Modellbasierte Laufzeitüberprüfung vernetzter eingebetteter Systeme

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
vorgelegt der Fakultät für Angewandte Informatik
der Universität Augsburg

Christian Drabek

Januar 2021

Erstgutachter:	Prof. Dr. Bernhard Bauer
Zweitgutachter:	Prof. Dr.-Ing. Rudi Knorr
Mündliche Prüfung:	14. Juli 2021

Kurzfassung

Vernetzte eingebettete Systeme sind heute allgegenwärtig. Um auch komplexere Dienste anbieten zu können, interagieren immer mehr Komponenten in einem System miteinander. Für die Qualität der entstehenden Applikation ist, neben der korrekten Funktion der einzelnen Komponenten, die geordnete Kommunikation der Komponenten untereinander entscheidend. Ein robustes System funktioniert weiter, nachdem ein unkritischer Fehler oder eine Abweichung von seiner Spezifikation aufgetreten ist. Daher kann es während einer einzigen Ausführung mehrfach von der Spezifikation abweichen. Die Abweichungen sollten behoben werden, denn sie können die Leistung reduzieren oder in anderen Situationen zu kritischen Fehlern führen. Die Fehlerbehebung wird erleichtert, wenn die einzelnen Abweichungen identifiziert werden. Daher wird in dieser Arbeit ein neuer Ansatz erforscht, der alle erkennbaren Abweichungen zwischen der Ausführung eines Systems und dessen Spezifikation findet.

Als Grundlage für diese Überprüfung wird ein Modell zur Beschreibung des Soll-Verhaltens solcher Systeme vorgestellt. Das Modell stellt dabei die Schnittstellen des Systems in den Vordergrund, da ein Kommunikationskanal häufig einfacher überwacht werden kann, als die Verarbeitung innerhalb einer Komponente. Es ist modular und in Schichten aufgeteilt, um eine hohe Erweiterbarkeit und Wiederverwendbarkeit zu erreichen.

Da eine Abweichung von der Spezifikation mit Verhalten einhergeht, das häufig nicht definiert ist, entsteht eine Ungewissheit über den aktuellen Systemzustand. Die neue Methode *Resumption* ermöglicht dennoch mehr als nur den ersten Verstoß gegen die Spezifikation zu erkennen, wenn diese als Automat vorliegt. Ein mit Resumption erweiterter Monitor meldet, wenn Beobachtungen nicht darauf abgebildet werden können. Zudem wird die Verwendung von Resumption mit der vorgestellten Modellierung untersucht.

Durch eine Realisierung des Ansatzes wird eine Überprüfung der Praxistauglichkeit und eine empirische Evaluierung von Resumption möglich. Nutzen und Verwendbarkeit des Ansatzes werden in verschiedenen Anwendungsbeispielen demonstriert. Durch den Vergleich unterschiedlicher Strategien zur Erweiterung von Monitoren mit Resumption wird untersucht, wann auf Basis solcher Soll-Beschreibungen alle beobachtbaren Abweichungen erkannt werden.

Der modulare Ansatz zur Modellierung ermöglicht eine schrittweise Erweiterung der Methodik und kann so auf Anforderungsprofile verschiedener Szenarien angepasst werden. Durch Resumption kann die Laufzeitüberprüfung einen Traces nach unerwartetem Verhalten für ein wählbares Fehlermodell segmentieren. Selbst wenn eine erkennbare Abweichung nicht direkt mit einem unerwarteten Ereignis zusammenfällt, kann sie auf ein Segment des Traces eingegrenzt werden.

Vorwort

Die vorliegende Arbeit entstand begleitend zu meiner Tätigkeit am Fraunhofer-Institut für Kognitive Systeme IKS, ehem. Fraunhofer-Institut für Eingebettete Systeme und Kommunikationstechnik ESK. Ich möchte allen danken, die mich bei dieser Aufgabe unterstützt haben.

An erster Stelle möchte ich mich bei meinem Doktorvater Prof. Dr. Bernhard Bauer für die Ermöglichung und die Unterstützung der Arbeit bedanken. Ebenfalls gilt mein Dank Prof. Dr.-Ing. Rudi Knorr für die Übernahme des Zweitgutachtens und dem damit verbundenen Interesse an meiner Arbeit.

Mein besonderer Dank gilt Dr. Gereon Weiß für die Übernahme der Betreuung meiner Arbeit, sowie ihm als meinem Vorgesetzten und meinen ehemaligen und aktuellen Kollegen bei Fraunhofer IKS für die angenehme Arbeitsatmosphäre. Zusätzlich möchte ich mich auch bei den Studenten und Projektpartnern bedanken, die mich bei der Realisierung der Konzepte und der Durchführung verschiedener Experimente unterstützt haben. Allen zusammen gilt mein Dank für die vielen interessanten Diskussionen.

Zu guter Letzt gilt mein Dank meiner Familie und meinen Freunden für ihre kontinuierliche Unterstützung und ihr Verständnis.

Inhaltsverzeichnis

Abbildungsverzeichnis

Glossar

Applikation	Der anwendungsspezifische Teil des Systems, dessen Verhalten geprüft werden soll.
Applikationslogik	Die Formale Beschreibung des Verhaltens der Applikation.
Automat	Kurzform für Zustandsautomat.
Client	Eine Komponente, die eine Schnittstelle von einem Server benötigt.
endlicher Automat	Automat mit einer endlichen Anzahl an Zuständen.
Ereignis	Äquivalenzklasse zur Beschreibung der Interpretation der Inhalte einer Nachricht.
Komponente	Ein Bestandteil eines Systems, der über Schnittstellen mit dem restlichen System interagiert.
Laufzeit	Der Zeitraum der Ausführung eines Systems, bzw. der dadurch geprägte Zustand.
Middleware	Software, die Kommunikation zwischen anderen (Software-)Komponenten ermöglicht.
Modell	Eine abstrakte Beschreibung eines bestimmten Sachverhalts oder ein Konzept zur Beschreibung dessen.
Nachricht	Eine konkrete Interaktion mit einem Nachrichtentyp zwischen zwei Komponenten.
Nachrichtentyp	Eine Interaktionsmöglichkeit zwischen zwei Komponenten.
robust	Nicht empfindlich gegenüber unvorhergesehenen Ereignissen, solange es sich um keine schwerwiegenden Fehler handelt.
Schnittstelle	Die Interaktionsmöglichkeiten einer Komponente.
Server	Eine Komponente, die Clients eine Schnittstelle anbietet.
Soll-Verhalten	Die Beschreibung des erwarteten Verhaltens eines Systems.
Spezifikation	Die Beschreibung des Systems als Grundlage für dessen Entwicklung.

Sprachen-Werkbank	Ein Werkzeug oder eine Sprache mit der DSLs erstellt werden können.
Trace	Eine Sequenz von Nachrichten oder Ereignissen.
Validierung	Überprüfung, ob ein System seinen Zweck bzw. seine Anforderungen erfüllt.
Verifikation	Überprüfung, ob ein System eine Spezifikation einhält.
vernetztes eingebettetes System	Zusammenschluss von Komponenten auf Rechenplattformen mit begrenzten Resourcen, die untereinander kommunizieren, um gemeinsam Aufgaben zu erfüllen.
Zeit-behafteter Automat	engl.: *Timed Automat*, Zustandsautomat mit rücksetzbaren Uhren.
Zustandsautomat	Ein abstraktes Modell eines Prozesses basierend auf Zuständen und Transitionen zwischen diesen.

Abkürzungen

CEP	komplexe Ereignis-Verarbeitung (engl.: *Complex Event Processing*)
DSL	domänenspezifische Sprache (engl.: *Domain Specific Language*)
EMF	Eclipse Modelling Framework
EMOF	Essential MOF
EV	eingebetteter Überprüfer (engl.: *Embedded Verifier*)
fUML	foundational UML
GPL	universale Sprache (engl.: *General Purpose Language*)
GUI	grafische Benutzerschnittstelle (engl.: *Graphical User Interface*)
HMI	Mensch-Maschine-Schnittstelle (engl.: *Human Machine Interface*)
IDL	Schnittstellenbeschreibungssprache (engl.: *Interface Definition Language*)
MOF	Meta Object Facility
OMG	Object Management Group
PSSM	Precise Semantics of UML State Machines
RV	Laufzeitüberprüfung (engl.: *Runtime Verification*)
SMT	Modulo-Theorie der Erfüllbarkeit (engl.: *Satisfiability Modulo Theory*)
SuO	System unter Beobachtung (engl.: *System under Observation*)
SuT	System unter Test (engl.: *System under Test*)
TADL	Zeitverhalten-erweiterte Beschreibungssprache (engl.: *Timing Augmented Description Language*)
UML	Unified Modeling Language

1 Kapitel 1
Einleitung

Dieses Kapitel motiviert die Arbeit und beschreibt die Herausforderungen bei der Überprüfung vernetzter eingebetteter Systeme. Darauf aufbauend werden die Ziele definiert und die wissenschaftlichen Beiträge formuliert. Anschließend wird die Arbeit in den Kontext bereits existierender Veröffentlichungen des Autors eingeordnet und auf die Struktur der Arbeit eingegangen.

1.1 Motivation

Software ist heutzutage allgegenwärtig und ermöglicht technischen Geräten einen hohen Grad an Flexibilität. Ohne Software wären viele der Funktionen nicht realisierbar. Moderne Software ist nicht monolithisch, sondern besteht aus einzelnen Komponenten die miteinander interagieren. Diese Modularität hat verschiedene Gründe und Vorteile, wie eine hohe Wiederverwendbarkeit und Austauschbarkeit einzelner Bausteine. Insgesamt wird dadurch die Wartbarkeit der Software vereinfacht. Modulare Software ermöglicht eine verteilte Entwicklung und wird oft von mehr als einem Entwickler oder einer einzelnen Firma entwickelt. Erst dadurch kann die für größere Projekte anfallende Arbeitslast bewältigt werden. In verschiedenen Anwendungsgebieten werden neue Arten von Diensten durch die Kombination von existierenden Softwarekomponenten erzeugt. Die Interaktionspartner einer Softwarekomponente sind nicht immer von vorneherein bekannt oder existieren noch gar nicht.

