Übersicht

Hubert B. Keller*, Oliver Schneider, Jörg Matthes und Veit Hagenmeyer

Zuverlässige und sichere Software offener Automatisierungssysteme der Zukunft – Herausforderungen und Lösungswege

Reliable, safe and secure software of connected future control systems – challenges and solutions

DOI 10.1515/auto-2016-0060
Eingang 1. April 2016; angenommen 31. Oktober 2016

Zusammenfassung: In der modernen Automatisierung ist die Zuverlässigkeit der Software grundlegend und durch konstruktive Maßnahmen zu sichern. Echtzeitsysteme müssen über Schedulingverfahren auf Basis des Prozesskonzepts der Automatisierungstechnik beweisbar die zeitlichen Anforderungen im worst case erfüllen. Fehlervermeidende Sprachen mit Typprüfungen sind notwendig für die Erfüllung der Zuverlässigkeit. Die Anforderungen der Informationssicherheit sind in offenen Automatisierungssystemen integrativ zu erfüllen. Schwachstellen in der Implementierung oder den additiven Sicherheitsmechanismen stellen eine inakzeptable Gefährdung dar.

Schlüsselwörter: Zuverlässigkeit, Sicherheit, Vernetzung, Echtzeit, Schwachstellen.

Abstract: Reliability of software is fundamental in modern control systems and has to be secured by constructive methods. Real time systems must guarantee real time requirements under worst case conditions based on scheduling algorithms and the process concept of computer science (operating system). Error avoiding languages with strong typing is fundamental for achieving reliability. Connected control systems have to satisfy information security requirements in an integrative manner. Vulnerabilities of the implementation or of security equipment represent unacceptable weaknesses.

***Korrespondenzautor: Hubert B. Keller,** Karlsruher Institut für Technologie (KIT), Institut für Angewandte Informatik (IAI), Kaiserstr. 12, 76131 Karlsruhe, E-Mail: hubert.keller@kit.edu
Oliver Schneider, Jörg Matthes, Veit Hagenmeyer: Karlsruher Institut für Technologie (KIT), Institut für Angewandte Informatik (IAI), Kaiserstr. 12, 76131 Karlsruhe

Keywords: Reliability, Safety, Security, Interconnection, Vulnerabilities, Real Time.

Zum 70. Geburtstag von Herrn Prof. Dr.-Ing. habil. Georg Bretthauer

1 Einleitung

Die softwarebasierte Automatisierung technischer Prozesse unter Einhaltung zeitlicher Bedingungen ist die Basis unserer Gesellschaft in allen Bereichen:

- Verfahrens- und produktionstechnische Anlagen
- Kritische Infrastrukturen wie Energieverteilungssysteme und Verkehrsleitanlagen
- Home-Automation-Systeme
- Automotive Systeme
- Transportsysteme
- Medizinische Systeme.

Aufgrund der Öffnung und Vernetzung automatisierter Systeme kommen Security-Aspekte und deren Wechselwirkung mit vorhandenen Safety-Anforderungen auf das einzelne System als auch die Vernetzungsstrukturen hinzu (vgl. [9, 32, 35]). Im vorliegenden Beitrag steht das einzelne System als Teil des Netzwerks im Fokus.

Die Zuverlässigkeit der softwarebasierten Funktionen ist eine zentrale und konstruktiv umzusetzende Anforderung und hat Auswirkungen auf Safety und Security Aspekte gleichermaßen. Die weitere Zukunft bringt eine noch stärkere Integration der Funktionalität (Funktionsvernetzung), eine noch höhere Integration der Daten mit intelligenter, interpretativer Auswertung in allen Bereichen und mit Smart Energy Grids [29], Cyber-Physical Systems [1, 28, 76] sowie Industrie 4.0 [2] eine deutlich höhere Systemkomplexität. Gleichzeitig werden die Hardware-

komponenten kleiner und durch Mehrkernrechner leistungsfähiger.

Softwarebasierte Funktionen sind in der Automatisierung zuverlässig und zeitlich deterministisch auszuführen. Sowohl der Entwurf, die Realisierung als auch der Betrieb müssen die Zuverlässigkeit und eine sichere Benutzung garantieren (vgl. [7]).

Bei allen zukünftigen Entwicklungen müssen aber die grundlegenden Eigenschaften von Echtzeitsystemen die zentralen Prinzipien bleiben. Das sind (siehe z. B. [7, 14, 21, 51, 54]):

- Rechtzeitigkeit
- Gleichzeitigkeit
- Determiniertheit
- unterbrechungsfreier Betrieb (24 h)
- Betriebssicherheit (technische Sicherheit)
- Informationssicherheit
- Zuverlässigkeit.

Rechtzeitigkeit bedeutet, dass die zeitliche Reaktion entsprechend der Dynamik des zu kontrollierenden Prozesses erfolgt. Eine kürzere Zykluszeit als physikalisch notwendig verursacht höhere Kosten, liefert aber kein besseres Ergebnis. In manchen Fällen ist eine schnellere Reaktion nicht nur unnötig, sondern falsch (zündet z. B. ein Airbag zu früh, fällt er in sich zusammen und federt den Aufprall nicht ab; zündet der Airbag zu spät, gilt gleiches).

Gleichzeitigkeit erfordert eine Reaktion auf gleichzeitig eintreffende und zu behandelnde Ereignisse. Im automatisierten Prozess laufen die realen Vorgänge parallel ab und müssen, jeder für sich und in der Summe, rechtzeitig behandelt werden.

Determiniertheit erfordert die Vorhersagbarkeit, die Wiederholbarkeit und die Nachvollziehbarkeit des Verhaltens eines Echtzeitsystems unter allen möglichen Bedingungen. In jedem Zustand des Systems muss bei jeder Eingabe der Folgezustand, die Ausführungszeit des Übergangs und die Ausgabe bekannt sein und für jede identische Vorbedingung exakt das gleiche Ergebnis erfolgen. Diese mathematische und informationstheoretische Forderung ist für Funktionen elementar. Bei komplexen Systemen mit hoher Wechselwirkung ist das sich ergebende Gesamtverhalten (Wechselwirkungsfunktion) ebenfalls deterministisch geplant, aber aufgrund einer hohen Informationsabhängigkeit nicht immer erkennbar oder erreichbar. In [7] wird dazu ausgeführt, dass „zwar die zu verarbeitenden Daten einer zufälligen Verteilung unterliegen, daraus aber nicht der Schluss der Nichtdeterminiertheit gezogen werden darf". Die „auszuführenden Reaktionen müssen jedoch genau geplant und vorhersehbar sein". Zu komplexen vernetzten Systemen kann aus [20] „Simon's Law" angeführt werden: „Hierarchical structures reduce complexity" und damit das Fehlerrisiko nach [62]. Auch bei großen Bandbreiten verteilter Systeme sind nichtdeterministische Latenzzeiten in den Übertragungssystemen vorhanden. Für die in sicherheitskritischen Anwendungen mit definierten Reaktionszeiten notwendige „worst case" Betrachtung sind solche nichtdeterministischen Latenzzeiten nicht zulässig, unbrauchbar bzw. müssen mit den schlechtesten Werten berücksichtigt werden. Die „Vorhersehbarkeit ihres Ausführungsverhaltens und ihre Verlässlichkeit" sind nach [7] Hauptmerkmale eines Echtzeitsystems.

Unterbrechungsfreie Ausführung ist für Echtzeitsysteme ebenfalls von grundlegender Bedeutung, da diese unvorhergesehene Abschaltungen oder Neustarts verbieten. Unbegrenzte Speicheranforderungen, Verletzungen zeitlicher Randbedingungen oder das Anwachsen von Fehlergrößen oder auch Verklemmungen dürfen nicht auftreten.

Betriebssicherheit oder technische Sicherheit garantiert, dass der Betrieb des Echtzeitsystems für das Umfeld ein Risiko unterhalb der tolerierbaren Schwelle hat. Gefahr für Leib und Leben, massive Sach- oder Geldschäden sind ausgeschlossen. Das verbleibende Restrisiko ist tolerabel (vgl. [23, 77]).

Informationssicherheit erfordert, dass Daten nicht verändert sind oder ein Absender vorgetäuscht wird und dadurch falsche Reaktionen ausgelöst werden. Das kann erfordern, dass Sensoren eine Identität erhalten und sich gegenüber dem Empfänger oder einer anderen Instanz ausweisen müssen, dass zu übertragende Daten verschlüsselt oder dass Übertragungswege gesichert werden (vgl. [23]). Allerdings sind durch Schwachstellen in der Software (vgl. [66]) auch bei korrektem Kommunikationsverhalten durch bestimmte Bitmuster („handcrafted" Daten bzw. Kommandos und Parameterwerte) Manipulationen möglich.

Zuverlässigkeit in der Ausführung der geforderten Funktionen erfordert, dass die Zielfunktion über einen definierten Zeitraum durchgehend verfügbar ist und korrekt ausgeführt wird (siehe z. B. [7]). Bei softwarebasierten Funktionen ist die Zuverlässigkeit im Gegensatz zu Komponenten z. B. aus dem Maschinenbau nicht über die klassische Zuverlässigkeitstheorie erschließbar. Nach [18] gilt bei Software, dass diese weder Alterung noch Verschleiß unterliegt. Einschränkungen der Zuverlässigkeit ergeben sich durch vorhandene und nicht erkannte Konstruktionsfehler. Für den Restfehlergehalt gibt es für den Reifegrad der Software verschiedene Ansätze. Diese Reifegradmodelle über die empirische Bewertung der Nutzung von Software liefern Aussagen nur bei gleichartigen Nutzungsprofilen. Neuere Ansätze versuchen Software in ihren

Komponenten zu modellieren und das Gesamtsystem aus dem Reifegrad aller Komponenten zu bewerten [8]. Diese Verallgemeinerung ist jedoch nicht möglich, da Software in ihren Ablaufstrukturen bzgl. den eingebbaren Werten nicht kontinuierlich bzw. unstetig ist. Systeme aus Komponenten hängen sehr stark von den Wechselwirkungseffekten im integrierten System ab [68]. Daher können aus Erfahrungen unter bestimmten Betriebsbedingungen keine allgemeingültigen Schlüsse über das Gesamtverhalten bei Veränderungen in den Eingabedaten gezogen werden. Aufgrund des großen Eingaberaums ist es nicht realistisch alle möglichen Eingaben zu testen. Da nicht alle Eingaben getestet werden, können Eingaben existieren, die zwar akzeptiert werden, jedoch zu unerwünschtem Verhalten führen. Die Korrektheit einer Funktion erfordert, dass bei zulässigen Eingabedaten der entsprechend korrekte Ausgabewert berechnet wird. Bei unzulässigen Eingabedaten sollte eine Fehlermeldung erfolgen oder eine, nicht auf den Eingabedaten basierte, Ersatzaktion durchgeführt werden. Dies setzt voraus, dass die Eingabedaten hinsichtlich ihrer Korrektheit geprüft werden, indem alle technisch möglichen Eingaben abgedeckt werden.

In [80] wird am Beispiel der Sprache C und dem Betriebssystem Unix dargestellt, wie Schwachstellen in der Implementierung für eine Manipulation verwendet werden können und damit unerwünschter Code ausgeführt werden kann. In Konsequenz kann der Angreifer die Kontrolle über das attackierte System übernehmen. Die überwiegende Anzahl der Security Schwachstellen können laut [80] auf diese Implementierungsschwachstellen zurückgeführt werden. Dies wird auch durch eine Studie des National Institute of Standards and Technology [58] bestätigt. Insbesondere die Sprachen C, C++ oder auch Java werden dort als „languages in which most of today's vulnerabilities have been identified" aufgeführt. Die sprachbedingten Schwachstellen [66] betreffen nicht nur Anwendungssysteme, sondern auch Betriebssysteme und gerade auch Security-Infrastruktursysteme wie Security-Router, virtuelle Maschinen, virtuelle private Netzwerke (VPN) etc.

