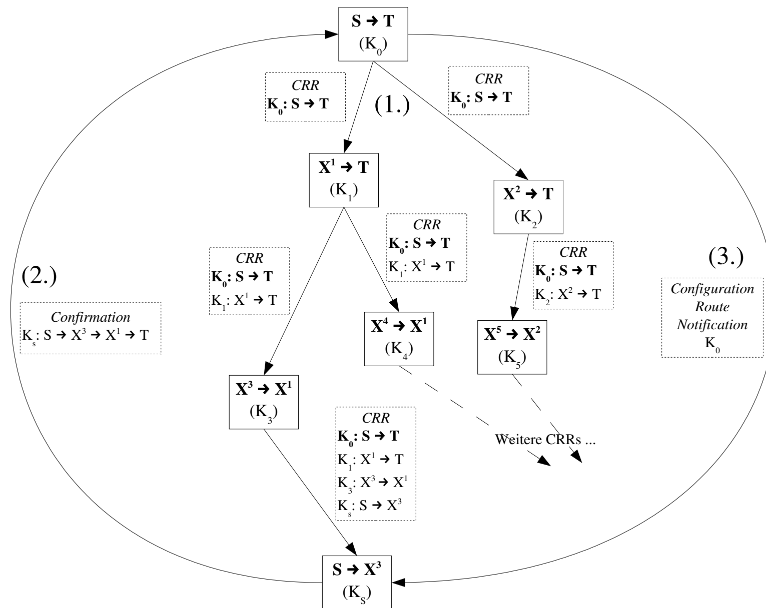

Abbildung 8. Beispielhafter Ablauf des DCR (In Anlehnung an [HaPe04])

Kontrolllogik in Monitore (zur Überwachung von Komponenten- oder Umgebungsmerkmalen) und Adapter. Letztere enthalten die eigentliche Konfigurationslogik, d.h. sie implementieren die beschriebenen (Teil-)Strategien. Die Verbindung der einzelnen Komponenten (d.h. Anwendungskomponenten, Monitore und Adapter) untereinander erfolgt durch Konnektoren, also in loser Kopplung.

Der Ansatz des Distributed Configuration Routing bietet ein hohes Maß an Flexibilität zur Ausführung von Konfigurationen des Systems. Das Hinzufügen oder Verändern von Strategien ist aufgrund der hohen Unabhängigkeit der einzelnen Komponenten auch zur Laufzeit des Systems möglich. Die Aufspaltung von Strategien in Teilstrategien kann zudem eine Komplexitätsreduktion der Konfigurationsimplementierungen bewirken und fördert die Wiederverwendung von Implementierungsteilen.

Nachteilig an dieser Architektur ist hingegen die Vernachlässigung der Evaluationsphase. Diese wird – entgegen der Separation of Concerns – nicht als explizites Architekturelement betrachtet, sondern implizit in Monitoren, Adaptern und Konnektoren verteilt. Eine Kombination des DCR mit dem in Abschnitt 5.1 erläuterten Configuration-Manager-Stil könnte allerdings eine hochgradig flexible und mächtige Architektur für adaptive Systeme ergeben.

5.4 Dynamic Dispatch

Eine weitere Variante der Design Diversity – angesiedelt auf der Ebene von Methoden – schlagen Laddaga et. al. [LaRS01] vor. Die zugrundeliegende Idee ist,

dass zur Bereitstellung einer bestimmen Funktionalität mehrere Implementierungen einer bestimmten Methode existieren und erst zum Zeitpunkt des Methodenaufrufs durch den Dispatcher entschieden wird, welche Implementierung zu verwenden ist.

