M.C. Jaeger, M. Werner, G. Mühl, H.-U. Heiß, U. Laude und C. Ruge

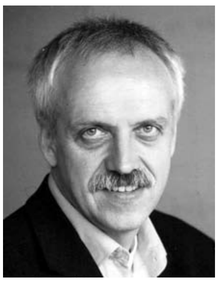
Autonomie in IT-Systemen

Ein Konzeptionelles Modell

Michael C. Jaeger hat Technische Informatik an der TU Berlin studiert und promovierte dort im Anschluss zum Dr.-Ing. Während dessen arbeitete als wissenschaftlicher Mitarbeiter in den Bereichen SOA, Modellierung und nicht-funktionale Eigenschaften. Zurzeit ist er bei Siemens Corporate Research and Technology in München im Bereich Software Architecture (CT SE 2) als Associate Research Scientist tätig.

Matthias Werner hat Elektrotechnik, Regelungstechnik und Technische Informatik an der Humboldt Universität zu Berlin studiert, von der er auch die Promotion erhielt. Danach arbeitete er unter anderem an der University of Texas at Austin, an der TU Berlin, beim Microsoft Research Lab in Cambridge, und bei Daimler Research. Sein Forschungsinteresse liegt im Bereich der nicht-funktionalen Eigenschaften, besonders im Bereich der Echtzeitverarbeitung und Verlässlichkeit. Zurzeit ist er Professor am Lehrstuhl für Betriebssysteme an der Universität Chemnitz.

Gero Mühl erhielt 1998 das Diplom in Informatik und das Diplom in Elektrotechnik von der FernUniversität in Hagen. Im Anschluss an sein Studium wechselte er an die Technische Universität Darmstadt und promovierte dort 2002 zum Dr.-Ing. Seit seiner Promotion ist er Wissenschaftlicher Assistent an der Technischen Universität Berlin und habilitierte dort im Jahr 2007. Sein aktueller Forschungsschwerpunkt ist Selbstorganisation in verteilten Systemen.



Hans-Ulrich Heiß studierte Informatik an der Universität Karlsruhe. Nach der Promotion in Karlsruhe war er 1988-89 als PostDoc am IBM Watson Research Center in Yorktown Heights und 1991 Gastprofessor an der Universität Helsinki. Seit 2001 ist er Professor für „Kommunikations- und Betriebssysteme“ an der TU Berlin, nachdem er zuvor Professuren an den Universitäten Ilmenau und Paderborn wahrgenommen hatte.

Uwe Laude studierte an der Freien Universität Berlin Biologie. Danach war er als wissenschaftlicher Mitarbeiter beim Institut für Gewässerökologie und Binnenfischerei (IGB), beim German Resource Center for Genome Research (RZPD) und bei der Proteinstrukturfabrik (PSF) tätig. In dieser Zeit promovierte er auch zum Dr. rer. nat. Es folgte ein Aufbaustudium an der FH Bund im Fach Verwaltungsinformatik. Seit 2005 ist Dr. Laude Mitarbeiter im Bundesamt für Sicherheit in der Informationstechnik (BSI), Referat „Hochverfügbarkeit in kritischen Geschäftsprozessen“.

Christian Ruge studierte Physik an der Universität Kiel, an der er im Anschluss im Fach Theoretische Physik zum Dr. rer. nat. promovierte. Danach arbeitete er als wissenschaftlicher Mitarbeiter am Hochschulrechenzentrum der Uni Marburg und darauf im Rechenzentrum des Forschungszentrums der ABB Schweiz. Seit dem Jahr 2000 ist Dr. Ruge Mitarbeiter im Bundesamt für Sicherheit in der Informationstechnik (BSI), Referat „Hochverfügbarkeit in kritischen Geschäftsprozessen“.

ZUSAMMENFASSUNG

Die Erforschung und Anwendung von autonomen Systemen ist momentan in der Informatik ein Themengebiet von wachsendem Interesse. Die Aussicht, mit autonomen Verfahren komplexe Systeme handhabbar zu machen und Kosteneinsparungen bei deren Betrieb zu erzielen, hat bereits die Softwareindustrie auf dieses Thema gelenkt und zu neuartigen Produkten geführt. Andererseits darf die Verlässlichkeit eines Systems nicht aufgrund eines autonomen Verfahrens herabgesetzt werden. Dieses Spannungsfeld ist ein Fokus verschiedener Forschungsbemühungen, um autonome Systeme alltagstauglich zu machen.

Beim Gebiet der autonomen Systeme handelt es sich um ein junges Themengebiet, welches noch nicht durch allgemein akzeptierte Definitionen geprägt ist. Dadurch entsteht der Bedarf einer terminologischen Basis, die sich momentan in der Phase der Etablierung befindet. Diese Arbeit beteiligt sich an diesem Prozess und schlägt ein konzeptionelles Modell vor. Dieses Modell benennt die grundlegenden Termini und zeigt deren Zusammenhänge auf. Es beschreibt eine Interpretation der rele-

PIK 31 (2008) 1
10.1515/piko.2008.002

vanten Begriffe und leitet daraus Relationen ab. Auf diese Weise fördert es das gemeinsame Verständnis und erleichtert die Kommunikation bezüglich spezifischer Fragestellungen innerhalb dieses Gebietes.

1 EINFÜHRUNG

In der Informatik wächst sowohl in der Forschung als auch bereits in der praktischen Anwendung das Interesse an autonomen Systemen. Als prominentes Beispiel ist hier die *Autonomic Computing Initiative* von IBM [10] zu nennen, die sowohl Forschung als auch Industrie beschäftigt. Es handelt sich bei den autonomen Systemen um ein noch recht junges Gebiet, in dem es häufig noch keine fest etablierten Vorgehensweisen und wenig exakt definierte Begrifflichkeiten gibt.

Typische Projekte auf dem Gebiet der autonomen Systeme vereinigen Auftraggeber aus der Wirtschaft, Forscher und Mitarbeiter mit Hintergrund aus anderen Disziplinen in ein gemeinsames Vorhaben. Aus den Erfahrungen, die wir in solchen Konstellationen gewinnen konnten, zeigt sich, dass selbst beim Gebrauch von fundamentalen Begriffen bereits viele Missverständnisse entstehen können. Schon bei häufig verwendeten Begriffen wie Service-Oriented Architecture (SOA) kann in einer Diskussion schnell Unklarheit darüber aufkommen, inwiefern die SOA die Architektur eines Systems darstellt – oder nur eine Abstraktion derselben. Die Benutzung vieler Termini ist zudem – wie stets in einem noch jungen Gebiet – durch den allgemeinen Sprachgebrauch beeinflusst, so dass es für viele Forschungsvorhaben einer Klarstellung bedarf, die grundlegende Begriffe in einen einheitlichen Rahmen zu stellt. Werden autonome Systeme behandelt, können schnell Missverständnisse über die Begriffe Adaptivität, Autonomie und Automatik aufkommen. Es existieren zwar Arbeiten, die den Zusammenhang dieser Termini bereits behandeln (z.B. Herrmann et al. [8]). Allerdings streben diese Arbeiten nicht die Schaffung eines umfassenden konzeptionellen Rahmens für die Autonomie in technischen Systemen der Informatik an.