Der vorliegende Beitrag analysiert im Folgenden die Schwachstellen, die aus der programmtechnischen Realisierung von Automatisierungssystemen resultieren und überwiegend für Angriffspunkte im Sinne Security ausgenutzt werden (vgl. [35, 74, 80]). Der Beitrag will ein kritisches Bewusstsein für die Art der zugrundeliegenden Problematik ermöglichen und eine entsprechend dringende wissenschaftliche Diskussion und Auseinandersetzung anregen. Nach den Meldungen des US-CERT [74] wurden allein bei CISCO Systemen 294 Schwachstellen innerhalb eines Jahres gefunden – Tendenz steigend, bei Juniper Private Network Systemen waren es 45 Schwachstellen innerhalb eines Jahres. Bei CoDeSys von 3S Smart Software Solutions, einem Tool für Automatisierungssysteme, ergaben sich 12 Schwachstellen innerhalb eines Jahres. Bei einem Windturbinen-Betriebssystem wurden 6 Schwachstellen innerhalb eines Jahres gefunden. Die überwiegende Ursache waren dabei fehlende typstrenge Prüfungen zur Laufzeit. Die Konsequenzen der Schwachstellen reichten vom „Ausführen beliebigen Codes" bis zur Abschaltung ganzer Anlagen (Windturbine). Eine Liste der bisher entdeckten Schwachstellen nach Hersteller ist unter https://ics-cert.us-cert.gov/alerts-by-vendor zu finden.

Die Schwachstellen aus der programmtechnischen Realisierung bieten erhebliche Angriffspunkte und sind durch geeignete Sprachen und Vorgehensweisen für zukünftige Automatisierungssysteme zu vermeiden. Security-Einrichtungen weisen, wie oben dargelegt, analoge Schwachstellen auf.

Auf diesem Hintergrund ist der vorliegende Beitrag wie folgt strukturiert: Im Abschnitt 2 wird der Status in der Entwicklung von Software dargestellt. In Abschnitt 3 werden die Echtzeitanforderungen und deren Erfüllung erläutert. Der Abschnitt 4 diskutiert die programmtechnische Realisierung von zuverlässigen Automatisierungsfunktionen unter Echtzeitaspekten und der Abschnitt 5 zeigt die integrative Betrachtung von Safety und Security auf. In Abschnitt 6 wird der vorliegende Beitrag zusammengefasst und es werden Hinweise für die zukünftigen Entwicklungen gegeben.

2 Status des Entwicklungsstandes von Software für die Automatisierungstechnik

Die Entwicklung zuverlässiger Software ist notwendig für sichere Systeme im Sinne Safety und Security. Die hohen Wartungskosten von Software, wie nachfolgend gezeigt, belegen merkliche Defizite in deren Herstellung, die auch zu angreifbaren Schwachstellen führen. Neben der Erfüllung der geforderten Spezifikation ist zudem sicher zu stellen, dass keine weiteren Verhaltensweisen in der Ausführung von Softwarefunktionen entstehen können. Nur wenn sichergestellt ist, dass ausschließlich die Spezifikation erfüllt wird, können z. B. Security Angriffe mit „handcrafted" Werten sicher verarbeitet werden. Nach Isermann [40] entstehen

> mechatronische Systeme durch Integration von vorwiegend mechanischen Systemen, elektronischen Systemen und zugehöriger Informationsverarbeitung. Wesentlich sind dabei die Integration der einzelnen mechanischen und elektronischen

RESOLUTION

	2004	2006	2008	2010	2012
Successful	29%	35%	32%	37%	39%
Failed	18%	19%	24%	21%	18%
Challenged	53%	46%	44%	42%	43%

Project resolution results from CHAOS research for years 2004 to 2012.

Abbildung 1: Erfolg bei Softwareprojekten [70].

> Elemente, die dadurch mögliche Erweiterung von Funktionen und die Erzielung synergetischer Effekte.

Weiter führt Isermann aus:

> Die Integration kann örtlich durch den Aufbau und funktionell durch die digitale Elektronik erfolgen. Die örtliche Integration erfolgt dabei im Wesentlichen durch die konstruktive Verschmelzung von Aktoren, Sensoren und der Mikroelektronik mit dem Prozess, also durch die Hardware. Die funktionelle Integration wird jedoch entscheidend durch die Informationsverarbeitung und damit durch die Gestaltung der Software geprägt.

Die Erfolge in der Entwicklung und Herstellung von Software sind noch nicht auf gleicher Ebene wie im klassischen Maschinenbau. Die Standish Group hat in ihrem Chaos-Report (vgl. [70]) Projekte hinsichtlich ihres Abschlusses untersucht. Kriterien waren dabei, ob ein Projekt erfolgreich, verzögert, mit verringertem Funktionsumfang, mit höheren Kosten oder gar nicht abgeschlossen wurde (vgl. Abbildung 1).

Obwohl nach der sogenannten „Softwarekrise" in den achtziger Jahren der methodische Aufwand und der Werkzeugeinsatz verstärkt wurden, sind letztlich die großen Erfolge in der Breite ausgeblieben. Es sind also weitere Faktoren zu berücksichtigen, die sich auf die Kompetenz der Mitarbeiter, die Vorgehensweise und auch die eingesetzte Programmiersprache gründen. Ganssle führt in [25] aus, dass Programme in C/C++ gegenüber Ada um den Faktor 10 höhere Fehlerraten aufweisen. Weiterhin zeigt er am Beispiel einer Infusionspumpe, dass auch bei Einsatz von Regelwerken und Normen mit Sprachen wie C/C++ eine merkliche Zahl kritischer Fehler beim Test unentdeckt bleibt. Nach dem Rückzug vom Markt aufgrund von Todesfällen lieferte eine statische Analyse 127 Restfehler.

In der Analysephase injizierte Fehler sind bei Behebung in späteren Phasen mit extremen Kosten und damit massiven zeitlichen Verzögerungen verbunden (vgl. Abbildung 4). Programmiersprachen, die explizit eine Fehlervermeidung unterstützen, stellen damit einen merklichen finanziellen Erfolgsfaktor von softwarebasierten Projekten dar.

Grundlegend ist und bleibt die Zuverlässigkeit der softwarebasierten Funktionen. Beispielsweise muss das Antiblockiersystem (inkl. seiner Software-Features) im Automobil während der Fahrt permanent funktionsfähig sein. Zuverlässigkeit erfordert, dass die Zielfunktion über einen definierten Zeitraum permanent verfügbar ist und korrekt ausgeführt wird. Bei softwarebasierten Funktionen ist die Zuverlässigkeit im Gegensatz zu Komponenten z. B. aus dem Maschinenbau nicht über die klassische Zuverlässigkeitstheorie erschließbar. Zur Bewertung des Reifegrades werden Zuverlässigkeitsmodelle eingesetzt (vgl. [18]). Mit verschiedenen Modellansätzen wird versucht aus der bisherigen Fehlerentdeckung eine Prognose abzuleiten. Auch bei festgelegten Benutzungsprofilen ist eine derartige Extrapolation schwierig. In [8] wird für die Prognosefähigkeit dieser Ansätze angeführt, dass aufgrund vieler Faktoren „große Variabilitäten auftreten, welche nur schwer in ein mathematisches Modell zu fassen sind". Änderungen in der Zusammensetzung der Entwicklergruppe oder auch Tagesformen einzelner Personen sind solche Faktoren. Daher wird in [8] der Ansatz verfolgt, die Kritikalität einzelner Komponenten bzgl. dem Gesamtsystem abzuleiten und diese Komponenten hinsichtlich der Zuverlässigkeit genauer zu untersuchen. Daraus können dann Anforderungen und Vorgehen für die Entwicklung abgeleitet werden. Eine Prognose über den Reifegrad im Einsatz der Software wird nicht als realistisch betrachtet und auch nicht bezweckt.

Tests können bei Software nicht die Abwesenheit von Fehlern, sondern immer nur das korrekte Funktionieren für die jeweilige Eingabe oder die dabei auftretende Abweichung feststellen (vgl. [20]). Bei komplexen aus vielen Komponenten bestehenden Systemen treten zudem noch Fehler rein aus der Komponentenwechselwirkung auf. Sowohl in [25] als auch in [20] werden Reviews vor dem Testen empfohlen: „Inspections significantly increase productivity, quality and project stability." In [78] wird ausgeführt, dass „Fehler in der Software immer vorhanden sind", dass „bestimmte Eingangsvoraussetzungen zu einer Fehleraktivierung führen" und dass „die Eintrittswahrscheinlichkeit der Voraussetzungen im Allgemeinen nicht bekannt ist!". Sind derartige Schwachstellen vorhanden, so werden diese von einem potentiellen Angreifer auch ausgenutzt. Software ist in ihren Ablaufstrukturen nicht kontinuierlich und daher können aus Erfahrungen unter bestimmten Betriebsbedingungen keine allgemein-

gültigen Schlüsse gezogen werden. Das Kontinuumsprinzip gilt nicht, eine kleine (Last-)Veränderung bei den Eingangsgrößen bewirkt nicht zwangsweise nur eine kleine Änderung bei den Ausgangsgrößen, die Wirkung kann beliebig sein.

Dass Testen nicht die Lösung ist, wird in [34] ausgeführt:

> A million-line program has typically 100,000 branch instructions. […] A simple network of 16 switches has 70 possible paths from A to B, 100 switches having 184,756 possible paths and 400 switches having 1.38E+11 possible paths. A system with 100,000 switches has, for practical purposes, an infinite number of possible paths.

Damit ist ein vollständiger Test großer Software – für Systeme unter Echtzeitbedingungen grundsätzlich – nicht machbar. Sogenannte „Race Conditions", also Bedingungen, bei denen zeitliche Vorgaben verletzt oder Verklemmungen auftreten können, sind trotz intensiven Testens nicht vermeidbar.[1] Die Konsequenz ist laut [34]:

> With the test-based software quality strategy, large-scale life-critical systems will be least reliable in emergencies – and that is when reliability is most important.

Daher muss bei der Entwicklung von Software mit der geforderten Zuverlässigkeit eine konstruktive Sicht unter Berücksichtigung der Injektion von Fehlern und deren Ursachen zugrunde gelegt werden. Bei sicherheitskritischen Systemen von hoher/höchster Zuverlässigkeit – die IEC 61508 [36, 37] spricht von „high demand request" bei 24-h-Betrieb und einem Safety Integrity Level 4 (SIL4) – ist nur eine deterministische Betrachtung erlaubt.[2] Bei hochzuverlässigen Systemen wird durch die Verwendung von Redundanzmechanismen (vgl. [19]) eine Sicherstellung der Zuverlässigkeit erreicht. Bei Ausfall von Funktionskomponenten stehen Ersatzkomponenten oder redundante Komponenten zur Verfügung, welche zumindest eine reduzierte Funktionalität sicherstellen. Mechanisch ist das Zweikreis-Diagonal-Bremssystem im Automobil ein anschauliches Beispiel.

Softwarebasierte Funktionen in sogenannten eingebetteten Systemen stellen mittlerweile über ein Drittel der Wertschöpfung dar. Die in Abbildung 2 skizzierte Entwicklung bezieht sich auf Beispiele aus dem Telekommunikationsbereich, der industriellen Automation und Robotik, der Unterhaltungselektronik, der Luft- und Raumfahrt und insbesondere der Automobilindustrie und dürfte in Richtung von 40% Wertschöpfungsanteil von Software in den nächsten Jahren gehen. Damit wird Software und deren Zuverlässigkeit immer mehr zum zentralen Faktor aller Entwicklungen. Die Frage nach der Zuverlässigkeit und Qualität im Herstellungsprozess sowie in der Betriebsphase ist allerdings nicht zufriedenstellend zu beantworten.

Laut Abbildung 3 beträgt der Kostenanteil der Elektronik an den Produktherstellungskosten, also Software und Hardware zusammen, im Mittel über 50%. Die Elektronik hat im Wesentlichen einen hohen Qualitätsstandard. Allerdings verursacht nach [39] eine Innovation z. B. im Automotive-Sektor Kosten in Höhe von 70–80 Mio. €. Bei Innovationen spielt Software also eine große Rolle, wobei aufgrund der momentanen Herstellungsprozesse der Software und ihrer Fehlerproblematik deren Wertschöpfung allerdings noch kostenmäßig kompensiert wird.

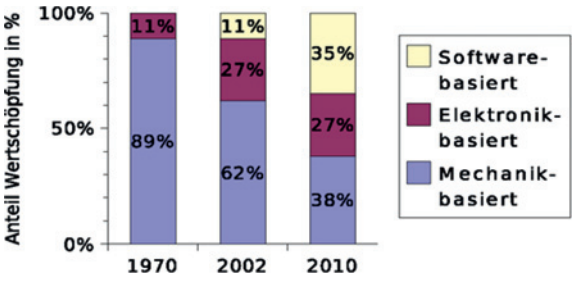

Abbildung 2: Wertschöpfungsanteil von Software (nach [64]).

Industriezweig	2003	2009*
Automobilindustrie	52%	56%
Luft- und Raumfahrt	52%	54%
Automatisierungstechnik	43%	48%
Telekommunikation	56%	58%
Unterhaltungselektronik und intelligente Häuser	60%	62%
Medizinische Geräte	50%	52%
Gewichteter Durchschnitt	51%	53%

Anmerkung: * Schätzung

Abbildung 3: Kostenanteil von Elektronik [22].