Komponenten können gemeinsam auf der gleichen oder verteilt auf vernetzter Hardware laufen. Bei einem Wechsel der Hardware kann sich diese Aufteilung wieder ändern. Um die Kommunikation transparent für die Komponenten zu gestalten, wird eine sog. Middleware zur Verbindung der Komponenten eingesetzt. Dadurch kann jede Komponente entwickelt werden, ohne sich mit dem realen Aufbau des Ziel-Systems befassen zu müssen. Diese Modularisierung kann aber nur funktionieren, wenn festgelegt wird, wie die

Interaktionen zwischen den Modulen ablaufen soll. Dabei bleibt es eine Herausforderung, die korrekte Funktionalität zu verifizieren und von der Spezifikation abweichende Komponenten zu identifizieren. Nicht nur die statischen Schnittstellen zwischen Komponenten müssen kompatibel sein, sondern auch der Ablauf der Interaktionen darüber [Dra+13; HKW17].

Nach Heffernan, Macnamee und Fogarty [HMF14] ist die komplette Verifikation eines eingebetteten Programms allgemein unlösbar. Der Verifizierungsprozess des finalen Produkts bleibt also zwangsläufig unvollständig. Deshalb wird in verschiedenen Ansätzen [Fal+18; Rab+17; LS09; DGR04] vorgeschlagen, solche Systeme durch den Einsatz von sogenannten Monitoren zur Laufzeit zu beobachten. Diese überprüfen während der Ausführung des Systems, ob seine Komponenten ihre Spezifikationen einhalten.

Ein robustes System kann weiterhin Dienste anbieten, nachdem ein unkritischer Fehler oder eine Abweichung von seiner Spezifikation aufgetreten ist. Daher kann es während einer einzigen Ausführung mehrfach von der Spezifikation abweichen. Die Abweichungen sollten behoben werden, denn sie können die Leistung reduzieren oder in anderen Situationen zu kritischen Fehlern führen. Die Fehlerbehebung wird erleichtert, wenn die einzelnen Abweichungen identifiziert werden, statt nur allgemein ein Abweichen festzustellen. Dadurch kann der Testaufwand zur Beobachtung von Abweichungen reduziert werden, insbesondere wenn sie selten und schwer reproduzierbar sind. Der Aufwand für die Erstellung eines solchen Monitors wird reduziert, wenn Artefakte aus der Spezifikationsphase wiederverwendet werden. Eine gängige Methode, solche Interaktionen und Protokolle zu spezifizieren, sind Zustandsautomaten. Diese geben häufig nur das erwartete Verhalten und ausgewählte Fehlerfälle an. Sie haben daher häufig eine unvollständige Transitions-Funktion. Damit bleibt offen, wie die Überprüfung im Fall einer unvorhergesehenen Abweichung im Interaktionsverhalten fortgesetzt werden kann. Der Monitor sollte aber nicht deshalb terminieren, sondern auch weitere Abweichungen erkennen. Im Fokus dieser Arbeit steht die Überprüfung zur Laufzeit auf Basis einer positiven Beschreibung des Soll-Verhaltens und die Rückmeldung aller damit erkennbaren Abweichungen.

1.2 Herausforderungen

Vernetzte eingebettete Systeme sind heute allgegenwärtig. Um auch komplexere Dienste anbieten zu können, interagieren immer mehr Komponenten in einem System. Für die Qualität der entstehenden Applikation ist, neben der Korrektheit der einzelnen Komponenten, die geordnete Kommunikation der Komponenten untereinander entscheidend. Um Komponenten unabhängig entwickeln zu können, wird die Kommunikation der Komponenten spezifiziert. Wird die Spezifikation bei der Implementierung nicht eingehalten oder falsch

verstanden, kommt es bei der Integration der Komponenten zu Problemen. Um den Konflikt aufzulösen, muss die Ursache identifiziert und lokalisiert werden.

Die nachfolgend erarbeiten Herausforderungen sind abgeleitet aus der Infotainment Domäne von Fahrzeugen, aber treffen in ähnlicher Form auch auf viele andere Domänen zu. Sie wurden vom Autor dieser Arbeit zunächst in [Dra+13] formuliert, sowie in [DPW15] und [DW17] aktualisiert.

Herausforderung 1.
***Technologie-unabhängige Beschreibung*:** Eine Vielzahl an Middleware-Technologien und Hardware-Schnittstellen werden verwendet und regelmäßig ersetzt. Um nicht bei jeder Migration von einer Technologie auf die nächste das Verhalten der Applikation erneut zu beschreiben, sollte die Modellierung des Verhaltens von diesen technischen Details abstrahieren. Das Verhalten muss dafür unabhängig von der Technologie beschrieben, aber dennoch auf diese genau abgebildet werden können. Anpassungen sollten nur dort und in dem Umfang von Nöten sein, wie sie durch den Technologiewechsel zwingend erforderlich sind. Entsprechend sollten sie sich möglichst auf Änderungen der Beschreibung dieser Abbildung beschränken.

Herausforderung 2.
***Klare Semantik der Beschreibung*:** Die gleiche Spezifikation wird von verschiedenen Entwicklern, die vielleicht nie direkt miteinander Kontakt haben, betrachtet und interpretiert. Dennoch sollen die entwickelten Komponenten austauschbar sein und miteinander interagieren können. Um diese Kompatibilität sicherzustellen, muss die Spezifikation den Interpretationsspielraum minimieren. Beispielsweise, indem die zur Spezifikation verwendete Beschreibungssprache auf eine ausführbare Semantik abgebildet wird.

Herausforderung 3.
***Zustandsexplosion*:** Viele Software-basierte Features entstehen durch die Interaktion mehrerer Software-Komponenten. Jede hat dabei eigene Zustände, die zu dem Zustand des Gesamtsystems kombiniert werden. Insbesondere wenn dynamisch instantiierte Interaktionen hinzukommen, sind spezielle Beschreibungsformen notwendig, da einzelne einfache Zustände diese nicht mehr repräsentieren können.

Herausforderung 4.
***Nicht-funktionale Anforderungen*:** Die korrekte Orchestrierung der Interaktionen zwischen Komponenten muss nicht-funktionale Anforderungen berücksichtigen, wie beispielsweise das Zeitverhalten. Dazu müssen diese Informationen in der Verhaltensbeschreibung beschreibbar sein und ebenso überwacht werden können. Dies ist unerlässlich für sicherheitsrelevante Funktionen, die häufig Echtzeitanforderungen erfüllen müssen, d. h. sie

antworten innerhalb einer vorher bestimmten Zeit. Aber auch für nicht-kritische Funktionen können beispielsweise fehlende Antworten erkannt werden, wenn eine maximale Antwortzeit definiert wird.

Herausforderung 5.
***Erkennung von Abweichungen*:** Durch einen Abgleich zwischen Spezifikationsmodell und Implementierung können Abweichungen vom Soll-Verhalten identifiziert werden. Tritt bei der Integration des Systems fehlerhaftes Verhalten auf, können auf diese Weise die Komponenten ermittelt werden, die nicht die Spezifikation erfüllen. Eine explizite Überprüfung hilft dabei, verstecktes Fehlverhalten einer Komponente frühzeitig zu finden. Sind die anderen Komponenten gegen einen Fehler robust implementiert, wird die Abweichung nicht nach außen sichtbar, außer in einer abweichenden Kommunikation. Entfällt diese Robustheit später, beispielsweise wenn eine der anderen Komponenten ersetzt wird, funktioniert das System nicht mehr einwandfrei, selbst wenn die neue Komponente ihre zugrundeliegende Spezifikation erfüllt.

Herausforderung 6.
***Unvollständige Beschreibung*:** Die Erstellung einer exakten Spezifikation bringt einen hohen Aufwand mit sich und kann effizientere Realisierungsmöglichkeiten verhindern. In einer sogenannten losen Spezifikation wird deshalb nur die Definition der notwendigen Anforderungen verlangt. Beispielsweise indem sie das nach außen sichtbare Verhalten vorgibt, aber nicht jedes Implementierungsdetail. Spezifikationsmodelle können deshalb absichtlich an einigen Stellen unvollständig sein und sich auf die wichtigen Teile des Systemverhaltens fokussieren. Da so auch leicht von der Spezifikation unerwartetes, aber erwünschtes Verhalten auftreten kann, sollte dieses während einer Überprüfung zwar gemeldet werden, die Überprüfung aber nicht abbrechen. Ohne Möglichkeit zur Fortsetzung müsste sonst erst zwangsweise die Spezifikation ergänzt werden, dass die nachfolgenden Interaktionen ebenfalls auf weitere mögliche Abweichungen untersucht werden können. Gerade bei einem verteilten System mit unabhängigen Komponenten ist es auch nicht immer oder nur schwer möglich, einen bestimmten Zustand als Startpunkt der Überprüfung festzulegen oder herzustellen. Der Mechanismus zur Verifikation muss deshalb eigenständig diesen Zustand erkennen können.