Aus diesen Umständen heraus entsteht die Motivation ein konzeptionelles Modell zu entwerfen, das die grundlegenden Termini im Bereich autonomer Systeme in einen gemeinsamen Zusammenhang bringt. Dieses Modell soll für zukünftige Projektvorhaben eine Referenz bieten und somit Kommunikation und Forschungsarbeiten erleichtern. Wir sind uns bewusst, dass dieses Modell zu diesem Zeitpunkt nicht etwas Feststehendes sein kann. Jedoch hoffen wir, dass wir mit diesem Beitrag helfen, zu einer gemeinsamen Fachsprache auf dem Gebiet der autonomen Systeme zu finden.

2 GRUNDLEGENDE BEGRIFFE

Für die Betrachtung von autonomen Systemen ist es zunächst notwendig, das Verständnis über grundlegende Begriffe wie System oder Architektur zu schaffen. Es existieren bereits mehrere Definitionen und Vorschläge, die ein System charakterisieren. Einige haben bereits in Standardisierungen Einzug gefunden – zum Beispiel in Standards der DIN oder der ISO [3, 11]. Diese verschiedenen Auffassungen zusammenfassend schlägt Werner [29] folgende Definition vor:

> **Definition 1 (System)** *Ein* System *ist eine Menge von Elementen, die in einer bestimmten Umgebung oder in einem bestimmten Kontext eine Einheit bilden oder als Einheit aufgefasst werden, wobei die Elemente miteinander in Beziehung stehen und interagieren können.*
> *Diese Menge kann mit ihrer Umgebung interagieren und sensibel gegenüber dem Verstreichen von Zeit sein. Die Elemente der Menge können wiederum als System aufgefasst werden.*

Diese Definition macht deutlich, dass ein System das Ergebnis einer Interpretation ist und nicht per se existiert. Weiterhin ist von Bedeutung, dass ein System eine Komposition von Systemen sein kann. Durch Komposition von Systemen können vielschichtige Systeme erschaffen werden, deren Handhabbarkeit durch Unterteilungen gewährleistet wird.

In der Informatik finden sich viele Beispiele für Systeme. Es existieren Betriebssysteme, die Bestandteil eines Computersystems sind und deren Ressourcen verwalten. Ein Betriebssystem beinhaltet seinerseits ein Dateisystem, das die Organisation von Daten auf einem Datenträger realisiert. Verteilte Systeme beispielsweise haben die Charakteristik, dass ihre Elemente nach bestimmten Gesichtspunkten aufgeteilt sind. Ein Beispiel hierfür sind so genannte Client/Serversysteme. Ein Client/Serversystem stellt eine Aufteilung der Systembestandteile in einen Client- und einen Server-Anteil in Form einer allgemein gültigen Regelung dar. An dieser Stelle spricht man auch von einer Architektur. In Anlehnung an das eigenständige Fach Architektur hat sich der Begriff der Architektur in der Informatik als Bezeichnung für den Aufbau und die Struktur einer Hard- oder Software etabliert. Des Weiteren stellt die systematische Anfertigung einer strukturierten Hard- oder Software ein Vorgehen dar, das der Architektur als Fachdisziplin ähnelt. Das ISO Reference Model for Open Distributed Processing (RM-ODP) definiert den Begriff der Architektur in der Informatik folgendermaßen [11]:

> **Definition 2 (Architektur)** *Eine* Architektur *definiert Regeln bezüglich der Struktur eines Systems und den Zusammenhängen zwischen seinen Elementen.*

Diese Definition ist für viele Beispiele in der Informatik nachzuvollziehen: Eine Kernelarchitektur bestimmt für Betriebssysteme, dass entweder nur notwendige Bestandteile zur Speicherverwaltung und Behandlung von Interrupts Bestandteile sein sollen (Mikrokernelarchitektur) oder aus Gründen der Effizienz alle zwingend notwendigen Basisbestandteile eines Betriebssystems Teil des Kernels sind (Monolithische Kernelarchitekur, [25]).

Im Bereich der Softwarearchitekturen sind schichtartige Modelle bekannt: Beispielsweise definiert das Schichtenmodell des Open System Interconnect (OSI) Standards der ISO [12] eine Architektur für die Kommunikation zwischen Systemen. Aus dem Bereich der verteilten Systeme wurde bereits das Client/Serversystem genannt, welches der Regel im Sinne einer Architektur folgt, in einen Client- und Server-Bestandteil aufgeteilt zu sein. Diese Beispiele machen deutlich, dass eine Menge von Regeln die Eigenschaften einer bestimmten Architektur festlegt. Allerdings wird hierbei auch deutlich, dass die Architektur eines Systems nicht vollständig von „der“ Client/Serverarchitektur beschrieben wird. Vielmehr stellt die Client/Serverarchitektur eine Abstraktion einer Architektur dar, die sich auf einen wesentlichen Umstand bezieht. Daher muss der Begriff eines *Architekturmusters* eingeführt werden, der eine Abstraktion einer konkreten Architektur im Sinne des RM-ODPs darstellt [11]:

Definition 3 (Architekturmuster) *Ein* Architekturmuster *abstrahiert von einer konkreten Architektur, um einen relevanten Bestandteil einer Architektur wiederzugeben.*

Die Architektur als Disziplin bezieht sich auch auf das Vorgehen, um ein Bauwerk zu erschaffen. Auch hierbei finden Regeln bzw. Konventionen verwendung, die sich jedoch nicht auf die Charakteristika eines Systems nach Definition 2 beziehen. Sie beziehen sich auf die Beschreibung eines Vorgangs, um ein System zu realisieren. Um diese Facette des Begriffs der Architektur von der zuvor gegebenen Definition abzugrenzen, soll hierfür der Begriff der Entwicklungsmethode mit folgender Definition eingeführt werden:

Definition 4 (Entwicklungsmethode) *Eine* Entwicklungsmethode *beschreibt einen geregelten Vorgang, dessen Produkt ein System mit einer bestimmten Architektur ist.*

Anhand der obigen vier Definitionen lassen sich viele Begriffe der Informatik einordnen: Systeme, die komponiert oder elementar sein können, Architekturen, die Systeme charakterisieren und Entwicklungsmethoden, mittels derer sich Systeme erschaffen lassen. Eine Entwicklungsmethode strukturiert das Vorgehen einer Entwicklung, die ein System zum Produkt hat. An dieser Stelle existiert auch der Begriff eines Entwicklungsprozesses. Hierfür muss unterschieden werden, ob dieser Begriff im Sinne einer Entwicklungsmethode verwendet wird, also ob damit ein Vorgehensmuster beschrieben wird, oder ob ein real durchgeführter Prozess bezeichnet wird, der die Realisierung eines konkreten Produkts verfolgt. Für das Modell wurden daher die Begriffe Entwicklungsmethode und Entwicklung gewählt, die diese Unterscheidung in eindeutiger Weise aufzeigen. Damit sind die grundsätzlichen Basiskonzepte geklärt und können in Beziehung gesetzt werden (vgl. Abb. 1).