1 Beim Pathfinder-Projekt wurde das System auf der Erde intensiv getestet. Auf dem Mars ergab sich dann eine etwas andere Konstellation, die schließlich zu einer Prioritätsinversion mit unbeschränkter Blockade und in Konsequenz über eine Zeitüberwachung zu einem sich wiederholenden Neustart führte. Das implementierte System hatte einen konkurrierenden Ressourcenzugriff mit Verdrängung und nachfolgend eine unbeschränkte Blockade durch mittelpriore Prozesse implementiert. Die unbeschränkte Blockade konnte erst nachträglich durch die Implementierung des „priority inheritance protocol" (PIP) behoben werden. Das neue Zugriffsprotokoll hat das vorliegende Problem gelöst, ist aber nicht deadlockfrei. In Sprachen wie Ada [3] ist das „priority ceiling protocol" (PCP) implementiert, welches sowohl die Prioritätsinversion mit unbeschränkter Blockade als auch die Verklemmung über gemeinsame Ressourcen bei Überkreuzzugriff vermeidet.

2 Z. B. kann eine Bremse, die nur in 99,9% der Anforderungen funktioniert, nicht die Lösung sein.

Ein Grund für den hohen Kostenfaktor sind die Kosten, die zeitlich nach der eigentlichen Entwicklung bei den Softwarekomponenten in der Betriebsphase anfallen. Schätzungen beziffern den Anteil der Wartungskosten auf 50 bis 75% [17, 60, 81]. Der Grund für die hohen Kosten liegt zu einem großen Teil darin, dass Fehler aus den frühen Phasen des Softwareentwicklungsprozesses nicht zur Übersetzungszeit, sondern erst in späteren Phasen erkannt und behoben werden. Je später dabei die Korrekturen erfolgen, umso höher sind die Kosten. Typstrenge Programmiersprachen erlauben Fehler frühzeitig zu erkennen und damit Kosten zu reduzieren. Handelt es sich um logische Fehler aus der Analyse der Anforderungen (Requirements), welche mit der gewählten Architektur nicht vereinbar sind, kann das ein komplettes Projekt zum Scheitern bringen.

Abbildung 4 zeigt, wie sich die Kosten entsprechend der späteren Fehlerbeseitigung erhöhen. Die Kosten in der Requirements-Phase sind dabei auf den Wert 1 normiert. Ein Fehler, der in der Analysephase auftritt, erkannt und behoben wird, schlägt dann mit dem Wert 1 zu Buche. Wird der Fehler erst in der Betriebsphase im Rahmen einer Wartung beseitigt, so steigen die Kosten auf das über 360-fache des Normwertes. Die Konsequenz daraus ist also die Notwendigkeit, Fehler so früh wie möglich zu erkennen und zu beseitigen.

Betrachtet man die Herstellung von Software als Konstruktionsprozess, so ergibt sich die Konsequenz, die Fehlerraten frühzeitig durch entsprechende konstruktive Maßnahmen zu minimieren, also nicht entstehen zu lassen. Hinzu kommen administrative Maßnahmen, wie den wirtschaftlichen und damit den zeitlichen Druck in der Entwicklung deutlich zu reduzieren und Zeit für intensive Analysen und Plausibilitätsprüfungen einzuräumen.

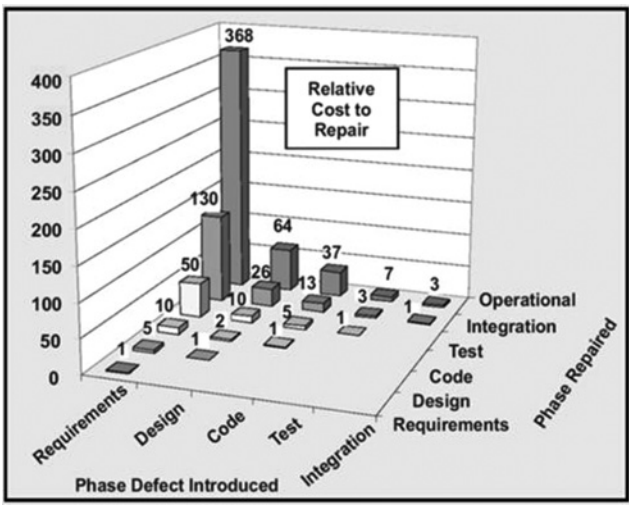

Abbildung 4: Injektionszeitpunkt von Fehler und Kosten der Behebung [15].

Eine Studie der Health and Safety Executive [33] hat die Ursachen für Fehlverhalten in automatisierten Systemen untersucht. Es handelte sich dabei um Vorfälle in unterschiedlichen technischen Systemen. Das Ergebnis war:

- „44% had inadequate specification as their primary cause
- 15% design and implementation
- 6% installation and commissioning
- 15% operation and maintenance
- 20% changes after commissioning".

Die Studie zeigt, dass „3/5 of control system failures are built-in before operation commences". Das liefert einen deutlichen Hinweis darauf, dass im Allgemeinen die Analysephase unterbewertet wird. Verschleppt sich die Beseitigung der Fehler in späte Phasen, so ist die Entstehung enormer Kosten, welche den Mehrwert durch die softwarebasierte Funktionalität reduzieren oder überkompensieren, nicht vermeidbar. Statistiken besagen (vgl. [52]), dass mit 0,5–7 Fehlern/kSLOC (kSLOC = thousand source lines of code) zu rechnen ist. Etwa 50% davon treten im ersten Einsatzjahr zutage. Eine Software für ein eingebettetes System unter Echtzeitbedingungen mit typischerweise 300 kSLOC kann also über 300 Fehler beinhalten. Die Konsequenz ist, dass es auf absehbare Zeit nicht möglich sein wird, fehlerfreie Software zu produzieren. Software ohne Fehler ist unrealistisch. Das Ziel besteht darin, robuste Software zu konstruieren, welche Fehler verlässlich (tolerant) handhaben kann. Konstruktiv sind das Vermeiden und frühe Erkennen von Fehlern im logischen Aufbau durch Analysen und bei der Implementierung durch den Compiler ein wichtiger Ansatzpunkt. Zusätzlich ist die direkte Behandlung von Fehlern zur Ausführungszeit („Exception Handling", vgl. [3]) durch das Laufzeitsystem (Typ-und Indexprüfung etc.) notwendig. Diese Forderungen sind auch in der IEC 61508 gefordert (vgl. [36, Anhang 7]).

Werden die Kosten der Softwareentwicklung über alle Phasen hinweg betrachtet, so ergeben sich Kosten in Höhe von $10–$200 pro ausgeliefertem SLOC. Beeinflussende Faktoren sind dabei die Komplexität der Anwendung und die geforderten Qualitätskriterien, die Effizienz des Entwicklungsprozesses und des Managements, die verwendeten Technologien und Werkzeuge sowie die Kompetenz der Personen. Im Rahmen des CMM (Capability Maturity Model) (vgl. [67]) erfolgt durch den „Personal Software Process" eine Intensivierung der Kompetenz der Softwareentwickler. Der resultierende Effekt ist eine deutliche Fehlerminimierung. Girish [26] folgert, dass „Personal Software

Process/CMM removed more than 75% of defects prior to test“. Als Konsequenz konstatiert er:

> „It is cheaper to remove defects earlier in the lifecycle phases than later“.

Ein wesentlicher Faktor zeigt sich in der Beherrschbarkeit der Komplexität von Software (vgl. [62]). Die Komplexität von Software kann durch das Maß „cyclomatic complexity“ beschrieben werden. Dabei werden von der Zahl der Zustandsübergänge (edges) die Zahl der Zustände (nodes) abgezogen und zwei addiert. Das Ergebnis stellt die so genannte zyklomatische Komplexität dar. Abbildung 5 zeigt typische Programmabläufe, wie sie auch aus der strukturierten Programmierung geläufig sind. Die Komplexität einfacher und übersichtlicher Strukturen, z. B. einfache Übergänge in Zustandsautomaten oder die Elemente aus der strukturierten Programmierung, liegen bei zwei.

Ebenfalls in [62] wird die Abhängigkeit des Risikos von Fehlern von der Zahl der Anweisungen und der vorhandenen zyklomatischen Komplexität einer Funktion dargestellt. Es zeigt sich, dass sowohl die Anzahl der Anweisungen als auch das Maß der zyklomatischen Komplexität gering zu halten ist, um das Risiko von Fehlern zu minimieren. „Keep it small and simple“ steht damit für eine geringe Zahl von Anweisungen und eine geringe Komplexität in der Programmstruktur. Die Beherrschung der Komplexität von Softwaresystemen erfolgt durch eine dem Problem entsprechende Architektur (z. B. Client-Server), eine adäquate Modularisierung mit Wiederverwendung allgemeiner Module, der Verwendung nebenläufiger Konzepte wie Tasks zur Strukturierung der Behandlung gleichzeitig auftretender Ereignisse mit unterschiedlichem Zeitverhalten sowie durch eine Aufteilung in mehrere unabhängige und

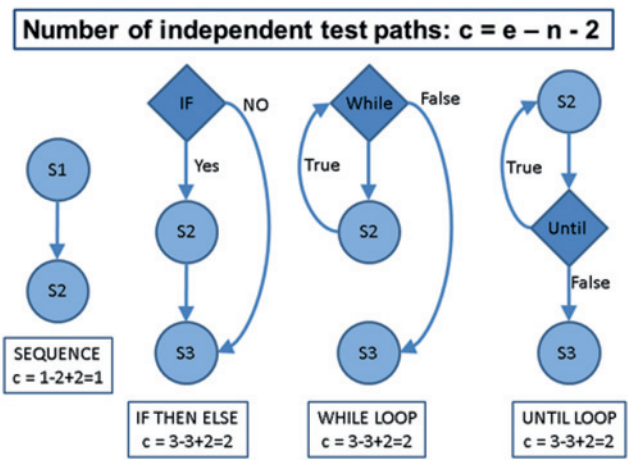

Abbildung 5: Zyklomatische Komplexität (aus [46]).

verteilte Systeme. Gesamtsysteme aus vielen verteilten Komponenten besitzen im Falle einer nichthierarchischen Wechselwirkung in ihrer Interaktion eine hohe Komplexität im Gesamtsystemverhalten.

Allgemein akzeptiert ist, dass bei der Herstellung von Software verschiedene Phasen (Schritte) existieren. Die Norm IEEE 610.12 (vgl. [61]) definiert Softwareengineering als die

> Anwendung eines systematischen, disziplinierten und quantifizierbaren Ansatzes auf die Entwicklung, den Betrieb und die Wartung von Software.

Das bedeutet die Anwendung der Prinzipien des ingenieurgemäßen Vorgehens auf die Softwareerstellung. Der von Pomberger [61] dargestellte Kreislauf der Entwicklung von Software kann als allgemein gültig betrachtet werden, da immer wieder neue Anforderungen zu berücksichtigen und zu integrieren sind (vgl. Abbildung 6). Detailliertere Vorgehensmodelle und Prozesse [71] konkretisieren dieses allgemeine Konzept mit Erweiterungen hinsichtlich der Aktivitäten einschließlich der begleitenden Maßnahmen wie Projektmanagement etc., den aktiven Personen und ihren Rollen, den spezifischen Produkten als Ergebnisse und den gegenseitigen Abhängigkeiten.

In der Analysephase erfolgt ein Requirements Engineering, um alle Anforderungen an das System zu fixieren und die Spezifikation zu erstellen. Safety-Anforderungen und Security Anforderungen bei offenen, vernetzten Systemen sind hier in den Lebenszyklus zu integrieren. Ebenfalls sind die zentralen Anforderungen zu erkennen, anhand derer eine nachhaltige Software-Architektur festgelegt werden kann.