Herausforderung 7.
***Reaktion auf Abweichungen im Betrieb*:** Auch wenn jede einzelne Komponente im System separat getestet wurde, kann es bei einem integrierten System zu ungewollten Nebeneffekten kommen, wodurch sich das Verhalten einer Komponente zur Laufzeit geringfügig verändern kann. Ebenso ist es aufgrund der hohen Komplexität solcher Systeme nicht immer wirtschaftlich oder möglich, alle Funktionalitäten mit wiederholbaren Testfällen

zu prüfen. Umso wichtiger ist es, dass Abweichungen vom gewünschten Verhalten zeitnah erkannt werden, um mögliche Fehlerursachen einzugrenzen. Beispielsweise kann, wenn unerwartetes Verhalten bereits während einer Testfahrt erkannt wird, die letzte Aktion wiederholt werden, um direkt mehr relevante Daten zu erzeugen. Da auch weiteres Auftreten des Fehlers gemeldet wird, kann die auslösende Aktion während der Fahrt eingegrenzt werden. Wird die Abweichung aber erst nach Abschluss der Testfahrt durch nachträgliche Analyse der Aufzeichnungen erkannt, muss das verursachende Szenario wieder rekonstruiert und anschließend über weitere Testfahrten überprüft werden. Nachdem das verursachende Szenario gefunden wurde, kann begonnen werden, den Fehler gezielt zu stimulieren und relevante Daten aufzuzeichnen. Die Erkennung der Abweichung zur Laufzeit ermöglicht es ebenso, Systeme zu entwickeln, die aktiv auf Abweichungen im Verhalten reagieren und beispielsweise eine problematische Komponente durch eine andere Implementierung austauschen, um einen sicheren Betrieb zu gewährleisten.

1.3 Wissenschaftliche Zielsetzung

Die wissenschaftliche Zielsetzung dieser Arbeit ist es einen neuen Ansatz zur Laufzeitüberprüfung der Interaktionen von Software-Komponenten vernetzter eingebetteter Systeme zu erforschen. Dieser beinhaltet verschiedene Methoden mit denen beobachtete Sequenzen von Ereignissen auf das zugehörige Spezifikationsmodell abgebildet und überprüft werden können. Als primärer Anwendungsfall wird die Kommunikation zwischen Software Komponenten im Fahrzeug betrachtet. Im Folgenden werden die Ziele dieser Arbeit aufgestellt und in Bezug zu den Herausforderungen gesetzt. Anschließend wird der in dieser Arbeit verfolgte Ansatz vorgestellt.

Ziel 1.
***Aufbau eines modularen Ansatzes zur Beschreibung von Soll-Verhalten*:** Im Rahmen dieser Arbeit soll ein neuer Ansatz zur Beschreibung des Soll-Verhaltens der Kommunikation eines vernetzten eingebetteten Systems entwickelt werden, mit dem der Aufwand für den Entwickler reduziert wird. Dazu werden gezielt die folgenden Herausforderungen aus Abschnitt 1.2 adressiert. Herausforderung 1 erfordert, dass gewisse Teile der Modellierung wiederverwendet und andere Teile, insbesondere solche, die sich mit technischen Details der Implementierung befassen, ausgetauscht werden können. Von der Modellierung wird deshalb eine strikte Trennung dieser Teile gefordert. Dabei wird darauf geachtet, dass die Semantik des Modells präzise bleibt, so dass, wie durch Herausforderung 2 gefordert, auch bei verteilter Entwicklung jeder das gewünschte Verhalten nachvollziehen kann. Die Semantik muss dabei so präzise sein, dass das Modell mit der modellierten Genauigkeit ausgeführt werden kann. Wie in Herausforderung 3 beschrieben, muss die

Modellierung dabei auch komplexe Interaktionen repräsentieren können. Ziel ist ein Kompromiss zwischen kompakter Modellierung und Ausdruckskraft. Es wird zudem erwartet, dass mit neuen Technologien und Einsatzzwecken in Zukunft auch neue Anforderungen an die Möglichkeiten zur Modellierung gestellt werden. Der Ansatz soll deshalb modular aufgebaut werden. Das vereinfacht es, bei Bedarf einzelne Teile der Beschreibungssprache gezielt auszutauschen oder um neue Konzepte zu erweitern. Um eine solche Erweiterung zu demonstrieren, wird betrachtet, wie die Überprüfung von Zeitverhalten in die Beschreibungssprache integriert werden kann. Eine solche Option zur Beschreibung dieser nicht-funktionalen Eigenschaften des Verhaltens adressiert Herausforderung 4.

Ziel 2.
***Methodik zum Finden aller Abweichungen vom Soll-Verhalten zur Laufzeit*:** Herausforderung 5 hebt hervor, dass alle Abweichungen der Ausführung einer Komponente im Vergleich zur Spezifikation gefunden werden müssen. Für die einfache Überprüfung, ob eine Komponente einer Spezifikation entspricht, genügt eine Unterscheidung der Bewertung nach *ja* oder *nein*. Aber für die Analyse und das anschließende Beheben einer bestimmten Abweichung ist es wichtig zu wissen, wann genau sie und ebenso ob sie mehrfach auftritt. Ein Ziel dieser Arbeit ist deshalb, eine Methodik zu erarbeiten, die es erlaubt auf Basis der Beschreibungssprache entsprechend Ziel 1 eine automatisierte Überprüfung durchzuführen. Der dazu entwickelte Verifikationsmechanismus muss in der Lage sein, alle erkennbaren Abweichungen zu finden und diese zu melden. Das beobachtete Fehlverhalten ist dabei normalerweise nicht vorhersehbar. Insbesondere erwartet die vorgeschlagene Beschreibungssprache keine Angaben über mögliche Auswirkungen von Abweichungen. Die Überprüfung muss in jedem Fall fortgesetzt werden, um alle Abweichungen finden zu können. Es wird deshalb in dieser Arbeit untersucht, wie ein Monitor ohne explizite Angabe des Verhaltens bei Abweichungen weiter überprüfen kann. Wie Herausforderung 6 heraushebt, sind Spezifikationen nicht immer vollständig. Es kann daher Verhalten geben, das nicht in der Spezifikation beschrieben wurde. Ein mit dem erarbeiteten Ansatz erstellter Monitor sollte bei Identifikation solchen Verhaltens nicht die Überprüfung abbrechen müssen. Er sollte auch das weitere beobachtbare Verhalten mit der Spezifikation vergleichen. Herausforderung 7 erfordert, dass die Überprüfung zur Laufzeit geschehen soll. Die Laufzeitkomplexität der entwickelten Methodik muss deshalb so gering wie möglich sein, um auch einen Einsatz in oder nahe dem Ziel-System zu ermöglichen.

Ziel 3.
***Realisierbare Methodik*:** Der vorgeschlagene Ansatz soll praxistauglich, also funktionsfähig und auf reale Anwendungsfälle anwendbar sein. Wie bereits Herausforderung 1 betont, muss der Verifikationsmechanismus mit einer Vielzahl an unterschiedlichen Datenquellen umgehen können. Es muss leicht möglich sein die gesammelten Daten aus der Ausführung

des Systems, die häufig in einem proprietären Format vorliegen, mit begrenztem Aufwand für die Überprüfung zu verwenden. Dazu ist ein modulares und erweiterbares Konzept vorzusehen, um die notwendige Flexibilität zu gewährleisten. Zur Demonstration der Anwendbarkeit ist eine Entwicklungsumgebung bereitzustellen, mit der zu überwachende Spezifikationen erstellt und Überprüfungen durchgeführt werden können. Zudem sollte die grafische Darstellung der Analyse-Ergebnisse zum besseren Verständnis möglich sein. Dies erleichtert die Plausibilisierung und weitere Auswertung der Ergebnisse einer Überprüfung. Ebenso muss die Möglichkeit bestehen, eine kompakte Version der Überprüfung zu erzeugen. Nur wenn der Mechanismus auf einem eingebetteten Gerät nahe dem Ziel-System lauffähig ist, kann die Auswertung und eine Reaktion auf Abweichungen zur Laufzeit gemäß Herausforderung 7 erfolgen.

Ziel 4.
***Evaluation des Ansatzes*:** Der Nutzen und die Verwendbarkeit des Ansatzes soll demonstriert werden. Um zu zeigen, dass der modulare Ansatz zur Beschreibung von Soll-Verhalten die gestellten Anforderungen erfüllt und flexibel anpassbar ist, wird die Anwendung auf verschiedene Szenarien betrachtet. Ebenso ist zu untersuchen, ob auf Basis solcher Soll-Beschreibungen alle beobachtbaren Abweichungen erkannt und gemeldet werden. Dabei sind verschiedene Strategien zur Wiederaufnahme zu vergleichen.

Bei der Erfüllung dieser Ziele werden folgende Fragestellungen beantwortet:

1. Wie kann das Soll-Verhalten wiederverwendbar beschrieben werden?
2. Wie kann die Überprüfung fortgesetzt werden, nachdem eine Anomalie festgestellt wurde, um alle erkennbaren Abweichungen zu identifizieren?
3. Welche Abweichungen sind erkennbar?
4. Wie und wann kann die Zustands-Explosion für die Fortsetzung der Überprüfung begrenzt werden?

Um diese Fragen zu beantworten, wird in dieser Arbeit ein neuer Ansatz erforscht, um alle erkennbaren Abweichungen zwischen der Ausführung eines System unter Beobachtung (SuO, engl.: *System under Observation*) und dessen Spezifikation mit einem einzigen Monitor zu finden, anstatt nur ungültige Traces zu melden. Dadurch entfällt die Notwendigkeit, die Spezifikation in unabhängige Merkmale aufzuteilen, die einzeln und wiederholt überprüft werden können. Bei komplex integrierten Komponenten ist eine solche Aufteilung oft nicht trivial. Deshalb untersucht diese Arbeit, wie ein Monitor die Überwachung wiederaufnehmen und mehrere Abweichungen erkennen kann. Dieser Ansatz wird im Folgenden *Resumption* genannt.