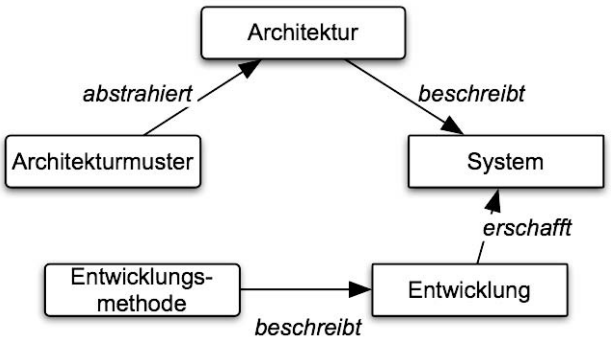


Abb. 1 Zusammenhang Architektur, System und Entwicklungsmethode.

3 EIGENSCHAFTEN VON SYSTEMEN

Allgemein stellt ein System seinem Benutzer eine (oder auch mehrere Funktionen) zur Verfügung. Diese kann (können) sich zum einen auf den Zugriff auf Information als auch auf die Durchführung einer Handlung beziehen. Ein Benutzer kann eine Person sein, die das System über eine Benutzerschnittstelle bedient. Ein Benutzer kann aber auch ein anderes (Software-)System sein, wie es im Fall von Web Services angestrebt wird. Darüber hinaus sind mit der Benutzung Aspekte verbunden, wie z.B. die Verarbeitungsgeschwindigkeit eines Systems, welche Organisation den Betrieb eines Systems verantwortet, mit welcher Zuverlässigkeit ein System seine Funktionen erbringt oder wie viele Ressourcen zur Durchführung einer Funktion benötigt werden. Diese und weitere Aspekte werden in der Informatik als *nicht-funktionale* Eigenschaften bezeichnet.

Eine Untergruppe der nicht-funktionalen Eigenschaften bildet die Dienstgüte (Quality-of-Service, QoS). Als Dienstgüte werden nicht-funktionale Eigenschaften bezeichnet, die von außerhalb des Systems beobachtbar sind und sich mittels einer formalen bzw. konkreten Methode beschreiben lassen (vgl. ISO QoS Framework 13]). Der ISO Standard 9004 definiert hierzu in kurzer Form [26]:

Definition 5 (Dienstgüte) *Die* Dienstgüte *beschreibt die Eigenschaften, die sich auf die Fähigkeit eines Systems beziehen, konkrete oder implizierte Anforderungen beim Erbringen einer Funktion zu erfüllen.*

Das UML QoS Profile der Object Management Group (OMG) [20] ist etwas konkreter und definiert Dienstgüte als quantifizierbare Eigenschaft eines Systems. Zusätzlich impliziert die Definition der ISO auch, dass die Dienstgüte im Kontext der Interaktion eines Systems mit Dritten von Bedeutung ist. Ein System an sich, ebenso wie eine Softwarekomponente oder eine Hardware erbringt keine bestimmte Dienstgüte, wenn diese nicht in einem Benutzungskontext steht. Eine einzelne Größe, die eine bestimmte Dienstgüte ausdrückt, wird Dienstgütemerkmal (*QoS characteristic,* [20]) genannt. In der Informatik existiert eine Vielzahl von Dienstgütemerkmalen für unterschiedliche Anwendungen. Im Bereich der Telekommunikationssysteme sind zum Beispiel die Verzögerung oder die operationale Verfügbarkeit besonders wichtig. Bei Internetanwendungen wird meist auf Datendurchsatz und Antwortzeiten besonderes Gewicht gelegt. Im Bereich der Verteilten Systeme, Middleware und auch IT-Systeme in Unternehmen gewinnt der Begriff der Verlässlichkeit an Bedeutung: Der Einsatz von IT-Systemen im wirtschaftlichen Umfeld bedingt die Entwicklung und den Einsatz verlässlicher Systeme, um die Wirtschaftlichkeit einer Unternehmung zu garantieren.

Aufgrund der durch den allgemeinen Sprachgebrauch gegebenen Bedeutung ist der Begriff *Verlässlichkeit* (engl.: *dependability*) nur ungenau abzugrenzen. Beispielsweise stellt sich die Frage, inwieweit die Verlässlichkeit sich vom Begriff der Zuverlässigkeit (engl.: *reliability*) abgrenzt. Ein Vorschlag einer solchen Definition für die Informatik, wie er ähnlich z.B. in [2, 14, 22, 29] zu finden ist, lautet:

Definition 6 (Verlässlichkeit) *Die* Verlässlichkeit *eines Systems ist das begründete Vertrauen darin, dass das System seine spezifizierte Funktionalität genau und rechtzeitig erbringt.*

Der Begriff Vertrauen ist in der Informatik diffizil, weil gerade im Bereich kritischer Geschäftsprozesse oder bei der Diskussion um die Hochverfügbarkeit von Systemen ein deterministisches Verhalten erwartet wird. Die Idee, Vertrauen in ein System zu haben, wird hierbei als nicht ausreichend empfunden. Das begründete Vertrauen, wie in dieser Definition genannt, soll jedoch auf der Betrachtung weiterer Dienstgütemerkmale basieren, die jeweils einen Aspekt der Verlässlichkeit eines Systems abdecken. Die Verlässlichkeit stellt in diesem Sinne eine Art Überbegriff dar, der sich in verschiedene Aspekte aufteilt.

Allerdings werden die Beziehungen zwischen diesen Aspekten in der Literatur unterschiedlich gehandhabt. Während beispielsweise Pohl [21] IT-Sicherheit als das zentrale Konzept be-