In der Entwurfsphase werden die Objekte, Klassen und Komponenten des Systems (Software) erarbeitet. Über die Beschreibung der Interaktionen wird modelliert, wie die Anforderungen durch die Wechselwirkung von Objekten etc. realisiert werden können. Das konkrete Verhalten von Objekten wird durch Zustandsautomaten beschrieben. Die Architektur der Software ergibt sich sowohl aus den statischen Strukturen der Klassenbezüge als auch aus den gewählten grundlegenden Konzepten wie Client-Server-Architekturen und Kommunikationsstrukturen entsprechend den Anforderungen. Die Implementierung kann durch eine manuelle Codierung erfolgen, wobei zufällige Fehler durch den Menschen als Programmierer entstehen. Bei Verwendung einer automatischen Codegenerierung durch die Transformation der Beschreibungsmodelle mit Anreicherung und Prüfung durch z. B. Templates im „Model Driven Software Engineering“ (vgl. [48, 49, 55]) treten keine zufälligen Fehler auf. Vorhandene Fehler in der Codegenerierung sind systematischer Natur

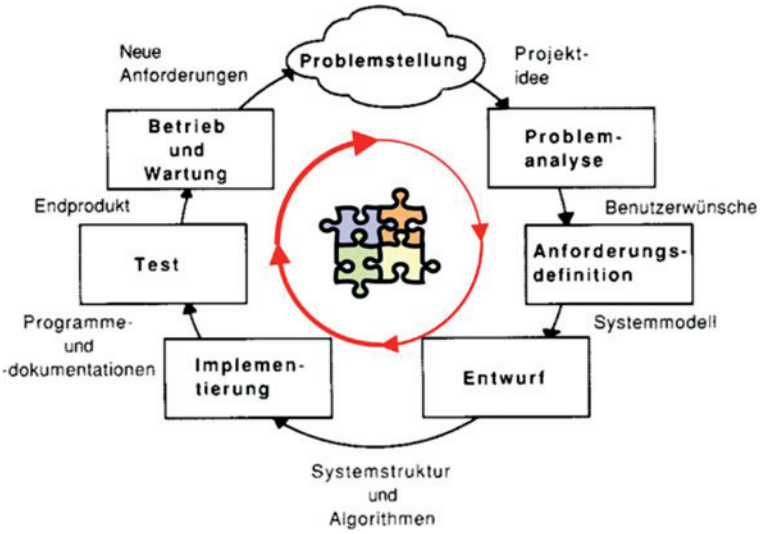

Abbildung 6: Life Cycle in der Softwareentwicklung (nach [61]).

und daher zielgerichtet zu erkennen. Logische Fehler sind in der Analysephase über die Darstellung mit UML zu vermeiden. Die Test- und Integrationsphase dient zum Testen der Komponenten sowie des Gesamtsystems. Dazu sind die Komponenten entsprechend zu integrieren und auf die Zielplattform zu bringen. Modellgetriebene Tests verwenden die in den Klassen definierten Attribute zur Generierung automatischer Tests. Funktionaltests und zeitlogische Tests kommunizierender Prozesse können allerdings keine Fehlerfreiheit liefern. Die Wahl der Prioritäten von nebenläufigen Elementen (Tasks) und des Schedulingverfahrens als Grundlage einer mathematischen Beweisführung muss in der Entwurfsphase erfolgen.

Zu den vorher genannten Phasen kommen noch begleitend die Versions- und Variantenverwaltung sowie das Projektmanagement hinzu. Dabei ist zu beachten, dass Personen mit ihren Kompetenzen und in ihren jeweiligen Rollen berücksichtigt werden müssen. In allen Prozessen und bei allen eingesetzten Methoden steht der Mensch im Mittelpunkt, besonders in der Analyse- und der Designphase. Insbesondere die Begriffswelt der Teammitglieder divergiert oft in deren Bedeutungsvorstellungen. Ein Begriffslexikon zur standardisierten Festlegung der Bedeutungen ist daher unerlässlich. Ebenso ist eine bildhafte Repräsentation zum besseren gemeinsamen Verständnis von Strukturen z. B. durch die UML [73] hilfreich.

Die im Lebenszyklus enthaltenen Phasen als zeitlich gegliederte Ablaufschritte können in einer Zeitlinie hintereinander, überlappend oder mehrfach wiederholend miteinander verknüpft auftreten (vgl. [71]). Vorgehensmodelle strukturieren diese Phasen in ihrem zeitlichen und gegenseitigen Bezug und können sequenziell oder iterativ sein. Ein Prozessmodell ist die Summe aus einem Vorgehensmodell, der Festlegung von Notationen, Methoden und Werkzeugen und der Definition von Rollen (Zuständigkeiten/Verantwortlichkeiten) und Produkten. Weiterhin sind begleitende Aktivitäten im Modell wie Qualitätssicherung einschl. Prozessverbesserung, Konfigurationsmanagement sowie Projektmanagement beinhaltet. Ein Prozessmodell beschreibt also den gesamten Prozess der Softwareentwicklung im „life cycle“ und der Produktion.

Bei der Herstellung von Software sind bestimmte Qualitätsmerkmale wie funktionale Korrektheit, Zuverlässigkeit (bzgl. Falscheingaben, Störungen) etc. zu erfüllen. Die geforderte Softwarequalität ist primär durch konstruktive Maßnahmen zu sichern. Durch Nachweisverfahren auf Quelltextebene oder durch Testen kann das unterstützt werden. Hinzu kommen begleitende (organisatorische) Maßnahmen, um die Qualität des Herstellungsprozesses zu sichern. Metriken können zum Messen von Software-Eigenschaftsgrößen verwendet werden. Die ISO 15939 [38] definiert hierzu ein Gerüst zur Softwaremessung im Softwareengineering-Prozess. Es ist ein internationaler Standard, der einen Softwaremessprozess festlegt, der auf alle softwarebezogenen und Managementdisziplinen anwendbar ist. Metriken dienen dazu, einen Prozess durch Messung und Berechnung geeigneter Größen transparenter zu machen. Dadurch kann der Zustand eines Prozesses allgemein dargestellt und beobachtet werden, es können Schwachstellen aufgedeckt und in der Folge beseitigt werden, und es können Prozessabläufe aufgrund von Erkenntnissen aus den Metrikauswertungen kontinuierlich verbessert werden.

Die internationale Norm ISO 9000 ist ein Standard im Bereich der Qualitätsmanagementsysteme. Innerhalb der ISO 9000 kann die ISO 9002 auf Software angewendet werden (vgl. [41]). Die ISO 9001 zielt auf das Management ab, alle Prozesse mit Qualitätsbezug sind unter Kontrolle,

alles hierzu wird dokumentiert und die Zertifizierung erfolgt durch externe Auditoren.

Aus den USA steht das CMM – Capability Maturity Model (Tauglichkeitsreifemodell) für Software bzw. das aktuelle Capability Maturity Model Integrated (CMMI) für die Systementwicklung dagegen. CMM wurde vom SEI, Pittsburgh (USA), als Antwort auf die ISO 9001 entwickelt. CMM geht von 5 Stufen der Reife eines Prozesses aus. CMM ist ausschließlich für Software entwickelt worden (ISO 9000 für Industrie allgemein) und ist sehr detailliert. CMM dient zur Einschätzung der Qualität, während die ISO 9001 die Organisation festlegt. Die Skala reicht bei CMM von 1 (schlecht) bis 5 (sehr gut). ISO 9001 entspricht etwa CMM 2–3. Die Erweiterung CMMI (Capability Maturity Model Integrated) deckt den gesamten Systementwicklungsprozess ab (vgl. [67]).

Agile Prozesse wie Extreme Programming [6] adressieren die Architektur nicht direkt und die Zielarchitektur ergibt sich eher implizit. Einzelne Methoden der agilen Vorgehensweise sind positiv und unterstützen das konstruktive Erreichen einer hohen Zuverlässigkeit. Der Systemtest z. B., bei dem immer alle bisherigen Test und für neue Komponenten zusätzlich definierte Tests ausgeführt werden, adressiert die Problematik der Integration von Komponenten (vgl. [68]). Erst nach Bestehen dieses umfangreichen Systemtests wird eine Software frei gegeben. Die agilen Prozesse vermeiden allerdings eine umfassende Analyse der Anforderungen zu Beginn eines Projektes. Sie gehen von einer permanenten sukzessiven Umgestaltung der sich ergebenden Architektur des Softwaresystems aus. Die permanente Umgestaltung einer Softwaresystemarchitektur mit anschließendem vollständigem Systemtest ist im Gegensatz zu vorher geplanten Verbesserungen oder Erweiterungen im Sinne Wartung bei großen Softwaresystemen wie in der Automatisierung unrealistisch[3]. Erst bei Festlegung einer adäquaten Architektur lassen sich zukünftige Anforderungen ohne erheblichen strukturellen Aufwand integrieren. Die sehr umfangreich definierten Prozesse wie das V-Modell, dessen Erweiterung zum V-Modell XT oder auch das Cleanroom-Konzept erfordern hohe Ressourcen und werden wiederum der Erkenntnis, dass ein Gesamtsystem nicht aus einem Guss entworfen werden kann, nicht gerecht. Eine detaillierte Analyse ist unumgänglich, es kann aber nicht davon ausgegangen werden, dass zu Beginn der Entwicklung eine vollständige Spezifikation erreicht werden kann. Komplexe Entwicklungsprozesse gehen von einer vollständigen Spezifikation des zu entwickelnden Softwaresystems aus und erfordern umfangreiche Ressourcen. Eine vollständige Systemspezifikation zu Beginn eines Projektes ist aber meist nicht möglich.

Damit bleibt der Ansatz, dass zu Beginn einer Entwicklung die Anforderungen genau analysiert werden müssen, jedoch noch keine vollständige Spezifikation erreicht werden kann. Die Entwicklung von softwarebasierten Funktionen erfordert daher ein iteratives, inkrementelles und risikobezogenes Vorgehen. Dies konzentriert sich auf die Kernanforderungen zur Ableitung einer nachhaltigen Architektur und stellt damit eine konsistente Erweiterung sicher. Safety- und Security-Aspekte sind integrativ in der Analysephase zu berücksichtigen. Das Einspielen von Updates zur Beseitigung von Fehlern oder Schwachstellen ist bei Automatisierungssystemen meist nur im Rahmen größerer Wartungsvorgänge möglich. Zwischenzeitliche Änderungen in der Software beziehen sich in der Regel auf Parametereinstellungen (vgl. hierzu [11]).

3 Echtzeitanforderungen

Die Automatisierung technischer Systeme basiert auf der Datenverarbeitung unter zeitlichen Bedingungen (vgl. [7, 51, 54]). Eine Berechnung ist nur dann korrekt, wenn sie in der Funktion und im geforderten Zeitpunkt korrekt ist. Die Definition eines Echtzeitsystems nach DIN 44300 lautet: „Echtzeitbetrieb ist der Betrieb eines Rechnersystems, bei dem Programme ständig betriebsbereit sind und Verarbeitungsergebnisse innerhalb vorgegebener Zeitspannen verfügbar sind“.

In der Automatisierung werden vom technischen Prozess durch Sensoren Messwerte erfasst, die Rechenvorschriften berechnen notwendige Stellgrößen, die als Prozesseingriffe über Aktoren an den Prozess gehen. Das Schema ist sowohl für einen Prozessrechner wie auch für ein Embedded-System gleich. Die Online-Kopplung bedeutet die direkte messtechnische Verbindung des Rechners mit dem technischen Prozess über Sensoren. Die Closed-Loop-Kopplung bedeutet den direkten Eingriff über Aktoren. Damit ist die Online-/Closed-Loop-Kopplung die engste Interaktionsart, das Ziel ist also die selbsttätige Regelung. Die Automatisierung technischer Systeme ist bei 16 Mrd. eingebetteter Systeme weltweit der intensivste Rechnereinsatz und entsprach 2006 ca. 90% und neuerdings bis zu 98% aller eingesetzten Prozessoren (vgl. [1, 4]).

3 Das Verwenden festgelegter Software-Architekturen bedeutet dabei nicht, dass die Architektur des zugrundeliegenden technischen Systems fix sein muss. Vielmehr muss die festzulegende Software-Architektur die Flexibilität des technischen Systems hinsichtlich zukünftiger Anforderungen von Anfang an berücksichtigen.

Hinsichtlich der zeitlichen Randbedingungen wird zwischen harten und weichen Echtzeitbedingungen unterschieden (siehe z. B. [13]):

- harte Zeitbedingungen sind absolute Zeitbedingungen. Die Verletzung der absoluten Zeitbedingungen führt zu katastrophalen Schäden. Beispiele für Systeme mit harten Zeitbedingungen sind das ABS (Antiblockiersystem), ESP (Elektronisches Stabilisierungssystem) oder das Landesystem bei Flugzeugen.
- weiche Zeitbedingungen sind eine Sollvorgabe. Die Verletzung von weichen Zeitbedingungen führt zu Komforteinbußen. Beispielkonsequenzen sind schlechter Klang beim Telefon etc.