Diese Arbeit schlägt eine Modellierungsmethodik zur Beschreibung der Spezifikation in sog. geschichteten Referenz-Modellen vor und zeigt ihre Verwendung an mehreren Beispielen. Durch Resumption muss in der Spezifikation nur das erwartete Verhalten definiert werden, um einen Monitor erzeugen zu können, der alle erkennbaren Abweichungen während der Ausführung einer Implementierung findet. Dadurch ist es für den Entwickler einfacher, erkannte Abweichungen zu verstehen, da sie direkt im Kontext der gesamten Spezifikation gesehen werden können. Darüber hinaus erleichtert die Generierung von Monitoren aus der Spezifikation die Konformität des jeweiligen Monitors sicherzustellen. Es werden die Bedingungen untersucht, unter denen Abweichungen erkannt und die Unsicherheit des Ist-Zustandes reduziert werden kann, d. h. wann Resumption erfolgreich sein kann und welche Abweichungen ohne weitere Informationen nicht gefunden werden können. Durch das entwickelte Evaluierungssetup kann für ein Modell untersucht werden, wie sich verschiedene Annahmen über mögliche Abweichungen auf die erzeugten Monitore auswirken. Um einen allgemeinen Überblick über den Einfluss zu geben, wird dies für verschiedene Fehlermodelle und Verhaltensmodelle evaluiert.

1.4 Eigene Veröffentlichungen

Teile dieser Dissertation basieren auf Forschungsarbeiten, die bereits in den folgenden *peer-reviewed* Veröffentlichungen publiziert wurden. Zu jeder Veröffentlichung werden die Beiträge des Autors dieser Arbeit genannt.

1. Christian Drabek, Thomas Pramsohler, Marc Zeller, und Gereon Weiss. „Interface Verification Using Executable Reference Models: An Application in the Automotive Infotainment.“ In ACESMB@Models 2013. Miami, Florida, USA, 2013. [Dra+13]
   In der Veröffentlichung werden die Herausforderungen bei der Überprüfung von Anwendungs-Schnittstellen für Infotainment Systeme in der Automotive Domäne aufgestellt. Zudem werden die Grundideen der geschichteten Referenz-Modelle, ihrer Ausführung zur Überprüfung von Traces und der Resumption vorgestellt. Sie werden in Kapitel 4, Abschnitt 6.4 und Kapitel 5 betrachtet. Das zusammen mit Thomas Pramsohler entwickelte Beispiel (siehe Abschnitt 7.1.1) zeigt, wie mit dem vorgestellten Ansatz Fehler im Kommunikationsverhalten früh entdeckt werden können.

2. Christian Drabek, Annette Paulic, und Gereon Weiss. „Reducing the Verification Effort for Interfaces of Automotive Infotainment Software“. SAE Technical Paper 2015-01-0166. Warrendale, PA: SAE International, 14. April 2015. [DPW15]
   Die Veröffentlichung vertieft den Ansatz, ausführbare Referenz-Modelle für die Verifikation zur Laufzeit wiederzuverwenden und stellt einen implementierten Prototypen des Ansatzes vor.

3. Christian Drabek, Gereon Weiss, und Bernhard Bauer. „Method for Automatic Resumption of Runtime Verification Monitors“. In SOFTENG 2017, 31–36. Venice, Italy, 2017. [DWB17]
   Der Konferenzbeitrag stellt den Ansatz zur automatisierten Erweiterung von Referenz-Modellen mit Resumption vor und vergleicht verschiedene Resumption-Strategien hinsichtlich ihrer Performance für unterschiedliche Arten von Fehlern und Zustandsautomaten. Dies ist die Grundlage des in Abschnitt 7.2 durchgeführten Vergleichs von Resumption-Strategien.

4. Christian Drabek, und Gereon Weiss. „DANA – Description and Analysis of Networked Applications“. In RV-CuBES 2017, 71–60, 2017. [DW17]
   Die Arbeit gibt einen Überblick über den in Kapitel 6 präsentierten Prototypen und die darin implementierten Ansätze. Zudem werden die zwei Anwendungsfälle aus Abschnitt 7.1.3 und Abschnitt 7.1.4 vorgestellt.

5. Christian Drabek, Gereon Weiss, und Bernhard Bauer. „Resumption of Runtime Verification Monitors: Method, Approach and Application“. International Journal On Advances in Software 11, Nr. 1 and 2 (30. Juni 2018): 18–33. [DWB18]
   Der Journalbeitrag untersucht die Theorie hinter Resumption, die in Kapitel 5 betrachtet und für die erweiterten Modellierungskonzepte ergänzt wird. Die dadurch erziele Verbesserung der Resumption-Strategien wird anhand einer aktualisierten Evaluation gezeigt.

1.5 Struktur der Arbeit

Dieser Abschnitt gibt einen Überblick über den Aufbau der restlichen Arbeit. Die Struktur ist in Abbildung 1.1 zusammengefasst.

In Kapitel 2 wird der Ansatz in die modellbasierte Entwicklung eingeordnet. Anschließend wird auf die Eigenschaften von Automaten und deren Notation eingegangen und das Themengebiet Laufzeitüberprüfung vorgestellt.

Kapitel 3 ordnet den Ansatz in den aktuellen Stand der Technik ein. Dazu werden bestehende Ansätze betrachtet und diese Arbeit von ihnen abgegrenzt.

Kapitel 4 stellt die entwickelte, modulare Modellierungsmethodik zur Beschreibung von Kommunikationsverhalten vor. Es wird ein Überblick über das geschichtete Referenz-Modell gegeben, bevor die einzelnen Teile zur Beschreibung von Topologie, Schnittstellen, Ereignissen und Verhalten erläutert werden. Erweiterungen gegenüber einfachen Automaten werden eingeführt, mit denen sich beispielsweise neben sequentiellen Abhängigkeiten

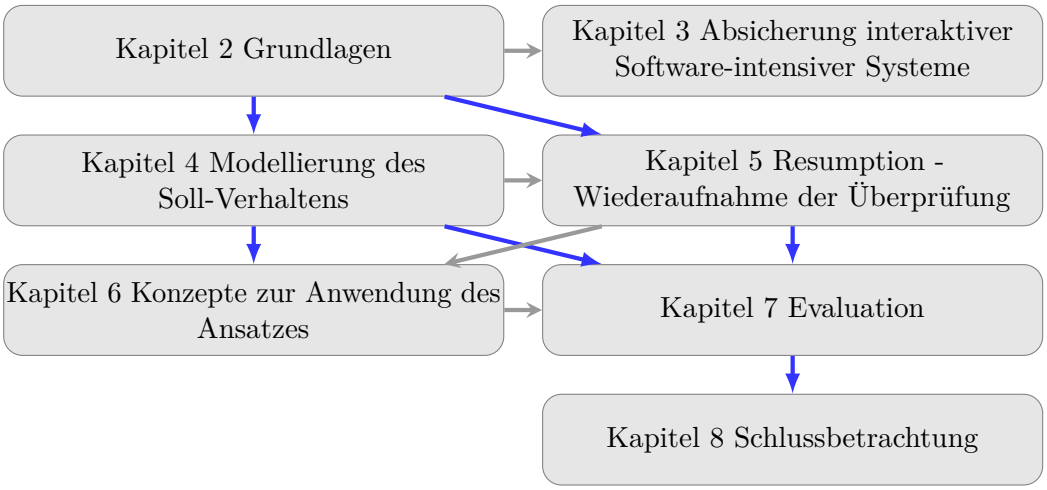


Abb. 1.1: Aufbau der vorliegenden Arbeit

des Kontrollflusses auch die Zusammenhänge von Daten der Nachrichten beschreiben lassen.

In Kapitel 5 wird der neue Ansatz zur Wiederaufnahme eines Monitors, der auf einer Beschreibung des Soll-Verhaltens basiert, präsentiert. Dieser Ansatz wird *Resumption* genannt und es wird untersucht, wie ein Monitor mehr als nur den ersten Verstoß gegen eine Spezifikation erkennen kann. Zudem wird untersucht, welche Abweichungen erkannt werden können und wann dies nicht ohne weitere Informationen möglich ist. Da komplexere Modellierungmethoden zur (mehrfach) exponentiellen Zunahme an Zuständen führen können, wird in der zweiten Hälfte von Kapitel 5 Rahmenbedingungen betrachtet, unter denen der Zustandsraum für Resumption vereinfacht werden kann.

Kapitel 6 gibt einen Überblick über weitere Konzepte zur Realisierung des in dieser Arbeit vorgestellten Ansatzes und ihre Integration in die Werkzeugplattform Dana.

Kapitel 7 evaluiert den vorgestellten Ansatz. Dort werden Nutzen und Anwendbarkeit in verschiedenen Anwendungsbeispielen demonstriert. Durch den Vergleich unterschiedlicher Strategien zur Wiederaufnahme der Überprüfung wird untersucht, ob auf Basis solcher Soll-Beschreibungen alle beobachtbaren Abweichungen erkannt werden.

Die Schlussbetrachtung in Kapitel 8 fasst die Beiträge der Arbeit zusammen und schließt mit einem Ausblick.

In dieser Arbeit werden, soweit sinnvoll und vorhanden, deutsche Bezeichnungen verwendet. Zu einigen Themengebieten existiert einschlägige Vergleichsliteratur nur auf Englisch. Übliche Abkürzungen hängen häufig mit den englischen Begriffen zusammen. In solchen Fällen wird bei der Einführung von Fachbegriffen auch die englische Bezeichnung aufgeführt.