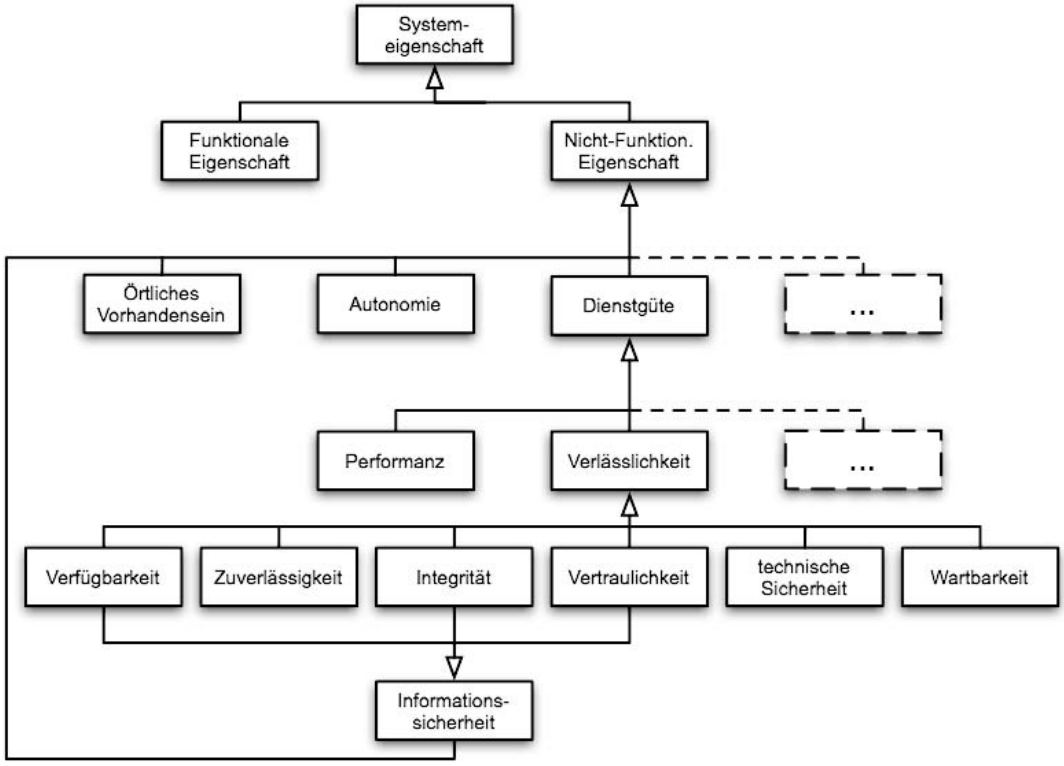


Abb. 2Konzeptionelles Modell der Systemeigenschaften.

trachtet, dem alle anderen üblichen Attribute der Verlässlichkeit untergeordnet werden, werden von Laprie [15] *security* und *safety* – die beiden Begriffe, die im Deutschen *Sicherheit* bezeichnen – der Verlässlichkeit untergeordnet. Zur Unterscheidung wird meist der Begriff *technische Sicherheit* für den englischen Ausdruck *safety* gebraucht, dagegen für *security* der Begriff *Datensicherheit*, der allerdings etwas unglücklich ist, da es dabei nicht ausschließlich um die Sicherheit von Daten geht. Datensicherheit wird in der Regel noch mit weiteren Aspekten untersetzt, typischerweise *Vertraulichkeit*, *Integrität* und *Verfügbarkeit*.[1] In der Wissenschaft hat sich heute mehrheitlich durchgesetzt, die Sicherheitsaspekte direkt der Verlässlichkeit zuzuordnen, vergleiche z.B. [1, 22, 29]. Gleichzeitig werden von Pohl die entsprechenden Aspekte dem Oberbegriff der Informationssicherheit zugeordnet [1]. Dadurch lässt sich keine verbindliche Hierarchie feststellen.

Die Diskussion der Dienstgüte im Allgemeinen und der Verlässlichkeit im Besonderen soll an dieser Stelle lediglich als ein Beispiel für ein bekanntes und gut definiertes Konzept dienen, an dem einerseits demonstriert wird, wie ein konzeptionelles Modell von Eigenschaftsbegriffen konstruiert werden kann und gleichzeitig, dass eine solches Modell nicht immer streng hierarchisch sein muss. In diesem Zusammenhang entsteht die Frage, inwiefern die Autonomie als Eigenschaft eines Systems zu verstehen ist. Die Autonomie in der Informatik bezieht sich zumeist auf das Verhalten eines Systems. Dabei wird ein System autonom genannt, wenn es in der Lage ist, sich autonom zu verhalten. Nach diesem Verständnis wird die Autonomie als nicht-funktionale Eigenschaft verstanden, orthogonal zu den klassischen Dienstgütemerkmalen verhält. Hierfür ist die Eigenschaft des örtlichen Vorhandenseins (z.B. von Ressourcen oder Diensten) ein Beispiel: Beim *örtlichen Vorhandensein* kann ein Dienst beispielsweise nur auf einem Server oder auch auf mobilen Endgeräten erbracht werden. Der Ort der Verfügbarkeit sagt jedoch nichts über seine Funktionalität aus.

Analog zum Beispiel des örtlichen Vorhandenseins ist die Eigenschaft des Systems zu sehen, eine Funktionalität autonom zu erbringen: Ein Router benötigt für die Erfüllung seiner Routingaufgaben Informationen über benachbarte Knoten. Diese können beispielsweise per Webbrowser durch einen Administrator eingegeben oder auch durch autonomes Verhalten selbstständig ermittelt werden. Ansätze hierfür sind bereits für unterschiedliche Anwendungen beschrieben worden (z.B. Herrmann et al. [7]). In beiden Fällen ändert sich die Funktionalität des Routers nicht. Allerdings ist die Maskierung von Fehlern durch Veränderungen in seinem Umfeld durch eine autonome Konfiguration des Routers verbessert – sofern diese kontinuierlich durchgeführt wird. Basierend auf dieser Überlegung leitet sich ein konzeptionelles Modell ab, welche die Begriffe zueinander in Relation setzt. Dieses Modell wird in Abb. 2 gezeigt.

In diesem Modell steht die Systemeigenschaft als Oberbegriff, die sich in funktionale und nicht-funktionale Eigenschaften unterteilt. Als nicht-funktionale Eigenschaften wurde bereits die Dienstgüte genannt. Als eine weitere nicht-funktionale Eigenschaft ist die Autonomie eines Systems zu betrachten. Im Vergleich zu Dienstgütemerkmalen weist die Autonomie folgende Unterschiede auf:

– Es ist zurzeit keine formale Methode bekannt, mit der sich die Autonomie eines Systems erfassen lässt.

1 Mitunter wird noch die *Zurechenbarkeit* oder *Authentizität* (engl.: acountacbility) hinzugenommen. Jedoch kann argumentiert werden, dass es sich hierbei nicht um eine *eigenständig* zu erzielende Eigenschaft handelt, sondern nur als Hilfsziel für die anderen Sicherheitseigenschaften, vergleiche [29].

– Mit dem Vorhandensein von Autonomie wird keine Güte eines Systems in positiver oder negativer Richtung impliziert.
– Würde man Autonomie nicht nur als duale Eigenschaft betrachten wollen, so ist zurzeit keine Einheit oder Metrik bekannt, nach der Autonomie bemessen werden könnte.

Daher ist die Autonomie nicht als Dienstgüte eines Systems zu sehen. Darüber hinaus kann autonomes Verhalten nicht durch ein System an sich repräsentiert werden, da dann die Frage offen bliebe, welche Funktionalität durch autonomes Verhalten abgedeckt würde. Autonomie stellt nur die Art dar, mit der eine Funktionalität erbracht wird. Damit ist Autonomie auch nicht notwendigerweise durch eine Komponente eines Systems zu repräsentieren. So verhält sich die Autonomie eines Systems bei dieser Betrachtung wie auch das örtliche Vorhandensein: Auch dies stellt keine Funktionalität eines Systems dar und wird nicht notwendigerweise durch einen Systembestandteil repräsentiert. Allerdings können Autonomie wie auch örtliches Vorhandensein durch Architekturmuster repräsentiert werden. In diesem Fall implementiert eine konkrete Architektur ein Architekturmuster.