Eine mögliche Umsetzung dieses Ansatz stellt das in [LaRS01] diskutierte *Probabilistic Dispatch* dar. Hierbei wird angenommen, das jede Methodenimplementierung mit einer gewissen Wahrscheinlichkeit erfolgreich terminiert. Zur Ermittlung der Wahrscheinlichkeit ist zu jeder Methodenimplementierung ein Meta-Datum in Form einer Kennzahl hinterlegt, welches vom Dispatcher nach jeder Verwendung der Implementierung gepflegt wird. (Beispielsweise inkrementiert der Dispatcher nach einer erfolgreichen Ausführung der Methodenimplementierung die Kennzahl und dekrementiert sie bei einem Error.) Weisen mehrere Implementierungen zum Zeitpunkt eines Aufrufs die gleiche Wahrscheinlichkeit auf, wählt der Dispatcher in einem vorher definierten Verfahren eine der Betreffenden aus. Die Auswahl einer Implementierung kann z.B. zufällig erfolgen oder eine über alle Implementierungen definierte Ordnung[26] herangezogen werden.

Implizite Grundannahme des Probabilistic Dispatch ist, dass im System eine Methodenimplementierung verfügbar ist, welche in allen Szenarien optimal funktioniert. Denn es erfolgt zwar eine Auswahl bzw. Bewertung von Implementierungen, diese berücksichtigt jedoch nicht den aktuellen Systemkontext (etwa die Prozessorlast), sondern ausschließlich die *durchschnittliche* Erfolgswahrscheinlichkeit einer Implementierung. Zudem berücksichtigt das Verfahren als einziges Qualitätsmaß die Wahrscheinlichkeit einer erfolgreichen Terminierung des Programmablaufs, nicht hingegen andere Merkmale wie die Ausführungsdauer oder der Ressourcenbedarf. Daher erscheint die praktische Eignung dieser Dynamic-Dispatch-Variante fraglich.

Interessanter ist eine zweite Variante, genannt *Expected Utility Dispatch*, welche Laddaga et. al. in [LaRS01] vorschlagen, jedoch nicht näher untersuchen. Sie verwendet als Grundlage der Implementierungsauswahl den von einer Implementierung *erwarteten Nutzen* (engl. expected Utility) in einem bestimmten Szenario. Dieser kann z.B. in der Relation von Genauigkeit eines Ergebniswertes und der für dessen Berechnung benötigten Zeit bestehen; die ausgewählte Methode ist also nicht notwendigerweise identisch mit der Methode mit der geringsten Fehlerwahrscheinlichkeit. Kritisch muss allerdings auch hierbei betrachtet werden, dass jede Methode nur über *einen* Expected-Utility-Wert verfügt, so dass ebenfalls keine Berücksichtigung des Systemkontextes über Durchschnittswerte hinaus erfolgt.

Grundsätzlich ist bei allen Varianten des Dynamic-Dispatch-Ansatzes fraglich, ob die Wahl von Methoden als Granulat der Anpassung ausreichende Flexibilität bietet. Die Distribution und Verwaltung einzelner Methodenimplementierungen (z.B. als COTS-Komponenten) ist deutlich aufwendiger als die umfänglicher Komponenten. Auch die dynamische Einbindung neuer

[26] [LaRS01] schlagen als Ordnungsmaß die Reihenfolge der Methodenimplementierungen im Quellcode vor.

Implementierungen oder Veränderungen an bestehenden Implementierungen gestaltet sich bei Methoden schwierig, da diese i.d.R. keine eigenständigen Build-Einheiten darstellen, sondern z.B. in Klassen oder Bibliotheken eingebettet sind. Somit bedingt der Austausch einer einzelnen Methodenimplementierung in den meisten Fällen auch den Austausch anderer Systembestandteile, die mit der Funktionalität der betreffenden Methode nicht direkt in Zusammenhang stehen. Die praktische Relevanz dieses Ansatzes ist daher insgesamt fraglich.

6 Zusammenfassung und Ausblick

Die Merkmale Verlässlichkeit und Adaptivität haben in heutigen Informationssystemen eine hohe Bedeutung erlangt. Da eine Verifikation realer Systeme i.d.R. nicht möglich ist, müssen andere Mechanismen zur Entwicklung verlässlicher, adaptiver Informationssysteme (DAIS) eingesetzt werden. Neben entsprechenden Vorgehensmodellen und Werkzeugen spielt dabei insbesondere die Architektur eines Systems eine wichtige Rolle, da sie eine hohe logische Abstraktionsebene bietet und die Entscheidungen der Entwurfs- und Implementierungsphase wesentlich beeinflusst.