2 Kapitel 2 Grundlagen

Dieses Kapitel stellt die Grundlagen vor, auf denen der präsentierte Ansatz zur modellbasierten Beschreibung und Überprüfung des Verhaltens vernetzter eingebetteter Systeme aufbaut. Zunächst werden sowohl die Grundzüge von modellbasierter Entwicklung vorgestellt, als auch verschiedene Modellierungssprachen. Anschließend werden Automaten, ihre Eigenschaften und die in dieser Arbeit verwendete Notation eingeführt. Abschließend wird ein Überblick über das Themengebiet Laufzeitüberprüfung (RV, engl.: *Runtime Verification*) gegeben.

2.1 Modellbasierte Entwicklung

Dieser Abschnitt stellt den Bereich modellbasierter Entwicklung vor, um diese Arbeit einzuordnen. Mit modellbasierter Entwicklung wird häufig intuitiv die Verwendung von Diagrammen statt Code assoziiert [Sch+02]. Beispielsweise werden Klassendiagramme für die Modellierung von Daten verwendet und Zustands- oder Blockdiagramme, um Verhalten und Funktionen zu spezifizieren. So gibt es Werkzeuge verschiedener Hersteller, die dafür grafische Editoren bereitstellen. Beispielsweise wird MATLAB/Simulink [MATLAB] zum Modellieren und Testen von kontinuierlichen Systemen und Steuerungsfunktionen verwendet. Für interaktive Systeme, wie sie z. B. im automobilen Infotainment häufig auftreten, sind von Ereignissen getriebene und auf Zuständen basierende Beschreibungen vorherrschend. Beispiele für dort verwendete CASE-Werkzeuge sind Enterprise Architect [EA] oder Rational Rhapsody [RHAPSODY], mit denen die Modelle zur Spezifikation und Code-Generierung erstellt werden. Als Open Source Werkzeug-Plattform verbreitet sich Eclipse [ECLIPSE] in der Automobilbranche und wird durch Erweiterungen wie ARTOP [Knü+10] oder EATOP [EATOP] ergänzt und so an die Bedarfe der Domäne angepasst.

Stahl u. a. [Sta+07] setzen die modellgetriebene Softwareentwicklung von der modellbasierten ab: „Modelle sind hier nicht nur Dokumentation, sondern sie sind gleichzusetzen

mit Code – die Umsetzung ist automatisiert.“ Sie sehen zwei grundlegende Ansätze, aus Modellen ausführbare Software zu erzeugen: Generatoren, die aus dem Modell Quelltext einer anderen Programmiersprache erzeugen, und Interpreter, die ein Modell zur Laufzeit einlesen und abhängig vom Inhalt verschiedene Aktionen ausführen. Ihr „Kerngedanke ist dabei, dass die Modelle die Rolle der Quelltexte einnehmen und generierte Artefakte nur eine Art temporäre Build-Zwischenprodukte sind. Änderungen werden an den Modellen vorgenommen, so dass der generierte Code immer einheitlich ist und die Modelle automatisch den aktuellen Stand widerspiegeln“ [Sta+07].

Schätz u. a. [Sch+02] sehen als Schlüssel für bessere Qualität und effizientere Prozesse die Integration von verschiedenen Modellen mit ggf. unterschiedlichen Abstraktionsgraden. Modellbasierte Entwicklung verwendet deshalb die Prinzipien Abstraktion und Restriktion. Sowohl im Prozess- als auch im Produkt-Modell unterscheidet er Abstraktionen auf horizontaler und vertikaler Ebene. Horizontal wird zwischen verschiedenen Aspekten unterschieden, beispielsweise Kommunikation, Zeitverhalten oder Sicherheit. Vertikal bezieht sich auf Verfeinerungen, beispielsweise der Struktur oder von Funktionen, die mit Fortschritt der Entwicklung einhergehen. Modellbasierte und modellgetriebene Entwicklung sind immer domänenspezifisch [Sch+02; Sta+07]. Auch wenn generische Ansätze wiederverwendet werden, ist häufig eine individuelle Anpassung notwendig. Dies wird beispielsweise an der Anpassbarkeit des V-Modells XT [VModellXT] an die jeweiligen Bedarfe deutlich. Aber auch dies kann noch nicht alle Anwendungsfälle vollständig abdecken. Deshalb gibt es regelmäßige Aktualisierungen des V-Modells XT und auch Vorschläge zur Erweiterung, z. B. stellen Hillemacher u. a. [Hil+18] eine Erweiterung des V-Modells vor, die eine modellbasierte Entwicklung von selbst-adaptiven autonomen Fahrzeugen ermöglichen soll.

Auch bei genauer Verfolgung der vordefinierten Prozesse gibt es immer noch viele Unwägbarkeiten in der Praxis, die zu Abweichungen zwischen Modell und Implementierung führen können [Sch+02]. Das kann u. a. daran liegen, dass bestimmtes domänenspezifisches Wissen nur schwer formalisierbar oder sehr detailliert ist. Selbst wenn dies formalisiert und integriert wird, kann es pragmatische Probleme geben, wie *Legacy-Code*, kundenspezifische Kodierungen und Zertifizierungsstandards oder einfach Eigenarten von Compilern oder Betriebssystemen, die zu Abweichungen vom Ideal des Modells führen. Nach Schätz u. a. können diese Probleme durch Einbeziehen der Experten aus der jeweiligen Domäne gelöst werden. Um zu erkennen, wo Abweichungen im Verhalten des Systems vorliegen, kann eine genaue Überwachung helfen. Dies soll durch den in dieser Arbeit verfolgten Ansatz ermöglicht und vereinfacht werden.

Viele verschiedene Modellierungssprachen sind im Laufe der Zeit definiert und verwendet worden, um interaktive, verteilte Systeme zu beschreiben. Im Folgenden werden die

Sprachen beschrieben, die im Rahmen dieser Arbeit verwendet wurden oder die Modellierung beeinflusst haben. Diese lassen sich grob in domänenspezifische Sprachen (DSLs, engl.: *Domain Specific Language*) und universale Sprachen (GPLs, engl.: *General Purpose Language*) einteilen, auch wenn dies nicht immer trennscharf ist. Entsprechend ihrem Namen fokussieren sich erstere auf einen bestimmten Einsatzzweck und letztere bieten vielseitige Möglichkeiten, um nahezu beliebige Anwendungsgebiete abzudecken. Voelter [Voe13] beschreibt die Bestandteile der Definition einer Sprache wie folgt. Die *konkrete Syntax* beschreibt die Notation, mit der ein Nutzer textuell, grafisch oder tabellarisch Programme definiert. Die *abstrakte Syntax* ist eine Datenstruktur, um die semantisch relevanten Informationen zu speichern. Die *statische Semantik* beschreibt zusätzliche Bedingungen oder Regeln, die Programme befolgen müssen. Die *operative Semantik* erklärt die Bedeutung der Ausführung des Programms.

2.1.1 Universale Sprachen

GPLs zielen darauf ab, den gesamten Software-Entwicklungsprozess zu begleiten und zu unterstützen. Prinzipiell gibt es dafür zwei mögliche Ansätze, die häufig als Mischung umgesetzt werden: Entweder ist die Sprache sehr mächtig oder sehr flexibel. Im ersten Fall bietet sie viele Möglichkeiten, um verschiedene Aspekte zu beschreiben. Dies bedeutet aber auch, dass der gleiche Sachverhalt auf viele verschiedene Arten modelliert werden kann. Den Personen, die mit diesen Modellen umgehen, müssen alle verwendeten Varianten bekannt sein, damit sie die Modelle richtig verstehen können. Dies gilt natürlich für jede Sprache, ob nun universal oder domänenspezifisch. Erstere sind aber von Natur aus umfangreicher, sodass es leichter zu Missverständnissen kommen kann. Flexible Sprachen befähigen die Benutzer vielmehr dazu, eine Basissprache für den gewünschten Zweck zu erweitern oder einzuschränken. Eine solche Sprache kommt normalerweise mit entsprechender Werkzeugunterstützung und wird deshalb auch als *Sprachen-Werkbank* bezeichnet. Sie dient als Baukasten zur Definition und Verknüpfung eigener DSLs. Auch allgemeine Programmiersprachen wie beispielsweise *C* oder *Java* können als GPL eingeordnet werden.

2.1.1.1 UML

Die Unified Modeling Language (UML) ist dem Namen nach eine vereinheitlichte oder vereinigende Modellierungssprache. Sie will Systemarchitekten, Software Ingenieuren und Software Entwicklern Werkzeuge für Analyse, Entwurf und Implementierung von Software-basierten Systemen sowie für die Modellierung von Geschäfts- und anderen Prozessen bereitstellen [OMG-UML]. Sie wird von der Object Management Group (OMG) regelmäßig aktualisiert.

Eines der Hauptziele von UML ist eine Interoperabilität zwischen Modellierungswerkzeugen zu schaffen und dadurch den Stand der Industrie voranzubringen. Um einen sinnvollen Austausch von Modellinformationen zwischen Werkzeugen zu ermöglichen, ist eine Absprache bezüglich Semantik und Syntax notwendig. Aus UML wurde deshalb ein Rahmen zur Beschreibung von Meta-Architekturen abgeleitet, die Meta Object Facility (MOF) [OMG-MOF]. Eine essenzielle Untermenge die sich an den Möglichkeiten von objektorientierten Programmiersprachen und XML orientiert, genannt Essential MOF (EMOF), vereinfacht eine direkte Abbildung von MOF Modellen.