Zur Behandlung von Ereignissen unter Echtzeitbedingungen bestehen folgende prinzipiellen Lösungsansätze (vgl. [47]):

1. Polling – zyklisches Abfragen
2. zeitgesteuert – Timer Interrupt
3. interruptgesteuert (IR) – externes Ereignis
4. synchroner Ablauf/Cyclic Executive – Schleife
5. asynchroner Ablauf/Prozesskonzept (Ereignisorientierung, Ereignis als Zeitfortschaltung)

Die Beherrschung des „worst case“ und eine hohe Zuverlässigkeit in der Funktionsausführung bei gleichzeitig wirtschaftlich vertretbarem Aufwand und Konsistenz gemeinsamer Daten erfordern beim Polling und zeitgesteuertem Ansatz eine Rechenleistung derart, dass alle Funktionen innerhalb der kleinsten Zykluszeit berechnet werden müssen. Das ist wirtschaftlich nicht vertretbar. Der IR-basierte Ansatz kommt mit einer adäquaten Rechenleistung aus, ist aber bei Änderungen aufgrund der IR-Vektoradresszuordnungen fehleranfällig und kann die Konsistenz gemeinsamer Daten aufgrund der jederzeitigen Unterbrechbarkeit durch höher priore Interrupts nicht sicherstellen. Der Cyclic-Executive-Ansatz ist bei Änderungen ähnlich fehleranfällig wie der IR-basierte Ansatz. Hinzu kommt, dass asynchrone Ereignisse konzeptionell nur im Rahmen des vorgesehenen Ablaufs bearbeitet werden können. Eine Erweiterung mit IR-Routinen führt zur Dateninkonsistenz. Die Verzahnung der Funktionsausführung mit der Verwaltung mindert die angestrebte Zuverlässigkeit ebenfalls.

Eine hohe Zuverlässigkeit bei softwarebasierter Funktionalität unter Echtzeitbedingungen wird erreicht, wenn eine strikte Trennung der

- Ausführung der Funktionen und der
- Verwaltung der Funktionsausführung realisiert wird (vgl. [69, 72]).

Dabei muss die Isolierung der Funktionsausführungen untereinander als gegenseitige Sicherheit gewährleistet werden. Fehler in einer Ausführung dürfen andere Funktionen durch z. B. falsche Speicheradressierungen etc. nicht beeinflussen. Das Prozesskonzept der Automatisierungstechnik ist hierzu ein geeignetes Prinzip (vgl. [51, 69, 72]): Jegliche Funktion, die als Berechnungsvorgang einem konkreten technischen Vorgang (Prozess) gegenübersteht, wird als eigenständiger Rechenprozess (oder auch Task genannt) realisiert. Dieser berechnet die Funktion korrekt und liefert das Ergebnis zeitgerecht. Die korrekte Ausführbarkeit aller Berechnungen unter allen zeitlichen Randbedingungen wird vor der Ausführung bewiesen. In Teilen der Literatur wird beim asynchronen Konzept Kritik an der vorhandenen Unplanbarkeit bei der Ausführung im Sinne „nur näherungsweise Erfüllung der Rechtzeitigkeit“ (vgl. [51]) geäußert. Die Unplanbarkeit in der tatsächlichen Ausführungsreihenfolge resultiert aber aus dem zufälligen Auftreten der externen Ereignisse als Stimulus (vgl. [7]). Mit dem im Jahr 1973 publizierten Rate Monotonic Analyse und Scheduling Verfahren (RMA/RMS) von Liu/Leyland [53] kann die Ausführbarkeit unter Einhaltung der geforderten zeitlichen Randbedingungen für den worst case Fall bewiesen werden.

Beim Prozesskonzept erfolgt eine standardisierte Verwaltung nach abstrahierenden Kriterien wie Wichtigkeit und Dringlichkeit mit einer prioritätenbasierten, verdrängenden Ausführung. Der Prozess wird als Ausführungseinheit auf der CPU betrachtet, sein Ausführungskontext (Zustand) wird als virtuelle CPU (vCPU) bezeichnet. Bei einer Unterbrechung der Ausführung wird der aktuelle Prozesszustand gesichert. Die Fortführung erfolgt beliebig durch Laden des vorherigen Zustands (z. B. Register). Prozesse besitzen Prioritäten, welche die Rangfolge der rechenbereiten Prozesse festlegen. Die Ausführbarkeit unter allen möglichen Bedingungen wird bewiesen, z. B. mittels Response-Time-Analyse (RTA, vgl. [5, 13]). Das Response Time Analyse Verfahren (RTA) garantiert bei prioritätsbasiertem, verdrängendem Scheduling für „worst case“ Szenarien die Einhaltung der zeitlichen Randbedingungen und ist optimal.

Bei Echtzeitsystemen erfolgt das Scheduling verdrängend (preemptive) mit Prioritäten zum Beispiel nach dem Rate Monotonic, dem Least Laxity First oder dem Earliest Deadline First Verfahren (vgl. [13]). Prozesse (Tasks) können autonom oder in Arbeitsteilung ablaufen und kooperieren (vgl. [31]). Kooperation erfordert eine Kommunikation, die auf Basis gemeinsamer Daten (Speicherbereiche) oder durch Nachrichten (geschützter Transportbereich) erfolgen kann. Wichtig ist das Sicherstellen der Datenkonsistenz bei Schreib- und Lesezugriffen (Synchro-

nisation), da sonst Inkonsistenzen in den Daten entstehen und Fehler in der Berechnung auftreten. Der Ansatz mit Semaphoren ist nicht fehlervermeidend und daher kritisch hinsichtlich einer zu erzielenden Zuverlässigkeit. Die Verwendung von Monitoren kann direkt über Sprachkonstrukte mit einer compilergestützten Implementierung des Monitorkonzepts (protected, synchronized) erfolgen. Damit werden Programmierfehler zur Synchronisation vermieden. Das Nachrichtenkonzept hat prinzipiell keine gemeinsamen Daten, erfordert aber eine Transportschicht und erzeugt einen nicht zu unterschätzenden Kopieraufwand (von Sender zum Empfänger). Für große Datenmengen, wie z. B. in der Verarbeitung von Bilddaten unter Echtzeitbedingungen, muss deshalb genau analysiert werden, welche Lösung die sinnvollere ist.

4 Programmierung von Echtzeitsystemen

Die programmtechnische Realisierung von softwarebasierten Funktionen zur Automatisierung technischer Systeme unter Echtzeitbedingungen muss der Zuverlässigkeitsanforderung genügen. Hierzu hat eine Programmiersprache entsprechende Sprachelemente semantisch exakt zu definieren, syntaktisch in ihren Strukturen fehlervermeidend und fehlererkennend festzulegen, mathematisch exakt die Umsetzung von algorithmischen Typstrukturen zu leisten sowie Architekturen mit klaren Beziehungen zwischen den Elementen und modularen hierarchischen Bezügen zu unterstützen. Außerdem sind Sprachmittel für die Definition von Nebenläufigkeit und der Kommunikation von Prozessen bereitzustellen und damit alle Aspekte der Realisierung von Echtzeitsoftware zu unterstützen (vgl. [82, S. 234], [65, S. 157], [44]).

Der Compiler sollte im Rahmen der Übersetzung umfangreiche Prüfungen leisten und auch das Laufzeitsystem muss jegliche Anweisungen überwachen können. Zuweisungen, welche den zulässigen Wertebereich einer Typdefinition verlassen, müssen erkannt und gemeldet werden. Eine Division durch Null oder Konvertierungsfehler müssen entdeckt werden.

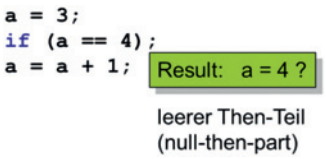

Abbildung 7: Syntaktischer Fehler in der IF-Anweisung.

Ada:

```
a := 3;
if a = 4 then
      a := a + 1;
[else
      statement;]
end if;
```

Abbildung 8: Syntaktische Struktur der If-Anweisung von Ada [47].

Die Fehlervermeidung einer Programmiersprache wird nachhaltig durch ihre syntaktische Fassung und den verfügbaren semantischen Informationen festgelegt [56, 79]. In der Sprache C bzw. C++ sind die syntaktischen Fassungen von Sprachelementen fehleranfällig. In Abbildung 7 ist die syntaktische Fassung der bedingten Anweisung der Sprache C/C++ nicht fehlervermeidend. Es kann ein leerer Anweisungsteil in der bedingten Anweisung auftreten, falls z. B. durch Kopieren und Einsetzen versehentlich ein Semikolon als Trennzeichen verwendet wird. Damit wird aber die nachfolgende Anweisung unbedingt ausgeführt, der Vergleich hat keine Wirkung. Der Fehler wird durch den Compiler nicht erkannt.[4]

In syntaktisch und semantisch eindeutig definierten Sprachen wie Ada ist das beispielsweise nicht möglich (vgl. [3] und Abbildung 8), da eine Zuweisung direkt nach if und ein null-then-part nicht zulässig sind. Ada bietet also eine syntaktische Fassung mit einer hohen Fehlererkennung durch eine Fehler-Distanz größer zwei.

Auch in der Spezifikation der Repräsentation von Typen im Speicher ist C/C++ fehleranfällig. Bei der Repräsentation von Aufzählungen besteht in C/C++ die Gefahr, dass mehrere Elemente mit der gleichen Darstellung aufgeführt werden (siehe Abbildung 9) und damit ein Nichtdeterminismus und Fehler eingeführt werden. Durch die vorgegebene Standardabbildung auf bestimmte ganze Zahlen und die nachfolgende Veränderung einzelner Werte durch den Programmierer können inkonsistente Wertebelegungen auftreten. In der Sprache Ada ist auch das beispielsweise nicht möglich, da der Compiler die Gesamtrepräsentation prüft (vgl. [3]).

Auch die Ergänzung der Sprache C/C++ durch umfangreiche Regeln und vorgelagerte Prüfwerkzeuge kann die Defizite nicht vollständig kompensieren. Entweder schränken syntaktische Regeln die Konstrukte einer

4 Die Fehlerdistanz zwischen zwei als syntaktisch korrekt erkannten Strukturelementen ist zu gering, in wichtigen Sprachelementen Null. Ein Fehler führt zu einer korrekten Darstellung und wird als solcher nicht erkannt.

Aufzählungstypen in C (set of named integer constant values):

```
enum abc {A,B,C,D,E,F,G,H} var_abc;
```

The values of the contents of abc would be A=0, B=1, C=2, etc.

C allows values to be assigned to the enumerated type as follows:

```
enum abc {A,B,C=6,D,E,F=7,G,H} var_abc;
```

This would result in: A=0, B=1, C=6, D=7, E=8, F=7, G=8, H=9

Abbildung 9: Inkonsistente Repräsentation von Aufzählungswerten.

Sprache massiv bis hin zur Nichtbenutzbarkeit ein oder es bleiben doch kritische Freiheitsgrade übrig. Der Umfang der statischen Prüfbarkeit durch den Compiler hängt direkt von der Verfügbarkeit der semantischen Informationen unabhängig von eingehaltenen Regeln ab. Der Quelltext wird mit allen Informationen in eine interne Repräsentation (typisierter abstrakter Syntaxbaum, Datenflussgraph und Kontrollflussgraph) überführt und kann dann statisch analysiert werden. In C/C++ sind auch bei Hinzunahme von Regelwerken Defizite vorhanden, die eine adäquate Analyse wie in Ada nicht zulassen (vgl. [50]). Hinzu kommt, dass die Zuverlässigkeit des Laufzeitsystems durch Regeln allein nicht erhöht werden kann.

Eine formale Verifikation eines Programms gegen seine Spezifikation bestätigt deren Einhaltung. Zusammen mit einem Funktionaltest wird die intendierte Zielfunktion überprüft. Zusätzlich erforderlich ist allerdings eine Ergänzung durch die Prüfung dynamischer Datenwerte, um die Ausschließlichkeit der Spezifikation zu garantieren und ungewolltes Verhalten, z. B. durch Angriffe bei vorhandenen Schwachstellen [66], zu vermeiden.