Wir haben eine Reihe von Architekturprinzipien verlässlicher, adaptiver Systeme identifiziert, die sich in nahezu allen DAIS wiederfinden. Dazu zählen die komponentenbasierte Architektur, Architectural Reflection und die redundante Auslegung von Systemteilen. Letztere ist vor allem in Form der Design Diversity von Bedeutung, welche die Verwendung unterschiedlich implementierter, funktionsgleicher Komponenten verfolgt. Darüberhinaus ist eine klare Separation of Concerns zwischen der Anwendungslogik und einer Kontrolllogik auf der Meta-Ebene wichtig, um Abhängigkeiten zwischen Systemteilen zu minimieren und so inbesondere die Verwendung unzuverlässiger COTS-Komponenten zum Bau verlässlicher Systeme zu ermöglichen. Eine Separation of Concerns innerhalb der Kontrolllogik in drei Phasen (Überwachung, Evaluation und Konfiguration) verringert die Komplexität und erhöht die Flexibilität der Architektur.

Diese Architekturprinzipien führen jedoch auch zu einer Reihe von Problemen bei der Umsetzung. Neben der Analyse von Ergebniswerten redundanter Komponenten und der notwendigen Isolation von Komponenten zur Verhinderung von Seiteneffekten ist vor allem der Transfer von Zuständen zwischen Komponenten problematisch. Auch muss beachtet werden, dass Design Diversiy nur unter bestimmten Voraussetzungen die Verlässlichkeit eines Systems erhöht. Des Weiteren existiert ein Trade-Off zwischen dem Grad der Diversifizierung funktionsgleicher Implementierungen und ihrer Integrierbarkeit in eine generische Schnittstelle.

Für verlässliche Systeme existieren in der Literatur zahlreiche Architekturkonzepte, wobei keine Architektur für den Einsatz in allen Kontexten optimal geeignet ist. Vielmehr trifft jede Architektur Annahmen, die für den jeweiligen Anwendungsbereich überprüft werden müssen, um die im Kontext beste Architekurvariante zu ermitteln. Die Konzepte reichen dabei von einfachen Ansätzen

wie N-Version-Software und Recovery Blocks bis zu sehr mächtigen Konzepten wie der Component Redundancy.

Im Bereich adaptiver Systeme existieren bisher hingegen keine umfassenden Architekturen, sondern nur eine ganze Reihe unterschiedlicher Ansätze. Dabei ist die Abgrenzung von verlässlichen und adaptiven Systemen eher theoretischer Natur, da beide Bereiche eng verwoben sind und einander in weiten Teilen bedingen. Wir haben beispielhaft die Ansätze der Contract-based Architecture, des Dynamic Configuration Routing und dynamischer Methodenaufrufe vorgestellt, welche das breite Spektrum adaptiver Systemarchitekturen demonstrieren.

Insgesamt lässt sich in DAIS-Architekturen zumindest bezogen auf den Aspekt der Verlässlichkeit eine Konvergenz gegen eine Grundmenge von Ideen erkennen, die sich im Wesentlichen in den diskutierten Prinzipien widerspiegeln. Neuere Architekturvorschläge unterscheiden sich i.d.R. nur geringfügig und in sehr speziellen Aspekten von bereits existierenden Ansätzen. Vor diesem Hintergrund ist eine Konsolidierung der verschiedenen Konzepte anzustreben, um in absehbarer Zeit umfassende Architekturlösungen für reale Informationssysteme zu schaffen.