Durch den starken Fokus auf Syntax und statische Semantik sowie die große Vielfalt an Beschreibungsmöglichkeiten wurde allerdings die operative Semantik vernachlässigt und häufig nicht vollständig geklärt. An dieser Stelle setzt foundational UML (fUML) [OMG-FUML] an; dies wird durch eine präzise Semantik für Zustandsautomaten, die Precise Semantics of UML State Machines (PSSM) [OMG-PSSM], ergänzt. Beide schränken Teile der UML ein, um undefinierte oder mehrdeutige Definitionen zu verhindern, und beschreiben die operative Semantik der verbliebenen Elemente. Auch wenn es sich dabei formal nicht um Profile handelt, entspricht dieses Vorgehen der Definition von Profilen für UML und wird auch in den Spezifikationen von OMG damit verglichen. Profile bieten die Möglichkeit UML auf eine Untermenge zu beschränken sowie in gewissem Rahmen Erweiterungen für existierende Elemente, beispielsweise zusätzliche Attribute, zu definieren. Damit kann innerhalb von UML eine DSL definiert werden. Da der Syntax der Elemente durch dieser Erweiterungen komplizierter wird, hängt die Benutzbarkeit der DSL dann stark von der Unterstützung durch den verwendeten UML Editor ab. Hier liegt aber auch ein potenzieller Vorteil gegenüber klassischen DSLs, da die so geschaffene Sprache nicht an einen bestimmten Editor gebunden ist.

Abhängig davon, wie die Trennung zwischen DSL und GPL definiert wird, kann UML auch als eine Sammlung von DSLs gesehen werden, die unterschiedliche Aspekte von Systemen beschreiben [Voe13]: beispielsweise Klassenstrukturen oder auf Zuständen basiertes Verhalten. Voelter ordnet sie aber eher als GPL ein, da die verschiedenen Diagramme dennoch die Domäne *Software* im Ganzen adressieren.

2.1.1.2 EMF

Das Eclipse Modelling Framework (EMF) [Ste+08] bietet einen Rahmen zur Modellierung in Eclipse, einer offenen Entwicklungsplattform [ECLIPSE]. Es stellt dazu *Ecore* bereit, eine Sprache zur Beschreibung von Meta-Architekturen. Sie ist die de facto Referenz-Implementierung für EMOF [hs10]. Auf ihr aufbauend können somit verschiedene Modelle, beispielsweise auch UML, beschrieben werden. EMF bietet zudem einen Basis-Satz an Werkzeugen, um aus Ecore z. B. einfache Editoren oder Code zur Repräsentation der

abstrakten Syntax zu erzeugen. Um diesen Kern herum hat sich eine breite Community gebildet, die vielfältige Werkzeuge bereitstellen, mit denen die verschiedenen Bestandteile einer Modellierungssprache auf unterschiedliche Weise realisiert und weiterverwendet werden können. Beispielsweise ermöglicht *XText* [XTEXT] die Definition von textuellen DSLs [Bet16]. *PapyrusUML* [Lan+09] stellt einen grafischen Editor und andere Werkzeuge zur Bearbeitung und Ausführung von UML bereit. Mit *Sirius* [VMP14] kann unkompliziert ein grafischer Editor für die eigene DSL umgesetzt werden. Durch die gemeinsame Basis auf Ecore können sie fast beliebig kombiniert werden und laden somit ein, modellbasierte Entwicklung an die eigenen Bedarfe anzupassen. Wie gut das funktionieren kann, zeigen viele der Eclipse Projekte bereits selbst, indem sie die Modelle sowohl zur Entwicklung als auch zur Laufzeit verwenden.

2.1.2 Domänenspezifische Sprachen

DSLs werden von Fowler [Fow11] als Programmiersprachen mit begrenzter Ausdruckskraft definiert, die auf eine bestimmte Domäne fokussiert sind. Weiterhin beschreibt er, dass sie damit Wege bieten, um Abstraktionen zu manipulieren. Bei der Software-Entwicklung werden solche Abstraktionen aufgebaut und oft auf verschiedenen Ebenen editiert. Er vergleicht DSLs deshalb mit einem Front-End zu den Bibliotheken die diese Abstraktionen realisieren bzw. zu deren semantischem Modell.

Im Folgenden werden DSLs vorgestellt, die für den vorgestellten Ansatz relevant sind. Hierbei sei angemerkt, dass die Domäne, auf die sie zielen, nicht das Einsatzgebiet einer entwickelten Anwendung ist, sondern ein bestimmter Aspekt bei der Entwicklung. *Franca IDL* beschreibt textuell die Elemente von Schnittstellen zwischen Komponenten eines Systems. *TADL* dient der Definition von Zeitverhalten mit Modellen. *Statecharts* gelten als Vorlage für viele moderne Beschreibungsformen von Zustandsautomaten. Darunter fallen beispielsweise auch *SCXML*, eine Sprache zur Beschreibung von ausführbaren Zustandsautomaten mit XML, und die Zustandsdiagramme von UML.

2.1.2.1 Franca IDL

Die Schnittstellenbeschreibungssprache (IDL, engl.: *Interface Definition Language*) *Franca IDL* definiert sich selbst als Sprachen-neutral und unabhängig von konkreten Bindings [Franca]. Als Binding wird eine Implementierung der Schnittstelle für ein bestimmtes Kommunikationsmedium bezeichnet. Schnittstellen, die mit Franca IDL definiert werden, können aus Attributen, Methoden und Broadcasts bestehen. Dabei dürfen vorgegebene, primitive Datentypen (z. B. Int16 oder String) verwendet werden oder selbst definierte Typen unter

Verwendung von Arrays, Strukturen, Enumerationen, Aliase, Abbildungen und anderen Elementen. Eine Spezifikation mit Franca IDL kann in mehrere Dateien aufgeteilt werden. Dies ist insbesondere nützlich, falls mehrere Schnittstellen definiert werden, die einige gemeinsame Datentypen und Strukturen teilen. Zusätzlich kann das Verhalten der Schnittstelle mit einem einfachen Zustandsautomaten beschrieben werden.

Franca IDL ist mit XText [Bet16] realisiert. Das bedeutet, sie verwendet eine konkrete Syntax, die textuell mit Schlüsselwörtern formuliert wird. Die abstrakte Syntax ist ein Baum von EMF-Objekten, die weiterverwendet werden können. Die Franca IDL ist dazu Teil eines gleichnamigen Frameworks, das Bausteine zur Verarbeitung der IDL anbietet. Beispielsweise werden die statische und dynamische Semantik über Validierungsregeln und Code-Generierung umgesetzt. Letztere erzeugt vom Binding unabhängige Methoden-Rümpfe für die beschriebene Schnittstelle, die in der Applikation verwendet werden können, um über sie zu kommunizieren. Das Binding für ein bestimmtes Kommunikationsmedium wird über einen spezialisierten Code-Generator erzeugt [z. B. Nik+19; De+19]. Dieser kann zur genaueren Konfiguration als zusätzliche Eingabe ein sog. Deployment-Modell erhalten, das spezifisch für ein Binding ist. Dieses wird über eine weitere DSL vom Binding deklariert und für den Use-Case definiert. Diese Aufteilung gewährleistet die Unabhängigkeit von Franca IDL und ermöglicht gleichzeitig eine strukturierte Beschreibung aller benötigten, beliebig spezifischen Parameter. In Franca+ [SBS20] wurde die IDL für Komponenten und Port-basierte Kommunikation erweitert.

2.1.2.2 TADL

Die Zeitverhalten-erweiterte Beschreibungssprache (TADL, engl.: *Timing Augmented Description Language*) ermöglicht es, Design-Modelle, die z. B. mit Modellierungssprachen wie EAST-ADL [Mub+17] oder AUTOSAR [Klo+10] erstellt wurden, um eine Beschreibung des vorhandenen oder benötigten Zeitverhaltens zu ergänzen [TIM09]. Dazu ermöglicht TADL die Beschreibung von Bedingungen für das Zeitverhalten, basierend auf von dem System exponierten Ereignissen. Das System-Design kann durch andere Modellierungssprachen in unterschiedlichen Detailgraden beschrieben werden.

Für die Anwendbarkeit von TADL ist dabei wichtig, dass diese beobachtbare Ereignisse definieren. Sie beschreibt damit die extern überprüfbaren Anforderungen an das Zeitverhalten des Systems. Durch die einfache, aber klar definierte Schnittstelle zur restlichen Modellierungssprache – die beobachtbaren Ereignisse – ist eine Adaption von TADL für andere Sprachen, insbesondere eine Wiederverwendung der Semantik, leicht möglich.

Die konkrete und abstrakte Syntax von TADL wurde in EAST-ADL integriert. Eine aktualisierte Version [TIM12] erweitert TADL um symbolische Zeitausdrücke und probabilistisches Zeitverhalten. Ebenso beschreibt der Projektbericht [TIM12] eine Abbildung auf die *Timing Extensions* von AUTOSAR. Auch die Semantik wird in den Projektberichten [TIM09; TIM12] definiert. Verschiedene Fallstudien [z. B. Per+12; Klo+10] zeigen die Anwendbarkeit in der Praxis.

2.1.2.3 Statecharts

Harel [Har87] beschreibt einen Ansatz zur Modellierung von Zustandsautomaten, der versucht, auf natürliche Weise eine Unterstützung für allgemeine und flexible Aussagen zu integrieren und nennt diese *Statecharts*. Er hat damit die Grundlage für viele nachfolgende Ansätze gelegt. Statecharts führen eine und/oder-Dekomposition von Zuständen mit Ebenen-übergreifenden Transitionen ein und einen Mechanismus zur Kommunikation zwischen verschiedenen nebenläufigen Komponenten. Damit sollen eine Reihe von Konzepten ermöglicht werden, die Harel für die Modellierung realer reaktiver Systeme als notwendig erachtet: die Gruppierung von Zuständen in einem übergeordneten Zustand, die Beschreibung von unabhängigen oder orthogonalen Abläufen, die Verallgemeinerung von Pfeilen mit einzelnen Ereignissen zu komplexen Transitionen und die Verfeinerung von Zuständen.