Umgekehrt kann ein Architekturmuster nicht per se eine nichtfunktionale Eigenschaft repräsentieren: Beispielsweise kann die Zuverlässigkeit nicht durch ein spezielle Architektur repräsentiert werden, da diese Eigenschaft in verschiedenen Anwendungen unterschiedlich ausgeprägt ist. Zwar könnte eine redundante Auslegung von Systembestandteilen als Architekturmuster hierfür verstanden werden. Aber Redundanz stellt nicht zwangsläufig eine Erhöhung der Zuverlässigkeit eines Systems sicher. Auf diese Weise ergibt sich der Zusammenhang zwischen den verschiedenen Konzepten:

– Ein System implementiert eine Architektur.
– Eine Architektur oder ein Bestandteil einer Architektur kann wiederum ein Architekturmuster implementieren. Auf diese Weise stellt ein Architekturmuster eine Abstraktion einer Architektur dar.
– Ein Architekturmuster kann eine nicht-funktionale Eigenschaft ermöglichen bzw. verhindern.

Diese Aufstellung soll deutlich machen, dass autonomes Verhalten als Eigenschaft durch ein Architekturmuster beeinflusst werden kann. Und in einer Architektur kann dieses Architekturmuster aufgegriffen werden. Ein System realisiert autonomes Verhalten durch die Implementierung einer Architektur, die gegebenenfalls ein Architekturmuster konkretisiert.

4 AUTONOMIE IN IT-SYSTEMEN

Der vorherige Abschnitt hat den Begriff Autonomie eingeordnet, ohne dabei selbst zu erläutern, was unter Autonomie in der Informatik verstanden wird. Autonomie wurde als Eigenschaft eines Systems vorgestellt, welche als autonomes Verhalten erkennbar wird. Der folgende Abschnitt erläutert Autonomie in der Informatik in Hinblick auf technische Systeme.

Autonomie wurde bereits von Herstellern in der Computerindustrie als viel versprechender Ansatz zur Handhabung komplexer Infrastrukturen in der IT erkannt. Ein Ansatz in diesem Gebiet ist die *Autonomic Computing Initiative (ACI)* [10] von IBM. Diese Initiative hat das Ziel, existierende Systeme und Anwendungen mit *Self-X*-Eigenschaften auszustatten, wobei das *X* ein Platzhalter für unterschiedliche Managementaufgaben darstellt, die ein System selbstständig erbringen soll. Darunter fallen die Begriffe Selbstkonfiguration, Selbstheilung, Selbstsicherung und Selbstoptimierung (*Self-Configuration, Self-Healing, Self-Protection und Self-Optimisation*). Diese spiegeln die Bereiche wider, in denen IBM autonomes Verhalten eines Systems als besonders wichtigen Vorteil erachtet: Konfiguration, Reparatur, Schutz und Optimierung von Computersystemen.

Um die Forschung in Deutschland auf diesem Gebiet zu prägen und zu bündeln, haben die Informationstechnische Gesellschaft im VDE (ITG) und die Gesellschaft für Informatik (GI) ein Positionspapier formuliert [28]. In diesem Papier wird die Einschätzung dargelegt, dass Selbstorganisation, ein Begriff, der durch Autonomie subsumiert wird, die zukünftigen Computersysteme prägen wird. Diese Systeme sollen Self-X-Eigenschaften aufweisen, wie sie auch von IBM ins Spiel gebracht werden, um Administratoren zu entlasten und Computer stärker ins alltägliche Leben integrieren zu können. Das Verbesserungspotential selbstorganisierender Systeme wird hierbei vor allem bei der Handhabung komplexer Situationen und bei der Energieeinsparung gesehen. Gleichzeitig stellt die vorgestellte Vision klar, dass Sicherheitskonzepte für den wirtschaftlichen Einsatz erforderlich sind, die zu vertrauenswürdigen und verlässlichen Computersystemen führen. Dieser Aspekt wird insbesondere von IBM's ACI aufgegriffen.

Die Grundidee Computersysteme mit autonomem Verhalten zu entwickeln besteht darin, ein System mit Algorithmen und Verfahren derart zu erweitern, dass das Sicherstellen bestimmter Eigenschaften, ausgedrückt in einer *Zielstellung*, selbstständig verfolgt wird, ohne dass der Benutzer oder der Administrator dafür dedizierte Maßnahmen ergreifen müssten. Diese Idee ist nicht neu. Erste Ansätze lassen sich auf eine Arbeit von Dijkstra aus dem Jahre 1974 zurückverfolgen, in der ein selbststabilisierender Algorithmus zur Zirkulation eines Tokens in einem unidirektionalen Ring vorgestellt wurde [4]. Ansätze mit ähnlicher Zielsetzung basieren häufig auf einem grundlegendem Prinzip aus der Regelungstechnik: dem Regelkreis, der eine Regeleinrichtung aus Messort und Stellort bildet (vgl. Unbehauen [27]). Dieses Architekturmuster ist in der Informatik als *Observer/Controller Pattern* bekannt. Nach diesem Muster wird ein System von einem Controller in einer Schleife überwacht. Der Controller modifiziert aufgrund seiner Beobachtung einen oder mehrere Parameter des Systems derart, dass sich das System kontinuierlich einer Zielstellung annähert bzw. diese sicherstellt (*closed control loop*). Die Anwendung dieses Musters in der Informatik wurde bereits in vielen wissenschaftlichen Publikationen aufgegriffen (vgl. Herrmann et al. [8], Müller-Schloer et al. [18, 19, 23]). Müller-Schloer et al. sehen dieses Musters als Bestandteil einer *Organic Computing-Initiative*, die, durch die Natur inspiriert, neuartige Computersysteme mit Self-X-Eigenschaften ermöglichen möchte.

Nachdem bereits unterschiedliche Termini aus dem Bereich der Autonomie erwähnt wurden, stellt sich die Frage, was genau unter den verschiedenen Self-X-Eigenschaften verstanden wird. Hierzu soll zunächst für die Adaptivität die folgenden Definition als Basis aufgestellt werden [17, 30]:

> **Definition 7 (Adaptivität)** *Ein System ist* adaptiv *bezüglich einer Menge von Eingaben, wenn es in der Lage ist, für diese Eingaben seine Zielstellung zu erfüllen.*

Eine Eingabe wirkt hierbei ab dem Systemstart fortlaufend auf das System ein und kann manuelle Eingaben von Benutzern

und Administratoren, Eingaben von anderen Systemen sowie Umwelteinflüsse (z.B. Temperaturänderungen) umfassen. Ob ein System seine Zielstellung erfüllt oder nicht wird durch die Evaluation seines Verhaltens mittels einer *Performanzfunktion* und eines *Akzeptanzkriteriums* entschieden. Da die Festlegung der betrachteten Eingaben, der Performanzfunktion sowie des Akzeptanzkriteriums vom Beobachter – im Prinzip willkürlich – vorgenommen wird, ist nicht die Frage, *ob* ein System adaptiv ist, sondern bezüglich *welcher* Eingaben und *welcher* Performanzfunktion. Aufbauend auf einer parametrisierten Performanzfunktion lässt sich auch ein abgeleitetes *Gütemaß* definieren, mit dessen Hilfe sich die Adaptivität von Systemen bezüglich der gleichen Eingaben dann nicht nur qualitativ sondern auch quantitativ bewerten lässt. Neben der Fragestellung, ob ein System adaptiv ist, ist häufig auch von Interesse, wie das System dies erreicht. Zum Beispiel kann sich das System intern mittels eines *Adaptionsprozesses* fortlaufend an die anliegende Eingabe anpassen.