Mit dem Ansatz der Model-Driven-Architecture kann eine UML-Modell-basierte Codegenerierung erfolgen. Ein Einsatz der MDA-Vorgehensweise erfolgte 1999 beim INSPECT-pro-control-System (vgl. [24]), dessen Erstinstallation Ende 2000 erfolgte. Das INSPECT-System ist ein Werkzeug für verteilte Echtzeitanwendungen im 24-h-Betrieb in verfahrenstechnischen Anlagen. Der Einsatz erfolgt mittlerweile weltweit in über 100 Installationen. Durch den MDA-Ansatz erfolgte die Generierung der Komponenten und der Ablaufsteuerung in Form von nebenläufigen Prozessen (Tasks) und deren Kommunikation. Es handelt sich um ein verteiltes System mit hoher Zuverlässigkeit und zeitlichen Randbedingungen bestehend aus ca. 350 000 Zeilen Ada und etwa 40 000 Zeilen Java mit zusätzlich 10 000 Zeilen C++ Code. Für die Codegenerierung wurde ein UML-Profil entwickelt, das über Stereotypes und Tagged Types die Abbildung steuert (vgl. [48] und [49, 55]). Durch den Einsatz des Laufzeitsystems von Ada wurden erhebliche Fehler in den extern generierten und eingebundenen C-Funktionen erkannt. Im Ada-Kernsystem wurden etwa 5 Fehler behoben, u. a. waren Fehler im Generator, welche zu systematischen Fehlern geführt hatten und leicht zu detektieren waren. Andere Fehler waren bei der manuellen Modifikation nicht generierter Teile aufgetreten. Es wurde über 70% Code für die Struktur und das Verhalten des Programms generiert. Die Vorgehensweise hat eine Einsparung von etwa 70% über einen 10-jährigen Zeitaufwand bei extremer Fehlerminimierung um den Faktor 50 erbracht.

Trotz MDA-Ansatz darf die Rolle von Programmiersprachen nicht unterschätzt werden. Zum einen werden Programmiersprachen als Abbildungsziel benötigt, d. h., die entsprechenden Sprachkonstrukte müssen syntaktisch vorhanden und semantisch exakt definiert sein (z. B. Tasks). Zum anderen wird für die Programmausführung ein Laufzeitsystem mit entsprechenden Eigenschaften benötigt (dynamische Prüfungen). Die semantisch vollständige und statische Typisierung einer Programmiersprache wirkt sich dabei über die Typinformationen auch zur Laufzeit aus, denn nur dann kann auf Bereichsverletzungen, Konvertierungsfehler, Speicherfehladressierungen, Division durch Null, falsche Dereferenzierungen etc. geprüft und Verletzungen erkannt und behandelt werden. Wie bereits erwähnt sind die Typinformationen auch für die statische Analyse relevant, um Fehler frühzeitig zu erkennen.

5 Integrative Betrachtung von technischer Sicherheit und Informationssicherheit

Bei software-/computergesteuerten Systemen ist Sicherheit zweiseitig zu begreifen, einmal als die Sicherheit des zur Automatisierung eingesetzten Systems und einmal als die Sicherheit der Umgebung des Systems. Ersteres wird meist mit dem englischen Wort Security und Letzteres mit dem Wort Safety adressiert. Hier werden die Begriffe Informationssicherheit und technische Sicherheit verwendet. Informationssicherheit wird nach [23] umfassend und vollständig auf den Datenraum bezogen, die technische Sicherheit auf den Ausfall von Systemkomponenten und deren Auswirkungen auf die Umgebung. Sicherheit als zweiseitige Eigenschaft ist überlebenswichtig und ist integrativ zu betrachten. Die Öffnung von bisher abgeschlossenen Automatisierungssystemen im sicherheitskritischen Bereich verbunden mit der Problematik fehlerhafter und unzuverlässiger Software erfordert eine stringente Betrachtung der Sicherheitsaspekte (vgl. [11, 12, 75]).

Die Grundlage der technischen Sicherheit ist die Zuverlässigkeit des eingesetzten Systems. Die Zuverlässigkeit kann durch Redundanzmechanismen erhöht werden. Im mechanischen Bereich sind es einfache Redundanzen (zwei Seile beim Aufzug, zwei Bremskreise im PKW). Im Hardware-Bereich sind es doppelte Redundanzen, da bei einem Fehler bei zwei Systemen und einem zugrunde liegenden Ausfall nicht erkannt werden kann, welches System korrekt funktioniert. Sogenannte Voting-Mechanismen, bei denen über eine zwei von drei Auswertungen eine (Mehrheits-)Entscheidung getroffen wird, ermöglichen es auch hier, Redundanzkonzepte zu realisieren. Bei softwarebasierten Funktionen werden die Algorithmen diversitär, also mit unterschiedlichen Verfahren und von unterschiedlich ausgebildeten Personen implementiert.

Ein System ist (vgl. [77])

- deterministisch sicher, wenn
  - durch eindeutige Wirkungsbeziehungen und eindeutig berechenbare Kausalitäten ein hinreichender Gefährdungsradius berechnet werden kann und
  - ein zusätzlicher Sicherheitsabstand nicht notwendige Reserven bereitstellt.
- probabilistisch sicher, wenn
  - aufgrund der Komplexität in den Wechselwirkungen keine deterministische Sicherheit abgeleitet werden kann, oder
  - eine deterministische Sicherheit wirtschaftlich nicht vertretbar und
  - auf jeden Fall das verbleibende Risiko akzeptabel ist.

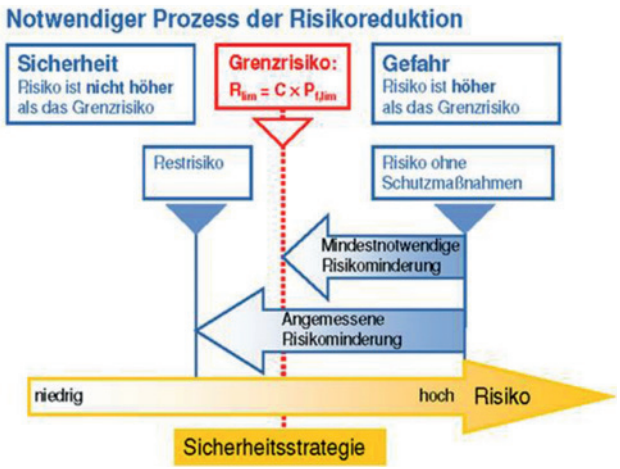

Abbildung 10: Risikobegriff [77].

Mit jeder Nutzung eines technischen Systems geht ein Risiko einher. Grundlage der Betrachtung ist der Risikobegriff, die Beherrschung des Risikos und geeignete Maßnahmen hierzu (vgl. Abbildung 10). Unter dem Begriff Risiko wird die Wahrscheinlichkeit des Eintretens eines Ereignisses verbunden mit dessen Schadensmaß verstanden. Falls das Risiko ohne zusätzliche Maßnahmen höher als das Grenzrisiko ist, so ist eine Risikoreduktion durch entsprechende Maßnahmen durchzuführen. Ziel ist es, das verbleibende Risiko unter eine Toleranzschwelle zu bringen. Die Toleranzschwelle wird als das tolerierbare Risiko bezeichnet. Es gibt also keine absolute Freiheit von Gefahr, sondern nur ein toleriertes Risiko.

Die IEC 61508 definiert einen Prozess zur Realisierung von Sicherheitseigenschaften im Sinne der Safety (vgl. [36]). In der Norm wird ein Safety Life Cycle eingeführt. Sogenannte Safety Integrity Levels (SIL) legen die Anforderungen an Sicherheit über maximale Eintrittswahrscheinlichkeiten fest. Dabei wird zwischen Systemen mit High Demand und Systemen mit Low Demand unterschieden. High Demand bedeutet dabei eine hohe Nutzungszahl, Low Demand eine geringe Nutzungszahl. Es gibt nach der IEC 61508 verschiedene SIL-Stufen, beginnend mit SIL 0 ohne Sicherheitsanforderungen bis zur Stufe SIL 4 mit höchster Anforderung aufgrund einer Lebensgefahr für Personen.

Im High-Demand-Bereich, z. B. beim ABS, gilt:

- auf der höchsten Stufe SIL 4 eine maximal zulässige Eintrittswahrscheinlichkeit von 10^{-9}/h und z. B. als Vergleich dazu
- auf der ersten Stufe SIL 1 eine maximal zulässige Eintrittswahrscheinlichkeit von 10^{-6}/h

Die Werte bei Low Demand errechnen sich zum gleichen Wahrscheinlichkeitswert bezogen auf den gleichen Zeitraum, aber mit unterschiedlichem Anforderungsprofil.

In Anhang 7 der IEC 61508 [36] wurden auch Anforderungen an Programmiersprachen festgelegt wie „strenge Prüfungen, um sicherzustellen, dass der korrekte Typ verwendet wird“, oder „analysierbare Kontrollstrukturen mit expliziter syntaktischer Klammerung“.

Der ZVEI nennt die „Bedrohung offener Automatisierungssysteme“ als eines der zentralen Handlungsfelder bzw. als Herausforderung der Zukunft [83]. Sicherheitslücken bzw. Schwachstellen finden sich bei allen Automatisierungssystemen und Sicherheitseinrichtungen (siehe [35, 74]). Aber auch bei reinen Wartungsarbeiten an Software, wie der der Tausch von Komponenten, sind Aspekte wie Integrität, Authentisierung, Autorisierung und Zurechenbarkeit wichtig. Hierdurch werden auch Safety-Aspekte beeinflusst.

Offene, über das Internet vernetzte Automatisierungssysteme müssen sicher gegenüber Angriffen aus dem Internet sein [9, 11, 45, 57, 59] oder durch geeignete Zusatzgeräte wie Security-Router etc. geschützt werden. Normen wie das ICS Kompendium des BSI [11] oder die IEC62443 [84], konzentrieren sich auf die Absicherung in den Zugängen zu den Systemen und mit der Architektur der Kommunikationsinfrastruktur und deren Härtung. Analog dem Safety Integrity Level (SIL) wird ein Security Assurance Level (SAL) eingeführt. Die im Angreifer Modell berücksichtigte Kompetenz des Angreifers ist im Sinne einer worst case Betrachtung fraglich.

Allerdings besitzen sowohl die existierenden Automatisierungssysteme als auch Sicherheitssysteme erhebliche Schwachstellen in ihrer Realisierung [74]. Teilweise sind dies Schwachstellen aufgrund der verwendeten Programmiersprache [56], teilweise auch schlichte Implementierungsfehler (z. B. Passwort und Benutzername unveränderlich in die Hardware einprogrammiert). Die Implementierungsschwachstellen liegen darin, dass die spezifizierten Typeigenschaften nicht für Laufzeitprüfungen zur Verfügung stehen. Wertezuweisungen, die länger als die ursprünglichen Werte sind, überschreiben nicht nur nachfolgende Datenbereiche sondern auch beliebige Adressbereiche z. B. beim Rücksprung aus einem Unterprogramm. Damit wird bei Unterprogrammaufrufen die Rücksprungadresse manipuliert und andere Programmbereiche angesprungen (vgl. [80]). In Konsequenz kann der Angreifer die Kontrolle über das attackierte System übernehmen. Die überwiegende Anzahl der Security Schwachstellen können laut [80] auf diese Implementierungsschwachstellen zurückgeführt werden. Die genannten Schwachstellen betreffen nicht nur Anwendungssysteme, sondern auch Betriebssysteme und gerade auch Security-Infrastruktursysteme wie Security-Router, virtuelle Maschinen etc.

Das US CERT – Computer Emergency Response Team der USA [74] und das ICS-CERT – Industrial Control Systems Cyber Emergency Response Team [35] stellen wöchentlich einen Überblick über entdeckte Schwachstellen in Software aller Bereiche zur Verfügung. Meldungen wie zu Juniper Junos (eine Software zum Aufbau von VPN Verbindungen), HP Network Manager (eine Software zur Zugangskontrolle), Cisco Sicherheits-Software, Windturbinen-Betriebssysteme, Cisco Security Verbindungs-Software, Virtuelle Maschine VMware, Web Administrations Software, Infusionspumpensysteme, Securityinterface von Panasonic und Solarspeichersystemen stehen stellvertretend für nahezu alle eingesetzte Software im Automatisierungsbereich.

Allen Schwachstellen ist gemein, dass überwiegend die Implementierung und dabei die fehlende Typprüfung das Problem darstellt. Kritisch ist das immer wiederkehrende Auftreten gleichartiger Schwachstellen, das auf grundlegende Probleme dieser Programme hinweist: Sie sind prinzipiell mit Schwachstellen aufgrund der Implementierung versehen und diese werden nur selektiv nach Bekanntwerden behoben. Die Verwendung typstrenger Sprachen wie beispielsweise Ada würde viele dieser Schwachstellen von Grund auf vermeiden.