Neben Architekturen finden sich in der Literatur zahlreiche sogenannte „Frameworks“ für die Erstellung verlässlicher und/oder adaptiver Systeme. Bei näherer Betrachtung zeigt sich jedoch, dass diese lediglich Architekturen darstellen, da sie keine generischen, auf Wiederverwendung ausgelegten Kernimplementierungen beinhalten. Frameworks im engeren Sinne existieren hingegen derzeit weder für adaptive noch für verlässliche Systeme. Allerdings gibt zahlreiche Projekte, welche die Schaffung von DAIS-Frameworks zum Ziel haben, so dass in absehbarer Zeit erste Ergebnisse zu erwarten sind.

Literatur

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr. “Basic Concepts and Taxonomy of Dependable and Secure Computing”, IEEE Transactions on Dependable and Secure Computing, vol. 01, no. 1, pp. 11–33, January–March, 2004

[AnCh04] “P. Andras, B. G. Charlton, “Self-aware Software – Will It Become a Reality?”, in O. Babaoglu et al. (Eds.), SELF-STAR 2004, LNCS 3460, Springer-Verlag, pp. 229–259, 2005

[Aviz95] A. Avizienis, “The Methodology of N-Version Programming”, Chapter 2 of Software Fault Tolerance, M. R. Lyu (ed.), Wiley, pp. 23–46, 1995

[AvLR01] A. Avizienis, J.-C. Laprie, B. Randell, “Fundamental Concepts of Computer System Dependability”, in IARP/IEEE-RAS Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments, Seoul, Korea, 2001

[BaPl01] D. Balek, F. Plasil, “Software Connectors and Their Role in Component Deployment”, in Proceedings of 3rd International Conference on Distributed Applications and Interoperable Systems (DAIS’01), Krakow, Kluwer, 2001

[BDH+98] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal, C. Szyperski, “What characterizes a (software) component?”, Software Concepts and Tools, vol. 19, pp. 49–56, 1998

[BrPe02] M. Brandozzi, D. Perry, "Architectural Prescriptions for Dependable Systems", ICSE, WADS 2002, 2002

[DiMu03] A. Diaconescu, J. Murphy, "A Framework for Using Component Redundancy for Self-Optimising and Self-Healing Component Based Systems", WADS Workshop, ICSE'03, Oregon, USA, 2003

[DuSN05] T. Dumitras, D. Srivastava, P. Narasimhan, "Architecting and Implementing Versatile Dependability", in R. de Lemos et. al. (Eds.), Architecting Dependdable Systems III, LNCS 3549, Springer-Verlag, pp. 212–231, 2005

[FeRu98] L. L. Ferreira, C. M. F. Rubira, "Reflective Design Patterns to Implement Fault Tolerance", in Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java, Int. Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, Canada, 1998

[FiGR03] F. C. Filho, P. A. de C. Guerra, C. M. F. Rubira, "An Architectural-Level Exception-Handling System for Component-Based Applications", in R. de Lemos et al. (Eds.), LADC 2003, LNCS 2847, Springer-Verlag, pp. 321–340, 2003

[GaRu01] A. F. Garcia, C. M. F. Rubira, "An Architectural-Based Reflective Approach to Incorporating Exception Handling into Dependable Software", in A. Romanovsky et al. (Eds.), Exception Handling, LNCS 2022, Springer-Verlag, pp. 189–206, 2001

[GaCS03] D. Garlan, S. Cheng, B. Schmerl, "Increasing System Dependability through Architecture-based Self-Repair", in A. Romanovsky et al. (Eds.), Architecting Dependable Systems, LNCS 2677, Springer-Verlag, pp. 61–89, 2003

[GRRL03] P. A. de C. Guerra, C. M. F. Rubira, A. Romanovsky, R. de Lemos, "Integrating COTS Software Components into Dependable Software Architectures", in Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC-03), IEEE Computer Society Press, pp. 139–142, 2003

[GuRL03] P. A. de C. Guerra, C. M. F. Rubira, R. de Lemos, "A Fault-Tolerant Architecture for Component-Based Software Systems", in R. de Lemos, C. Gacek, A. Romanovsky (Eds.), Architecting Dependable Systems, LNCS 2677, Springer-Verlag, pp. 129–149, 2003