Adaptive Systeme lassen (um ihre Zielstellung zu erreichen) auch Kontrolleingaben zu (z.B. kann eine Klimaanlage manuell ein- und ausgeschaltet werden, um die Raumtemperatur in einem definierten Intervall um die Solltemperatur zu halten). Will man die Kontrolleingaben vermeiden, so kommt man zu den selbstverwaltenden Systemen [17]:

> **Definition 8 (Selbstmanagement/Selbstverwaltung)** *Ein System ist* selbstverwaltend (selbstmanagend)*, wenn es adaptiv ist, ohne von außen kontrolliert zu werden.*

Diese Definition von Selbstverwaltung impliziert eine Unterteilung der Systemeingabe in *Kontrolleingaben* und *Regulärengaben* (vgl. [16]). Nur ein System, welches *keine* Kontrolleingaben erhält, wird als selbstverwaltend bezeichnet. Lendaris führt hierzu aus, dass die Performanzfunktion festlegt, welche Eingaben als Kontroll- und welche als Reguläreingaben zu sehen sind. Selbstverwaltende Systeme können mittels des Observer/Controller Patterns realisiert werden, indem ein nicht selbstverwaltendes System um einen Controller erweitert wird, welcher die benötigten Kontrolleingaben bereitstellt. Autonomie und Selbstverwaltung sind aus unserer Sicht Synonyme. Eine Möglichkeit, selbstverwaltende Systeme zu realisieren ist die Selbstkonfiguration, wobei die *Konfiguration* eines Systems hierbei ein Teil des Systemzustands ist, welcher das Systemverhalten wesentlich beeinflusst:

> **Definition 9 (Selbstkonfiguration)** *Ein System ist* selbstkonfigurierend*, wenn es seine Konfiguration ohne externen Kontrolleingriff anpasst, um seine Adaptivität sicherzustellen.*

Das grundlegende Ziel von Selbstkonfiguration ist, dem menschlichen Administrator die Konfiguration aus genannten Gründen weitestgehend zu ersparen. Eng verwandt mit der Selbstkonfiguration ist der Begriff *Selbstoptimierung*, der zusätzlich betont, dass ein Optimierungsziel durch fortlaufende Selbstkonfiguration verfolgt werden soll. *Selbstschutz* wiederum ist die Selbstkonfiguration mit dem Ziel der Sicherstellung von Sicherheitseigenschaften.

Ein weiterer Begriff im Bereich der autonomen Systeme ist die Selbstorganisation. Dieser bezieht sich auf die Struktur eines Systems. Selbstorganisierende Systeme verändern selbständig ihre Struktur, um trotz einer sich ändernden Umwelt eine Zielstellung zu erfüllen. Diese Eigenschaft selbstorganisierender Systeme wird auch als *Strukturadaptivität* bezeichnet. Hierbei kann Struktur Beziehungen zwischen den Systembestandteilen im Sinne einer Architektur beschreiben (z.B. Kommunikationsbeziehungen). Zusätzlich wird eine dezentrale Kontrolle des Systems verlangt, um einen Single-Point-of-Failure auszuschließen [9, 17]:

> **Definition 10 (Selbstorganisation)** *Ein* selbstorganisierendes *System ist ein* selbstverwaltendes *System, welches zusätzlich strukturadaptiv ist und eine dezentrale Kontrolle hat.*

Es ist sinnvoll, Kenntnisse über die Struktur eines selbstorganisierenden Systems, die innerhalb des Entwurfsprozesses gewonnen wurden, beim Betrieb des Systems mit zu verarbeiten. Dies kann z.B. durch die Anwendung eines modellgetriebenen Entwurfprozesses erreicht werden. Durch diese Vorgehensweise können die spezifischen Eigenschaften einer Anwendung bezüglich der Selbstorganisation in dem realisierten System berücksichtigt werden.

Selbststabilisierung ist eine Eigenschaft, die für Computersysteme sehr erstrebenswert ist. Ein selbststabilisierendes System ist in der Lage, sich von beliebigen transienten Fehlern zu erholen, wenn für eine gewisse Zeit (die *Stabilisierungszeit*) keine weiteren Fehler auftreten. Das heißt, es darf während der Stabilisierung ein beliebiges, inkorrektes Verhalten aufweisen, muss aber spätestens nach Ablauf der Stabilisierungszeit wieder ein korrektes Verhalten aufweisen. Im Gegensatz hierzu können Systeme, die nicht selbststabilisierend sind, eventuell für alle Zukunft ein inkorrektes Verhalten aufweisen, falls ein Fehler auftritt, der nicht maskiert werden kann. Die Definition selbststabilisierender System geschieht hingegen üblicherweise auf Basis einer Unterteilung der Zustände in legale und illegale Zustände [24]:

> **Definition 11 (Selbststabilisierung)** *Ein System ist* selbststabilisierend*, wenn es im fehlerfreien Fall,* (1) *ausgehend von einem beliebigen Zustand einen legalen Zustand in beschränkter Zeit erreicht und* (2) *ausgehend von einem legalen Zustand seinen Zustand in der Menge der legalen Zustände hält.*

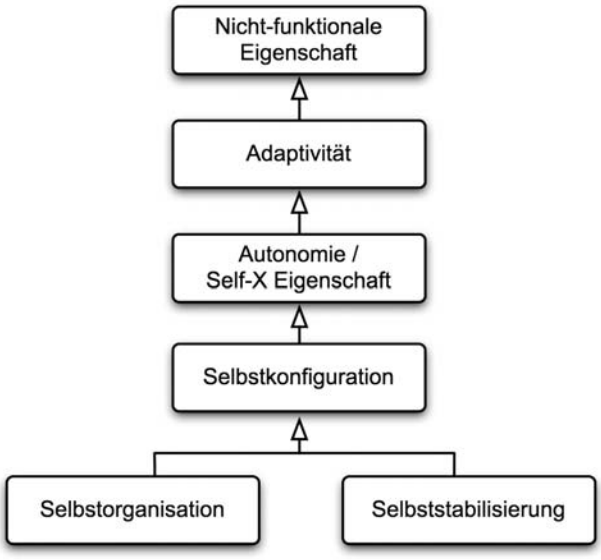


Abb. 3 Konzeptionelles Modell von Adaptivität und Autonomie.