Safety- und Security-Anforderungen sind bei zukünftigen Entwicklungen in der Automatisierung integriert umzusetzen und müssen unter dem Aspekt langlebiger Automatisierungssoftware gerade auch für das Thema „Industrie 4.0“, Smart Energy Grids oder Cyber-Physical Systems integrativ (vgl. [57]) behandelt werden. Neben der allgemeinen Securitybetrachtung (Zugangskontrollen, Abschottung etc.) hat im Entwicklungsprozess zusammen mit der Safety-Analyse auch eine Security-Analyse zu erfolgen. Je nach Sicherheitsanforderung resultieren hier unterschiedliche Anforderungen und Festlegungen für Verschlüsselung, Schlüsselzahl und Gültigkeitsdauer, Identity-Management und Security-Infrastruktur des Systems und für einzelne SW-Komponenten bzw. -Funktionen. Die Komplexität der funktionalen Vernetzung hat auch Auswirkungen auf die Securityproblematik. Je vermaschter funktionale Beziehungen sind, desto höher sind die Anforderungen an ein Securitymanagement aufgrund der vielen Querbeziehungen, welche alle abzusichern sind. Grundsätzlich reduzieren hierarchische Strukturen die Systemkomplexität und sind damit überschaubarer (vgl. [20]). Secure by Default beschreibt die Anforderung, dass Funktionen und Vorgehensweisen auf dem Stand der Technik der IT-Security für die Automation standardmäßig in den Komponenten und Systemen der Automatisierungslösungen verfügbar sind. Secure by Design geht einen Schritt weiter und fordert, dass Konzepte und Funktionen der IT-Security für die Automation bei der Formulierung der Anforderungen bis hin zur Entwicklung von automationstechnischen Komponenten und Lösungen zu berücksichtigen und damit integraler Bestandteil der automationstechnischen Komponenten und Lösungen sind. Secure by Implementation ist der letzte Schritt und fordert, dass alle Funktionen einer Komponente oder einer Lösung einschließlich der integrierten IT-Security-Funktionen durch geeignete Programmiersprachen und Betriebssysteme umgesetzt werden müssen.

Die Zuverlässigkeit der Zielfunktion bei sicherheitskritischen Systemen muss auch bei Securityvorfällen (Angriffen) gesichert sein. Zur Erhöhung der Zuverlässigkeit und der Funktionsabsicherung bei Securityvorfällen kön-

nen diversitäre Software und/oder das Rechnen in anderen Zahlenräumen (z. B. Primzahlen) eingesetzt werden. Bei diversitärer Software sind entsprechende Überprüfungen an Inspektionspunkten oder ein Vergleich der Berechnungsergebnisse notwendig, um erfolgte Angriffe (Manipulationen) zu erkennen und die jeweilige Berechnungen von eine Verwendung auszuschließen (vgl. [27]). Bei mehrfach vorhandenen/redundanten Komponenten für Safety-Anforderungen können also durch den Einsatz von Voting Mechanismen sowohl zur Bewertung des Ergebnisses einer Berechnung als auch zur Verhaltensanalyse an Inspektionspunkten Manipulationen erkannt werden. Voraussetzung hierfür ist die voneinander unabhängige Implementierung der zu berechnenden Funktionen. Damit bieten sich Redundanzmechanismen direkt auch zur Erhöhung der Security-Eigenschaften an.

Beim Rechnen in alternativen Zahlenräumen besteht bei entsprechenden Prüfungen eine hohe Entdeckungswahrscheinlichkeit von Manipulationen mit der Möglichkeit auf alternative Berechnungen zu wechseln. In [10] wurde eine HW-unterstützte Berechnung im Raum der Primzahlen vorgeschlagen. Das erlaubt, Manipulationen zu erschweren bzw. über vom Zahlenraum abweichende Werte zu erkennen. Durch die HW-Unterstützung kann das Verfahren auch effizient realisiert werden. Das sicherheitskritische System kann dann zumindest in einen sicheren Zustand überführt werden.

6 Zusammenfassung

Softwarebasierte Funktionen erbringen einen erheblichen Mehrwert in allen technischen Produkten und Anlagen. Damit der Mehrwert nicht durch die dabei entstehenden Kosten zunichtegemacht wird, muss für die Entwicklung von Software ein entsprechender Herstellungsprozess zugrunde gelegt werden. Dabei spielt die Zuverlässigkeit von softwarebasierten Funktionen die zentrale Rolle und ist durch konstruktive Maßnahmen und nicht durch Testen sicherzustellen. Der Ansatz der Model Driven Architecture oder des Model Driven Software Development auf Basis der Unified Modeling Language, wie beispielsweise bei der Entwicklung des INSPECT-Systems angewandt, ist eine Möglichkeit, um aus den Klassendiagrammen und den zugehörigen Zustandsdiagrammen automatisch den Quelltext zu generieren. Mit diesem Ansatz wird die Zuverlässigkeit softwarebasierter Funktionen auch für den Einsatz unter Echtzeitbedingungen deutlich erhöht.

Ein Echtzeitsystem muss im Kern von seinen Eigenschaften her der Forderung nach Zuverlässigkeit von softwarebasierten Funktionen unter Echtzeitbedingungen entsprechen. Die Trennung der Funktionsausführung von der Funktionsausführungsverwaltung ist ein wesentlicher Aspekt, um Zuverlässigkeit bei gleichzeitig ausgeführten Funktionen zu erreichen. Das Rate Monotonic Scheduling mit der Response-Time-Analyse beweist die Ausführbarkeit.

Die Implementierung softwarebasierter Funktionen unter Echtzeitbedingungen erfordert Spracheigenschaften, die grundlegend für die Erfüllung der Zuverlässigkeit sind. Nach [58] gibt es Sprachen, die „by design, more suitable for secure programming“ sind. „Choosing such languages mitigates many security risks.“ Beispielsweise besitzen Sprachen wie Ada eine sehr gute Grundlage aufgrund ihrer fehlervermeidenden syntaktischen Struktur und der mathematisch präzisen Betrachtung von Typeigenschaften. Die Anforderungen sind auch für das Laufzeitsystem zwingend zu erfüllen. Bei automatischer Codegenerierung können der generierte Code mit Laufzeitprüfungen angereichert und Sprachdefizite kompensiert werden, die gesamten Informationen für eine statische und dynamische Analyse sind zusätzlich abzuleiten.

Offene und vernetzte Automatisierungssysteme müssen einerseits sicherheitskritische Anforderungen erfüllen und andererseits den Anforderungen der Informationssicherheit genügen. Die zweiseitige Sicherheit im Sinne der Safety und Security kann nur integrativ erfüllt werden. Security-Anforderungen sind durch direkte Eigenschaften der softwarebasierten Automatisierungsfunktionen zu realisieren.

Gerade in Hinblick auf zukünftige Entwicklungen wie der Industrie 4.0 (vgl. [2]), den Cyber-Physical Systems oder den sicherheitskritischen Infrastrukturen wie im Energiebereich [29] sowie der Cyber Crime Szene [16, 45] sind grundlegende Zuverlässigkeitsanforderungen in Verbindung mit Safety- und Security-Eigenschaften und deren Wechselwirkung nur in Gesamtheit der dargestellten Vorgehensweisen, Methoden, Sprachen und Konzepte integrativ zu erreichen.

Literatur

1. Cyber-Physical Systems. Acatech Position 2011. Deutsche Akademie der Wissenschaften, Berlin.
2. Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0. Deutsche Akademie der Wissenschaften, Berlin, 2013.
3. Ada Reference Manual, ISO/IEC 8652:2012(E), Language and Standard Libraries. Springer, Heidelberg, 2014.
4. ARTEMIS Strategic Research Agenda. ARTEMIS SRA WG, 2006.
5. Audsley, N.; Burns, A.; Richardson, M.; Tindell, K.; Wellings, A.: Applying New Scheduling Theory to Static Priority Pre-emptive

Scheduling. Report RTRG/92/120. Department of Computer Science, University of York (February 1992).
6. Beck, K.: Extreme Programming Explained. Embrace Change. 1st Edition, Addison Wesley, 2000.
7. Benra, J.; Keller, H. B.; Schiedermeier, G.; Tempelmeier, T.: Synchronisation und Konsistenz in Echtzeitsystemen. In Benra, J. T. [Hrsg.]: Software-Entwicklung für Echtzeitsysteme Berlin [u. a.], Springer, 2009, S. 49–65.
8. Bertsche, B.; Göhner, P.; Jensen, U.; Schinköthe, W.; Wunderlich, H.-J.: Zuverlässigkeit mechatronischer Systeme. Grundlagen und Bewertung in frühen Entwicklungsphasen. Springer, Berlin Heidelberg, 2009.
9. Bettenhausen, K. D. et al.: Infomrationssicherheit in der Automatisierung. atp, 49(4) (2007), p. 76–79.
10. Braun, J.; Mottok, J.; Miedl, C.; Geyer, D.; Minas, M.: Increasing the reliability of single and multi core systems with software rejuvenation and coded processing. In Plödereder, E.; Dencker, P.; Klenk, H.; Keller, H. B.; Spitzer, S. (Hrsg.): Proceedings Band 210 Automotive – Safety & Security 2012: Sicherheit und Zuverlässigkeit für automobile Informationstechnik Tagung 14.–15.11.2012 in Karlsruhe, Bonn, Gesellschaft für Informatik e.V.
11. ICS-Security-Kompendium. Bundesamt für Sicherheit in der Informationstechnik – BSI, 53133 Bonn, 2013.
12. Die Lage der IT-Sicherheit in Deutschlanbd 2014. Bundesamt für Sicherheit in der Informationstechnik – BSI, 53133 Bonn, 2014.
13. Buttazo, G. C.: Hard Real-time Computing Systems, Kluwer Academic publishers, 1997.
14. Cooling, J.: Software Engineering for Real Time Systems. Addison-Wesley, Pearson, Harlow, 2002.
15. CROSSTALK The Journal of Defense Software Engineering, December 2005, p. 16 ff.
16. The Internet Organised Crime Threat Assessment. Europol – European Police Office 2014.
17. DeMarco, T.: Warum ist Software so teuer? ... und andere Rätsel des Informationszeitalters. Carl Hanser Verlag, München, Wien 1997 (Titel der englischen Originalausgabe: „Why Does Software Cost So Much? And Other Puzzles of the Information Age“. 1995 by Tom DeMarco.
18. DGQ-NTG-Schrift Nr. 12–51. Software-Qualitätssicherung. Beuth/VDE-Verlag Berlin, 1986.
19. Echtle, K.: Fehlertoleranzverfahren. Springer Berlin Heidelberg, 1990.
20. Endres, A.; Rombach, H. D.: A Handbook of Software and Systems Engineering. Pearson/Addison Wesley, Essex, 2003.
21. Färber, G.: Prozeßrechentechnik. Springer-Lehrbuch. Springer; Auflage: 3., überarb. Aufl., 1994.
22. F.A.S.T., TU München, 2005.
23. Freiling, F.; Grimm, R.; Großpietsch, K.-E.; Keller, H. B.; Mottok, J.; Münch, I.; Rannenberg, K.; Saglietti, F.: Technische Sicherheit und Informationssicherheit – Unterschiede und Gemeinsamkeiten. Accepted for: Informatik Spektrum, GI, Springer, 37(1) (2014), S. 14–24.
24. Zipser, S.; Gommlich, A.; Matthes, J.; Fouda, Ch.; Keller, H. B.: Anwendung des Inspect-Systems zur kamerabasierten Analyse von Verbrennungsprozessen am Beispiel der thermischen Abfallbehandlung. FZKA 7014. Wissenschaftliche Berichte. Forschungszentrum Karlsruhe in der Helmholtz-Gemeinschaft, Institut für Angewandte Informatik, Forschungszentrum Karlsruhe GmbH, Karlsruhe, 2004.
25. Ganssle, J.: The Way ahead in Software Engineering. In Plödereder, E.; Dencker, P.; Klenk, H.; Keller, H. B.; Spitzer, S. (Hrsg.): Proceedings Band 210 Automotive – Safety & Security 2012: Sicherheit und Zuverlässigkeit für automobile Informationstechnik Tagung 14.–15.11.2012 in Karlsruhe, Bonn, Gesellschaft für Informatik e.V.
26. Girish, S.: High Maturity Pays Off. CROSSTALK – The Journal of Defense Software Engineering, January/February 2012, p. 9 ff.
27. Gherbi, A. et al.: Software Diversity for Future Systems Security. CROSSTALK, The Journal of Defense Software Engineering, September/October 2011, p. 10 ff.
28. Thesen und Handlungsfelder. Cyber-Physical Systems: Chancen und Nutzen aus Sicht der Automation. Verein Deutscher Ingenieure e.V. VDI/VDE-GMA. Düsseldorf 2013.
29. Hagenmeyer, V., Kemal Çakmak, H., Düpmeier, C., Faulwasser, T., Isele, J., Keller, H. B., Kohlhepp, P., Kühnapfel, U., Stucky, U., Waczowicz, S. and Mikut, R. (2016), Information and Communication Technology in Energy Lab 2.0: Smart Energies System Simulation and Control Center with an Open-Street-Map-Based Power Flow Simulation Example. Energy Technology, 4: 145–162. 10.1002/ente.201500304
30. Haykin, S.: Neural networks: A comprehensive foundation. Prentice Hall, Upper Saddle River, NJ, 1999.
31. Herrtwich, R. G.; Hommel, G.: Kooperation und Konkurrenz. Springer Verlag, Berlin, 1989.
32. Common Cybersecurity Vulnerabilities in Industrial Control Systems. Homeland Security, 2011. Control Systems Security Program. National Cyber Security Division.
33. Out of control – Why control systems go wrong and how to prevent failure. Health and Safety Executive (2003), No. 238.
34. Humphrey, W. S.: The Software Quality Challenge. CROSSTALK – The Journal of Defense Software Engineering, June 2008, p. 4 ff.
35. The Industrial Control Systems Cyber Emergency Response Team (ICS-CERT), https://ics-cert.us-cert.gov/
36. IEC 61508: Functional safety for electrical / electronic / programmable electronic safety-related systems. International Electrotechnical Comission, 1998.
37. DIN EN IEC 61508 CENELEC: Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme, Dez. 2001.
38. ISO/IEC 15939: Systems and Software Engineering – Measurement Process, 2007.
39. Ili, S.; Albers, A.; Miller, S.: Open innovation in the automotive industry. R&D Management, 40(3), (2010), S. 246 ff.
40. Isermann, R.: Mechatronische Systeme – Grundlagen. Springer Heidelberg, 2008, ISBN 978-3-540-32512-3
41. EN ISO 9000: 2000. Ersetzt durch die EN ISO 9000: 2005.
42. Keller, H. B.: Entwicklung von Software-Systemen mit Ada. Ada-Deutschland Workshop Bremen 1998, FZKA 6177.
43. Keller, H. B.: Maschinelle Intelligenz: Grundlagen, Lernverfahren, Bausteine intelligenter Systeme. Unter Mitarbeit von Fick, A.; Weinberger, T.; Gorges-Schleuter, M.; Eppler, W.; Schmauch, C.; Braunschweig [u. a.]: Vieweg, 2000 (Computational Intelligence).
44. Keller, H. B.; Müller, R.; Schiedermeier, G.; Tempelmeier, T.: Programmierung. Benra, J. T. [Hrsg.]: Software-Entwicklung für Echtzeitsysteme Berlin [u. a.]: Springer, 2009, S. 129–170.