[HaFS04] S. Hallsteinsen, J. Floch, E. Stav, "A Middleware Centric Approach to Building Self-Adapting Systems", in T. Gschwind, C. Mascolo (Eds.), SEM 2004, LNCS 3437, Springer-Verlag, pp. 107–122, 2005

[HaPe04] M. J. Hawthorne, D. E. Perry, "Exploiting Architectural Prescriptions for Self-Managing, Self-Adaptive Systems: A Position Paper", ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04), ACM SIGSOFT Foundations of Software Engineering (FSE 2004), Newport Beach, California, USA, 2004

[HaPe05] M. J. Hawthorne, D. E. Perry, "Architectural Styles for Adaptable Self-Healing Dependable Systems", 2005

[Koop03] P. Koopman, "Elements of the Self-Healing System Problem Space", Workshop on Software Architectures for Dependable Systems (WADS2003), ICSE'03 International Conference on Software Engineering, Portland, Oregon, 2003

[Ladd00] R. Laddaga "Active Software", in Proceeding of IWSAS'00, pp. 11–26, 2000

[LaRS01] R. Laddaga, P. Robertson, H. Shrobe, "Probabilistic Dispatch, Dynamic Domain Architecture, and Self-adaptive Software", in R. Laddaga et al. (Eds.), Self-Adaptive Software, Second International Workshop, IWSAS 2001, LNCS 2614, pp. 227–237, Springer-Verlag, 2003

[LiGR02] F. J. C. Lima Filho, P. A. de C. Guerra, C. M. F. Rubira, "FaTC2: An Object-Oriented Framework for Developing Fault-Tolerant Component-Based Systems", In Proc. ICSE 2003 Workshop on Architecting Dependable Systems, pp. 13–18, Portland, USA, 2002

[LiRi02] C. Liu, D. J. Richardson, "RAIC: Architecting Dependable Systems through Redundancy and Just-In-Time Testing", ICSE, Workshop on Architecting Dependable Systems (WADS), Orlando, Florida, USA, 2002

[MuGl03] A. Mukhija, M. Glinz, "CASA - A Contract-based Adaptive Software Architecture Framework", Technical Report, `http://www.ifi.unizh.ch/groups/req/ftp/papers/CASA2003.pdf`, IFI, University of Zurich, 2003

[MuKi04] S. Mustafiz, Jörg Kienzle, "A Survey of Software Development Approaches Addressing Dependability", in N. Guelfi et al. (Eds.), FIDJI 2004, LNCS 3409, Springer-Verlag, pp. 78–90, 2005

[Poet03] Poetzsch-Hefter, A., Skriptum zur Vorlesung "Entwicklung von Softwaresystemen II", SS 2003, TU Kaiserslautern, 2003

[PoSR00] P. Popov, L. Strigini, A. Romanovsky, "Diversity for Off-The-Shelf Components", Int. Conf. on Dependable Systems and Networks, pp. B60-B61, New York, USA, 2000

[RaMe01] M. Rakic, N. Medvidovic."Increasing the Confidence in Off-The-Shelf Components: A Software Connector- Based Approac", Proceedings of SSR'01, Symposium on Software Reusability, ACM/SIGSOFT Software Engineering Notes, 26, pp. 11-18., 2001

[RLS00] P. Robertson, R. Laddaga, H. E. Shrobe, "Introduction: The First International Workshop on Self-Adaptive Software", in Proceedings of IWSAS'00, pp. 1–10, 2000

[Sha01] L. Sha, "Using Simplicity to Control Complexity", IEEE Software, pp. 20–28, July/August, 2001

[TSCS00] F. Tisato, A. Savigni, W. Cazzola, A. Sosio, "Architectural Reflection – Realising Software Architectures via Reflective Activities", in W. Emmerich, S. Tai (Eds.), EDO 2000, LNCS 1999, Springer-Verlag, pp. 102–115, 2001