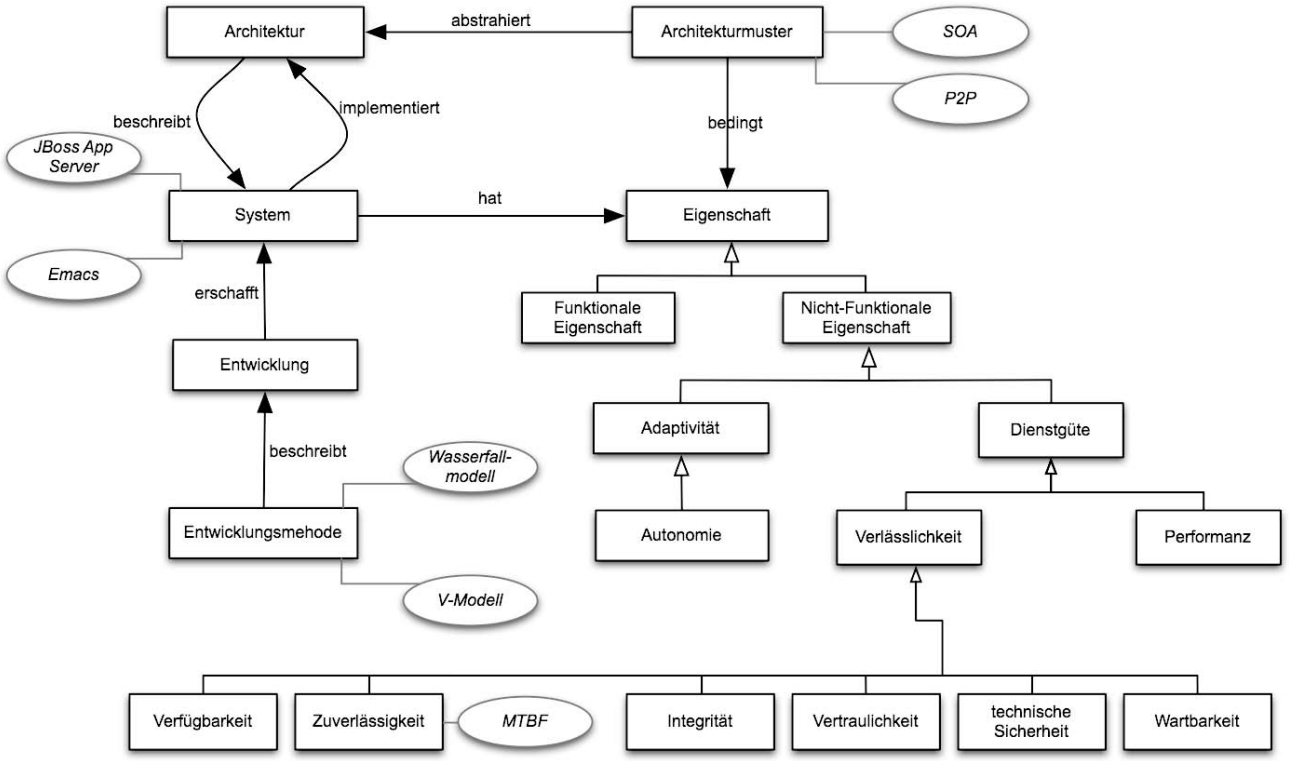


Abb. 4 Zusammenfassung der Konzepte und deren Zusammenhänge.

Selbststabilisierung kann mit Fehlermaskierung kombiniert werden, um für *bestimmte* Fehler auch ein nur zeitweises Versagen des Systems zu verhindern. Des Weiteren kann durch *Superstabilisierung* [5] und *Fault Containment* [6] erreicht werden, dass für bestimmte Fehlerklassen kein beliebiges sondern ein definiertes Verhalten auftritt bzw. dass die Auswirkungen von Fehlern auf einzelne Systemteile beschränkt bleiben. Der Begriff Selbstheilung ist als Synonym zur Selbststabilisierung zu sehen. Die gegebene Definition von Adaptivität impliziert, dass Autonomie als Spezialisierung derselbigen aufgefasst wird. Abb. 3 zeigt diesen Zusammenhang auf und ordnet auch die weiteren behandelten Begriffe ein. Im Gegensatz zu Abb. 2 wird in dieser Abbildung die Differenzierung zwischen Adaptivität und Autonomie eingeführt.

5 DAS ZUSAMMENGEFASSTE MODELL

In Abb. 4 werden die bisher dargelegten Konzepte in einen gemeinsamen Zusammenhang gestellt. Diese Abbildung greift die Elemente aus Abb. 1 auf und arrangiert um diese die Konzeption der Eigenschaften, die in Abb. 2 etwas detaillierter gezeigt wurde. Aus Platzgründen wurden hierbei nicht alle Konzepte aus Abb. 2 übernommen. Hinzu kommen in diesem Modell konkrete Beispiele, die Ausprägungen der jeweiligen Konzepte darstellen. Im objektorientierten Sinn wären dies Exemplare einer Klasse. Entsprechende Exemplare sind durch ovale Felder gekennzeichnet, während für Konzepte Rechtecke beibehalten wurden.

Diese Beispiele greifen die erläuterte Kategorisierung auf: Bei der SOA oder einem Peer-to-Peer (P2P) Netzwerk bezeichnet man Architekturmuster, die von einer konkreten Architektur abstrahieren und daher in verschiedenen Architekturen Verwendung finden können. Das im vorherigen Abschnitt behandelte Konzept der Entwicklungsmethode beinhaltet Vorgehensmodelle, wie das V-Modell oder das Wasserfallmodell. Analog dazu sind, um den Begriff *System* zu verdeutlichen, konkrete Systeme genannt, die zu der Auffassung dieses Begriffs passen. In dieser Abbildung sind als Beispiele der Anwendungsserver JBoss oder das Softwarepaket Emacs genannt. Um das Konzept der Verlässlichkeit zu konkretisieren, zeigt die Abbildung die Mean-Time-Between-Failure (MTBF) als konkrete Ausprägung der Zuverlässigkeit.

6 FAZIT

Das vorgestellte konzeptionelle Modell stellt die genannten Begriffe in Beziehung zueinander und kann dadurch von Forschungsarbeiten aufgegriffen werden, die sich in diesem Bereich bewegen. Das vorgestellte Modell bietet auch die Grundlage, sich mit den aufgezeigten Beziehungen zwischen den Termini weiter auseinander zu setzen. Ziel dieser Arbeit ist die Etablierung eines konsistenten Modells bezüglich der individuellen Begriffe und deren Relationen zueinander. Hierfür ist zu beachten, dass bezüglich zweier Konzepte in einzelnen Anwendungsfällen auch andere Sichtweisen möglich sind:

– **Architekturmuster.** Die Einführung des Architekturmusters als Abstraktion einer Architektur ist eine künstliche Trennung vom Begriff der Architektur. Die Architektur in Hinblick auf ein System, das durch sie beschrieben wird, stellt im gewissen Sinne bereits eine Abstraktion dar. Es wäre daher auch möglich von einem Abstraktionsgrad bei Architekturen zu sprechen. Der in dieser Diskussion dargestellte Begriff des Architekturmusters hat dennoch seine Berechtigung, weil diese er die Trennung zwischen Begriffen wie SOA oder MDA und einer konkreten Architektur eines Systems ermöglicht.