Er beschreibt auch weitere Modellierungskonzepte, die eine Beschreibung reaktiver Systeme erleichtern sollen. Beispielsweise können Zeitschranken an einem Zustand annotiert werden. Parametrisierte Zustände sollen helfen, viele ähnliche Zustände zusammenzufassen, wenn sie sich nur in ihren Parametern unterscheiden. Überlappende Zustände sollen das mehrfache Modellieren der enthaltenen Zustände vermeiden können, wenn das gleiche Verhalten in verschiedenen orthogonalen Konstellationen auftreten kann.

In seiner Arbeit gibt Harel hauptsächlich eine Beschreibung des Ansatzes auf visueller Ebene in einem Diagramm und verweist bei der Diskussion ihrer Semantik teilweise auf einige andere Arbeiten oder lässt diese bewusst frei. Viele neuere Methoden zur Modellierung von Zustandsautomaten übernehmen einige der dort gesammelten Konzepte oder interpretieren sie neu. Entsprechend verfolgen die abgeleiteten Sprachen unterschiedliche Ansätze, wie sie die Semantik formal beschreiben [z. B. MLS97; LM01; HK04; SCXML15; Sam16; OMG-PSSM]. Crane und Dingel [CD07] haben die Unterschiede zwischen einigen Ansätzen untersucht. Beispielsweise werden in UML-Zustandsdiagrammen Konflikte zwischen Transitionen eher objektorientiert aufgelöst, d. h. die innerste Transition hat Priorität, während in klassischen Statecharts die in der Hierarchie höchste präferiert wird.

State Chart XML (SCXML) [SCXML15] basiert ebenfalls auf den Konzepten von Statecharts, verzichtet aber völlig auf die Definition einer visuellen Beschreibung. Wie aus dem Namen bereits hervorgeht, beschreibt SCXML eine XML-basierte Syntax und definiert dafür eine präzise operative Semantik. Da sich dies als Vorlage für die Implementierung einer generischen Ausführung von Statecharts anbietet, gibt es diverse Interpreter für SCXML in verschiedenen Programmiersprachen [SCXML].

2.2 Automatentheorie

Zustandsautomaten, im Folgenden häufig kurz als Automaten bezeichnet, werden bereits seit langer Zeit herangezogen, um das Verhalten von Systemen zu beschreiben und bestimmte Eigenschaften daraus abzuleiten und zu beweisen [Har87]. Dieser Abschnitt stellt die in der vorliegenden Arbeit verwendete Formalisierung von Automaten vor.

2.2.1 Automaten

Ein Zustandsautomat ist ein Graph aus Knoten und gerichteten Kanten. Die Knoten repräsentieren Zustände (engl.: *states*) und die Kanten sind Zustandsübergänge, auch als Transitionen (engl.: *transitions*) bezeichnet, zwischen diesen. Ein Beispiel für die grafische Darstellung gibt Abbildung 2.1. Zustände werden mit Kreisen dargestellt. In ihnen steht der Name des Zustands. Für Zustandsübergänge werden Pfeile verwendet. Diese haben eine Beschriftung, die, sofern nicht anders bestimmt, das auslösende Ereignis bezeichnet.

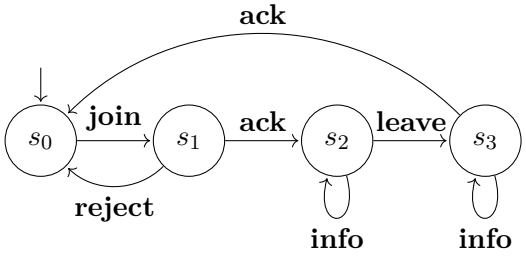


Abb. 2.1: Beispiel der grafischen Darstellung eines Zustandsautomaten.

Formal wird ein einfacher Automat dargestellt als ein 4-Tupel $A = \langle \mathbb{B}, \mathbb{S}, s_0, \delta \rangle$.

- $\mathbb{B}$ ist die Menge möglicher Beschriftungen, beispielsweise die Menge der Eingaben $\mathbb{E}$.
- $\mathbb{S}$ ist die Menge der Zustände des Automaten, inklusive eines initialen Zustands s_0.
- $\delta \subseteq \mathbb{S} \times \mathbb{B} \to \mathbb{S}$ ist die Relation der Transitionen.

Man spricht von einem deterministischen Zustandsautomaten, wenn für jede Kombination aus Zustand und Eingabe maximal eine Transition existiert, d. h. wenn δ eindeutig ist. Der Automat wird als vollständig bezeichnet, wenn für jede der Kombinationen eine Transition definiert ist. Er ist endlich, wenn seine Zustandsmenge eine endliche Größe besitzt. Sonst ist der Automat entsprechend nicht-deterministisch, unvollständig oder unendlich. Die Größe eines Zustandsautomaten wird definiert als $|A| = |\mathbb{S}| + |\delta|$.

2.2.2 Erweiterte Notation für Automaten

In dieser Arbeit werden e_i und s_i verwendet, um allgemein auf Eingaben und Zustände als Bestandteil eines Zustandsautomaten Bezug zu nehmen. Dabei bezeichnet s_0 immer den initialen Zustand. Da in dieser Arbeit nicht nur statisch auf die Elemente des Zustandsautomaten Bezug genommen wird, sondern auch in Abhängigkeit von der Sequenz während einer konkreten Ausführung des beobachteten Systems, wird auch eine zweite Notation für die Eingaben und Zustände verwendet: a_i und q_i. Für eine übersichtliche Schreibweise werden in dieser Arbeit die Funktion zur Benennung $e(a_i)$ und $s(q_i)$ ausgespart und direkt a_i und q_i als Kurzform für die Elemente verwendet. Daraus ergibt sich, dass ein *Trace* v eine Sequenz der Eingaben $a_1 a_2 \dots a_n = v \in \mathbb{E}^*$ ist, für die der Automat die Sequenz der Zustände $q_1 \dots q_{n+1}$ durchläuft. Eine Eingabe als Teil eines Traces wird auch Ereignis genannt. Die Relation der Transitionen wird wie üblich [LY96; San05] erweitert, damit sie Traces (2.1) und Mengen mit (möglichen) Zuständen (2.2) akzeptiert.

$$\delta(q_1, v) = \delta(\delta(q_1, a_1 \dots a_{n-1}), a_n) = \delta(q_n, a_n) = q_{n+1} \tag{2.1}$$

$$\delta(Q, v) = \{s \in \mathbb{S} \mid \exists q \in Q : \delta(q, v) \mapsto s\} \tag{2.2}$$

Im Allgemeinen muss δ nicht vollständig definiert sein. Das bedeutet, es gibt Kombinationen aus Zustand und Eingabe, für die kein Zielzustand definiert wurde. Das Verhalten des Automaten ist in diesem Fall undefiniert. Um den Bezug auf Elemente mit einer definierten Zuordnung zu formalisieren, sei $\mathrm{dom}(\delta)$ die Domäne der Teilfunktion δ, d. h. die Menge der Eingaben und Zustände mit einer definierten Zuordnung. Weiterhin sei $\mathbb{E}^s$ die Menge der Eingaben mit einem definierten Zustandsübergang im Zustand s (2.3) und $\mathbb{S}^e$ sei die Menge der Zustände mit einem definierten Zustandsübergang für die Eingabe e (2.4). Mit Bezug auf den Automaten aus Abbildung 2.1 sind beispielsweise $\mathbb{E}^{s_1} = \{\mathrm{ack}, \mathrm{reject}\}$ und $\mathbb{S}^{\mathrm{ack}} = \{s_1, s_3\}$.

$$\mathbb{E}^s = \{e \in \mathbb{E} \mid \langle s, e\rangle \in \mathrm{dom}(\delta)\} \tag{2.3}$$

$$\mathbb{S}^e = \{s \in \mathbb{S} \mid \langle s, e\rangle \in \mathrm{dom}(\delta)\} \tag{2.4}$$

2.2.3 Hierarchische Automaten

Wie bereits bei den *Statecharts* von Harel [Har87] festgestellt wurde, erleichtern gewisse Strukturen in Automaten deren Verwendung in der Praxis. Im Kern ist damit die Möglichkeit gemeint, Zustände hierarchisch anzuordnen. Damit können verschiedene Detailniveaus des Verhaltens als auch weitestgehend unabhängig verlaufende Verhaltensstränge kompakt beschrieben werden. Dieser Abschnitt stellt eine Formalisierung von hierarchischen Automaten vor, basierend auf dem Ansatz von Mikk, Lakhnechi und Siegel [MLS97], angepasst an die in dieser Arbeit verwendete Notation.