45. Keller, H. B.: Im Kampf gegen Cybercrime. Chemie&more, 6.2012.
46. Keller, H. B.: Gesellschaftliche Relevanz der Informatik 16 als Strukturtechnologie. „Chancen und Risiken in der Wagnisgesellschaft“, 15. Oktober 2014, Bundesanstalt für Materialforschung und -prüfung (BAM), Berlin. Dokumentation des FORUM46 – Interdisziplinäres Forum für Europa e.V., Berlin, 2014, S. 16 ff.
47. Keller, H. B.: Technische Informatik. Vorlesungsskript SS 2015. KIT, Fakultät Maschinenbau, 2015.
48. Kersten, M.; Matthes, J.; Fouda, C.; Zipser, S.; Keller, H. B.: Customizing UML for the development of distributed reactive systems and code generation to Ada 95. Ada User Journal, September, 2001.
49. Kersten, M.; Keller, H. B.: Die Problematik der Abbildung von UML-Modellen auf Konstrukte der Programmiersprache Ada 95. In P. Dencker et al., [Eds.], Ada und Softwarequalität: Ada Deutschland Tagung 2001, Shaker Verlag, München, Ottobrunn, 2001.
50. Kanig, J.: A Comparison of SPARK with MISRA C and Frama-C. AdaCore, New York, 2016.
51. Lauber, R.: Prozessautomatisierung. Springer Verlag, Berlin, 1989.
52. Laird, L.; Brennan, C.: Software Measurement and Estimation: A Practical Approach. Wiley & Sons, 2006.
53. Liu, C. L.; Layland, J. (1973): Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of the ACM, 20, 1 (1973), p. 46–61.
54. Liu, J. W. S.: Real Time Systems. Prentice Hall, New Jersey, 2000.
55. Matthes, J.; Keller, H. B.; Heker, W.-D.; Kersten, M.; Fouda, C.: Zuverlässige Software durch den Einsatz von UML, MDA und der Programmiersprache Ada. Informatik 2003 – Schwerpunkt Sicherheit – Schutz und Zuverlässigkeit, 29.09.–02.10.2003, Frankfurt/Main.
56. Morre, J. W. et al.: New ISO/IEC Technical Report describes Vulnerabilities in Programming Languages. CROSSTALK The Journal of Defense Software Engineering, March/April2012, p. 27–30.
57. NAMUR Empfehlung NE 153: Automation Security 2020 – Anforderungen an Design, Implementierung und Betrieb künftigerindustrieller Automatisierungssysteme, NAMUR 2015.
58. Source Code Security Analysis Tool. Functional Specification Version 1.1. NIST Special Publication 500-268 v1.1. U. S.Department of Commerce. National Instiuteof Standards and Technology, 2011.
59. Guide to Industrial Control Systems (ICS) Security. NIST Special Publication 800-82, Revision 2. U. S.Department of Commerce. National Instiuteof Standards and Technology, 2015.
60. Nosek, J. T.; Palvia, P.: Software maintenance management: Changes in the last decade. Journal of Software Maintenance: Research and Practice. 2(3), (September 1990), p. 157–174.
61. Pomberger, G.: Softwaretechnik und Modula 2. Hanser, München, 1984.
62. Ramberger, S.; Gruber, T.: Error Distribution in Safety-Critical Software & Software Risk Analysis Based on Unit Tests. Experience Report. WSRS Ulm – 20 Sept. 2004. ARC Seibersdorf research GmbH.
63. Rojas, R.: Neural networks: A systematic introduction. Springer, Berlin, 1996.
64. Rosenstiel, W.: Abschlussbericht DFG-Schwerpunktprogramm 1040: Entwurf und Entwurfsmethodik eingebetteter Systeme, Universität Tübingen, 1997–2003/ BMW AG.
65. Schäfer, M.; Gnedina, A.; Bömer, T.; Büllesbach, K.-H.; Grigulewitsch, W.; Rueß, G.; Reinert, D.: Programmierregeln für die Erstellung von Software für Steuerungen mit Sicherheitsaufgaben. Fb 812, Bundesanstalt für Arbeitsschutz und Arbeitsmedizin. Dortmund/Berlin, 1998.
66. Seacord. R. C.; Householder, A. D.: A Structured Approach to Classifying Security Vulnerabilities. TECHNICAL NOTE, CMU/SEI-2005-TN-003, January 2005.
67. CMMI – Capability Maturity Model Integration. CMU/SEI-2002-TR-029, ESC-TR-2002-029. August 2002.
68. Sneed, H. M.; Baumgartner, M.; Seidl, R.: Der Systemtest. Hanser Verlag München, 2007.
69. Stallings, W.: Betriebssysteme. 4. Auflage. Pearson Studium, München, 2003.
70. CHAOS MANIFESTO 2013, Think Big, Act Small. The Standish Group International, Inc., www.standishgroup.com.
71. Summerville, I.: Software Engineering. Pearson Studium, München, 2007.
72. Tannenbaum, A. S.: Moderne Betriebssysteme. 2. Auflage. Pearson Studium, München, 2003.
73. OMG Unified Modeling Language, Version 1.4, June 1999.
74. Bulletins of the Department of Homeland Security’s United States Computer Emergency Readiness Team (US-CERT), https://www.us-cert.gov/ncas/bulletins
75. VDE-Trendstudie IKT-Sicherheit. VDE Verband der Elektrotechnik Elektronik Informationstechnik e.V., Frankfurt, 2012.
76. Deutsche Normungs-Roadmap Smart Home + Buidling.VDE Verband der Elektrotechnik Elektronik Informationstechnik e.V., Frankfurt, 2015.
77. Eschenfelder, D.; Gelfort, E.; Graßmuck, J.; Keller, H. B.; Langenbach, C.; Lemiesz, D.; Otremba, F.; Pilz, W. D.; Rath, R.; Schulz-Forberg, B.; Wilpert, B.: Qualitätsmerkmal ‚Technische Sicherheit‘. Eine Denkschrift des Vereins Deutscher Ingenieure, Düsseldorf: VDI-Verl., 2010, ISBN 978-3-931384-68-5.
78. Voges, U.: Software-Diversität und ihr Beitrag zur Sicherheit. In [42].
79. Information Technology – Programming Languages – Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use ISO/IEC TR 24772 Edition 2 (TR 24772 WG 23/N 0389), ISO/IEC JTC 1/SC 22/WG 23, 2012.
80. Younan, Y.: An Overview of Common Programming Security Vulnerabilities and Possible Solutions. Dissertation, Freie Universität Brüssel, 2003.
81. Zelkowitz, M. V.; Shaw, A. C.; Gannon, J. D.: Principles of Software Engineering and Design, Prentice Hall Inc., Englewood Cliffs, NJ, 1979.
82. Zöbel, D.; Albrecht, W.: Echtzeitsysteme. Grundlagen und Techniken. Thomson Publishing, 1995.
83. Integrierte Technologie-Roadmap Automation 2015+, ZVEI Automation 2006.
84. ISA, Security for industrial automation and control systems. ISA 99.02.02. Copyright © 2011 by ISA, 67 Alexander Drive, P. O. Box 12277, Research Triangle Park, NC 27709 USA sowie www.isa.org

Autoreninformationen

Hubert B. Keller
Karlsruher Institut für Technologie (KIT),
Institut für Angewandte Informatik (IAI),
Kaiserstr. 12, 76131 Karlsruhe
hubert.keller@kit.edu

Hubert B. Keller leitet die Arbeitsgruppe ProSys – Prozessoptimierung, intelligente Sensorsysteme und sichere Software am Institut für Angewandte Informatik des KIT. Er ist für das Querschnittsthema „Sichere Software“ im Energy Lab 2.0 und im BMBF Projekt KASTEL für das Thema „Anomalieerkennung durch modellbasierte Plausibilitätsprüfung“ zur Erhöhung der Sicherheit in industriellen Leitsystemen verantwortlich. Sein Diplom in den Ingenieurswissenschaften erwarb er 1982 an der Universität Karlsruhe, seine Promotion erhielt er in 1988 von der Universität Clausthal. Bis 1984 hat er als Dipl.-Ingenieur bei Siemens im Bereich Entwicklung hochverfügbare Prozessleitsysteme gearbeitet. Seine Arbeits- und Forschungsschwerpunkte sind zuverlässige und sichere Echtzeit-Systeme, Software Engineering, Maschinelle Intelligenz, intelligente Sensornetze und Prozessleitsysteme. Er ist Dozent für Technische Informatik der Fakultät für Maschinenbau des KIT sowie für Realzeitsysteme an der DHBW Karlsruhe.

Jörg Matthes
Karlsruher Institut für Technologie (KIT),
Institut für Angewandte Informatik (IAI),
Kaiserstr. 12, 76131 Karlsruhe

Jörg Matthes arbeitet am Institut für Angewandte Informatik am Karlsruher Institut für Technologie (KIT) in der Arbeitsgruppe ProSys – Prozessoptimierung, intelligente Sensorsysteme und sichere Software. Hauptarbeitsgebiete: Maschinelles Sehen, Hochtemperaturprozesse, Mess- und Regelungstechnik.

Oliver Schneider
Karlsruher Institut für Technologie (KIT),
Institut für Angewandte Informatik (IAI),
Kaiserstr. 12, 76131 Karlsruhe

Oliver Schneider arbeitet am Institut für Angewandte Informatik am Karlsruher Institut für Technologie (KIT) in der Arbeitsgruppe ProSys – Prozessoptimierung, intelligente Sensorsysteme und sichere Software. Hauptarbeitsgebiet: Softwareentwicklungsprozesse für sichere Software.

Veit Hagenmeyer
Karlsruher Institut für Technologie (KIT),
Institut für Angewandte Informatik (IAI),
Kaiserstr. 12, 76131 Karlsruhe

Veit Hagenmeyer ist Direktor des Instituts für Angewandte Informatik und Professor für Energieinformatik in der Fakultät für Informatik des KIT. Davor war er Kraftwerksdirektor bei BASF in Ludwigshafen. Sein Interesse liegt im Umgang mit polyvalenten Energiesystemen (Strom, Erdgas, Dampf, lokale Wärmeverteilung) und Co-Erzeugung, im robusten Entwurf von Energiesystemen auf Basis von Energiezustandsdaten und der praktischen Umsetzung im Demonstrator Energy Lab 2.0 des KIT.