– **Autonomie als Eigenschaft.** Die Autonomie als nicht-funktionale Eigenschaft eines Systems zu interpretieren hat den Vorteil, dass der Blick auf die eigentliche Funktion eines Systems geschärft wird. Anders ausgedrückt fordert diese Sichtweise: Die Funktionalität eines Systems ist nicht die Autonomie; es stellt sich vielmehr die Frage, welche Funktion autonom erbracht werden soll.

  Andererseits kann einem Systembestandteil die Implementierung von autonomen Verhalten als Subsystem zugewiesen werden. In diesem Fall stellt das autonom agierende Subsystem einen funktionalen Kern zu Verfügung. Die Konsequenz wäre dann, dass die Autonomie eine funktionale Eigenschaft eines (Sub-)Systems darstellt. Für die Konzeption des hier vorgestellten Modells hat die Parallele zum örtlichen Vorhandensein in Hinblick auf die Plausibilität überwogen, weswegen hier die Autonomie als nicht-funktionale Eigenschaft dargestellt wird.

Insgesamt streben die Autoren an, dass durch den vorliegenden Text, die zusammengestellten Definitionen und das daraus abgeleitete konzeptionelle Modell die wissenschaftliche Auseinandersetzung bezüglich der Interpretation der genannten Begriffe weiter fortgesetzt wird. Das Ziel ist Bildung eines Konsenses bezüglich dieser Begriffe, um so eine begriffliche Grundlage für weitere Arbeiten in der Informatik zu bilden

Danksagung.
Die Autoren danken den Gutachtern für ihre detaillierten und konstruktiven Anmerkungen und Kommentare.

LITERATUR

[1] Algirdas Avizienis; Jean-Claude Laprie; Brian Randell; and Carl Landwehr: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Comput., 1 (1): 11-33, 2004.

[2] William C. Carter: A time for reflection. In Proceedings of the 8th IEEE International Symposium on Fault Tolerant Computing (FTCS-8), Seite 41, Santa Monica, CA, USA, Juni 1982.

[3] Deutsches Institut für Normung e.V. DIN 44300-1, 1988.

[4] Edsger W. Dijkstra: Self-stabilizing systems in spite of distributed control. Comm. of the ACM, 17 (11): 643-644, 1974.

[5] Shlomi Dolev and Ted Herman: Superstabilizing protocols for dynamic distributed systems. Chicago Journal of Theoretical Computer Science, 4. Special Issue on Self-Stabilization, Dezember 1997.

[6] Sukumar Ghosh; Arobinda Gupta; Ted Herman; and Sriram Pemmaraju: Fault-containing self-stabilizing algorithms. In Proceedings of the Fifteenth Annual ACM Symposium of Distributed Computing (PODC96), Seiten 45-54. ACM, 1996.

[7] Klaus Herrmann; Kurt Geihs; and Gero Mühl: Ad hoc service grid – a self-organizing infrastructure for mobile commerce. In Proceedings of the IFIP TC8 Working Conference on Mobile Information Systems (MOBIS 2004), Seiten 261-274, Oslo, Norwegen, September 2004. Kluwer Academic.

[8] Klaus Herrmann; Gero Mühl; and Kurt Geihs: Self-management: The solution to complexity or just another problem? IEEE Distributed Systems Online (DSOnline), 6 (1), Januar 2005.

[9] Klaus Herrmann; Matthias Werner; Gero Mühl; and Hans-Ulrich Heiß: Ein methodischer Ansatz zur Klassifizierung selbstorganisierender Softwaresysteme. In Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme (SAKS '06), Kassel, März 2006.

[10] IBM. An architectural blueprint for autonomic computing (4th edition). IBM White Paper, IBM Corporation, Juni 2006.

[11] ISO/IEC. ITU.TS Recommendation X.902 – ISO/IEC 10746-2: Open Distributed Processing Reference Model – Part 2: Foundations, August 1996.

[12] ISO/IEC. ISO/IEC 7498-1: Information Technology – Open Systems Interconnection – The Basic Model, 1997.

[13] ISO/IEC. ITU-T Recommendation X.641 – ISO/IEC 13236: Information Technology – Quality of Service: Framework, 1998.

[14] Jean-Claude Laprie: Dependability: From concepts to limits. In 12th IFAC International Conference on Computer Safety, Reliability and Security, 1993.

[15] Jean-Claude Laprie, editor: Dependability: Basic Concepts and Terminology, volume 5 of Dependable Computing and Fault Tolerance. Springer, 1992.

[16] George G. Lendaris: On the definition of self-organizing systems. Proceedings of the IEEE, 52: 324-325, 1964.

[17] Gero Mühl; Matthias Werner; Michael A. Jaeger; Klaus Herrmann; and Helge Parzyjegla: On the definitions of self-managing and self-organizing systems. In: KiVS 2007 Workshop: Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme (SAKS 2007), Informatik Aktuell. Springer, 2007.

[18] Christian Müller-Schloer; Christops von der Malsburg; and Rolf P. Würtz: Organic computing. Informatik Spektrum, 27 (4): 332-336, August 2004.

[19] Christian Müller-Schloer: Organic computing: On the feasibility of controlled emergence. In: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '04), Seiten 2-5, 2004.

[20] Object Management Group (OMG): UML Profile for Modelling Quality of Service and Fault Tolerance Characteristics and Mechanisms. ptc/2004-06-01, Juni 2004.

[21] Hartmut Pohl: Taxonomie und Modellbildung in der Informationssicherheit. Datenschutz und Datensicherheit, 28 (11): 678-685, 2004.

[22] B. Randell: Dependability-a unifying concept. In: Computer Security, Dependability, and Assurance: From Needs to Solutions, 1998, York, UK & Williamsburg, VA, USA, Seiten 16-25. IEEE, 1999.

[23] Urban Richter; Moez Mnif; Jürgen Branke; Christian Müller-Schloer; and Hartmut Schmeck: Towards a generic observer/controller architecture for organic computing. In: INFORMATIK 2006 – Informatik für Menschen, LNI P-93, Seiten 112-119, Bonn, September 2006. Bonner Köllen Verlag.

[24] Marco Schneider: Self-stabilization. ACM Computing Surveys (CSUR), 25 (1): 45-67, 1993.

[25] Andrew S. Tanenbaum and Maarten van Steen: Distributed Systems Principles and Paradigms. Prentice Hall, Upper Saddle River, New Jersey, USA, 2002.

[26] Technical Committee ISO/TC 176: Quality Management and Quality Assurance. Quality Management and Quality System Elements; Part 2: Guidelines for Services, 1991.

[27] Heinz Unbehauen: Regelungstechnik. Vieweg, 1989.

[28] VDE, ITG, GI: Organic computing: Computer- und Systemarchitektur im Jahr 2010, VDE/ITG/GI Positionspapier, 2003.

[29] Matthias Werner: Eigenschaften Verlässlicher Systeme. Habilitationsschrift der Technischen Universität Berlin, 2007.

[30] Lotfi A. Zadeh: On the definition of adaptivity. Proceedings IEEE (Correspondence), 51: 469-470, März 1963.