Ein hierarchischer Automat ist ein 3-Tupel $HA = \langle \mathbb{A}, \mathbb{E}, \alpha \rangle$. Dabei beschreibt $\mathbb{A} = \{A_1, A_2, \ldots\}$ eine Menge sequentieller Automaten. Das sind einfache Automaten, wie sie bereits in Abschnitt 2.2.1 vorgestellt wurden. Es wird angenommen, dass sie paarweise disjunkte Zustandsräume $\mathbb{S}_{A_i}$ verwenden und $\mathbb{S} = \bigcup_{A_i \in \mathbb{A}} \mathbb{S}_{A_i}$ sei die Menge aller Zustände. $\mathbb{E}$ ist die Menge aller Eingaben. Eine Kompositionsfunktion $\alpha : \mathbb{S} \mapsto \mathcal{P}(\mathbb{A})$ beschreibt für jeden Zustand, welche Automaten ihm untergeordnet sind und muss diese drei Bedingungen erfüllen: Sie bestimmt genau einen Wurzelautomaten $\sqrt{\alpha}$, der keinem Zustand zugeordnet ist (2.5); jeder nicht Wurzelautomat wird immer genau einem anderen Zustand zugeordnet (2.6); und sie ist frei von Zyklen (2.7).

$$\sqrt{\alpha} \notin \bigcup_{s_i \in \mathbb{S}} \alpha(s_i) \tag{2.5}$$

$$\bigcup_{s_i \in \mathbb{S}} \alpha(s_i) = \mathbb{A} \setminus \{\sqrt{\alpha}\} \wedge \forall A \in \mathbb{A} \setminus \{\sqrt{\alpha}\}.\exists_1 s \in \mathbb{S} \setminus \mathbb{S}_A.A \in \alpha(s) \tag{2.6}$$

$$\forall S \subseteq \mathbb{S}.\exists s \in S.S \cap \bigcup_{A \in \alpha(s)} \mathbb{S}_A = \emptyset \tag{2.7}$$

Ein Zustand s wird durch genau einen Automaten verfeinert, wenn $|\alpha(s)| = 1$, und durch eine parallele Komposition von Automaten, wenn $|\alpha(s)| > 1$. Anderenfalls ist $|\alpha(s)| = 0$ und s wird als *einfacher Zustand* bezeichnet. Durch α wird eine Relation $s \blacktriangleright s'$ zwischen jedem Zustand und den Zuständen der Automaten, die ihn verfeinern, induziert (2.8), sowie ihre irreflexive transitive Hülle $\overset{+}{\blacktriangleright}$ (2.9).

$$s \blacktriangleright s' \iff s' \in \bigcup_{A \in \alpha(s)} \mathbb{S}_A \tag{2.8}$$

$$s \overset{+}{\blacktriangleright} s'' \iff s \blacktriangleright s'' \vee (s \overset{+}{\blacktriangleright} s' \wedge s' \blacktriangleright s'') \tag{2.9}$$

Die gleichzeitig in HA aktiven Zustände werden genau dann als seine Konfiguration $K \subseteq \mathbb{S}$ bezeichnet, wenn sie genau einen Zustand des Wurzelautomaten $\sqrt{\alpha}$ enthält (2.10) und sie entsprechend der durch α induzierten Hierarchie abgeschlossen ist (2.11). Das heißt für jeden

Automaten, der einen Zustand der Konfiguration verfeinert, wird genau ein Zustand als aktiv bestimmt. Die Menge aller Konfigurationen wird mit $\mathbb{K}$ bezeichnet.

$$\exists_1 s \in \mathbb{S}_{\sqrt{\alpha}}.s \in K. \tag{2.10}$$

$$\forall s \in K.\forall A \in \alpha(s).\exists_1 s' \in \mathbb{S}_A.s' \in K. \tag{2.11}$$

In *HA* sind verschiedene Automaten gleichzeitig aktiv. Um Abhängigkeiten zwischen ihnen zu beschreiben und eine interne Kommunikation zu ermöglichen, werden die Beschriftung in A erweitert zu 4-Tupeln $\langle sr, e, ac, td\rangle$. Dabei ist $sr \subseteq \mathbb{S}$ eine Menge von Zuständen, die aktiv sein müssen, e eine Eingabe, $ac \subseteq \mathbb{E}$ ist eine Menge von auszulösenden (internen) Ereignissen und td ist die Menge von zu aktivierenden Unterzuständen des Zielzustands. Eine Transition $\langle s, \langle sr, e, ac, td\rangle, s'\rangle$ akzeptiert $\langle K, \mathbb{E}\rangle$, wenn $(\{s\} \cup sr) \subseteq K$ und $e \in \mathbb{E}$. Ebenso wird dann $\langle K, \mathbb{E}\rangle$ von dem die Transition enthaltenden Automat A akzeptiert. Die Ausführung der Transition löst die Ereignisse in ac aus und aktiviert die Zustände in td.

Ein Ausführungsschritt von *HA* für $\langle K, \mathbb{E}\rangle$ besteht aus der synchronen Ausführung aller Transitionen in einer sogenannten *maximalen, konfliktfreien Menge an Transitionen*. Wie die Bezeichnung bereits andeutet, kann es Konflikte zwischen akzeptierenden Transitionen geben, wenn mehrere den gleichen Ursprungszustand verändern würden. Dies kann durch eine Priorisierung der Transitionen aufgelöst werden. Die Menge ist maximal, wenn sie alle Transitionen mit der höchsten Priorisierung enthält. Mikk, Lakhnechi und Siegel [MLS97] schlagen vor, diese Priorisierung an die Hierarchie zu binden und wie in Statecharts werden Transitionen auf einer höheren Ebene bevorzugt. Wenn s_1 und s_2 die Zustände sind, von denen zwei im Konflikt stehende Transitionen ausgehen und $s_1 \overset{+}{\blacktriangleright} s_2$, dann hat die Transition von s_1 gegenüber der von s_2 Vorrang. Wie Mikk, Lakhnechi und Siegel durch Definition einer Kripke-Struktur zeigen, hat dies den Vorteil einer einfachen deklarativen Beschreibung der Semantik.

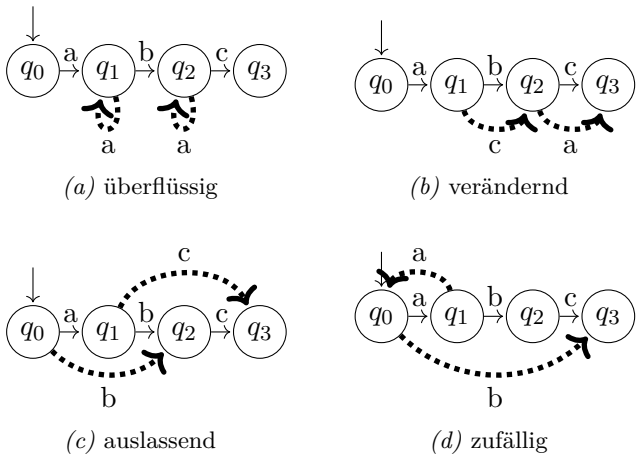
Es sei angemerkt, dass *HA* auf einen einfachen Automaten A abgebildet werden kann, der für jede Konfiguration in $\mathbb{K}$ einen Zustand enthält, die durch entsprechende Transitionen verbunden sind. Dies funktioniert analog zu der von Mikk, Lakhnechi und Siegel [MLS97] vorgestellten Abbildung auf eine Kripke-Struktur. Durch Berücksichtigung der Priorisierung der Transitionen bei der Abbildung kann in vielen Fällen dabei auf die erweiterte Beschriftung verzichtet werden. Jeder Zustand von A gibt bereits vor, welche Zustände in *HA* aktiv sind, wodurch sr ausgewertet und unpassende Transitionen gefiltert werden können. Mögliche weitere, von ac ausgelöste Transitionen können zu einer Transition zusammengefasst werden, die direkt den endgültigen Zustand erreicht. Voraussetzung ist, dass neue Eingaben von *HA* nur in einem stabilen Zustand akzeptiert werden, also wenn alle

internen Ereignisse abgearbeitet wurden. Die Existenz dieser Abbildung macht deutlich, dass Automaten durch die Strukturierung nicht an Ausdruckskraft gewinnen. Einfache Automaten benötigen in bestimmten Fällen allerdings (mehrfach) exponentiell mehr Zustände als hierarchische Automaten um ein bestimmtes Verhalten zu beschreiben [Yan00; vZij04]. Die Hierarchie dient somit hauptsächlich einer besseren Benutzbarkeit und kompakteren Schreibweise. In dieser Arbeit wird daher häufig nicht explizit auf hierarchische Automaten eingegangen.

2.2.4 Arten von Abweichungen

Dieser Abschnitt betrachtet, wie die Ausführung eines realen Systems von seiner Spezifikation abweichen kann und wie diese Abweichungen modelliert werden können. Eine Abweichung kann, muss aber nicht, zu einem Ausfall des gesamten Systems führen. In einem robusten System ist sogar davon auszugehen, dass viele Abweichungen abgefangen und behandelt werden. Gerade deshalb ist es wichtig, Abweichungen zu erkennen. Zum einen, damit reaktive Mechanismen aktiviert werden, die das System robust machen sollen, zum anderen ist anzunehmen, dass Abweichungen die Leistung des Systems herabsetzen. Avizienis, Laprie und Randell [ALR01] beschreiben die fundamentalen Konzepte von *Zuverlässigkeit* und klassifizieren dazu Ausfallarten und Fehler. Dabei beschreiben Abweichungen die Folge eines ausgelösten Fehlers und übertragen diesen potenziell zu einem Ausfall. Im Folgenden wird beschrieben, welche Abweichungen für diese Arbeit relevant sind.

Wie sich Abweichungen darstellen und wie ihre Auswirkungen interpretiert werden, wird in einem Fehlermodell bestimmt. Die Fehlermodelle dieser Arbeit sind geprägt durch die Annahme, dass eine Abweichung immer ein Verstoß gegen die Spezifikation ist. In Bezug auf eine Beschreibung des erwarteten Verhaltens mit einem Zustandsautomaten bedeutet das, dass es keinen passenden, gültigen Zustandsübergang zu einem auftretenden Ereignis gibt. Aus Sicht der Spezifikation kann das System folglich in einem Zustand q_s genau durch ein Ereignis abweichen, das dort nicht erwartet wird ($\chi \notin \mathbb{E}^{q_s}$). Diese Arbeit betrachtet verschiedene Gruppen von Abweichungen, die entsprechend des Zustands (q_t) aus der Spezifikation klassifiziert werden, den das System nach der Abweichung einnimmt. Abbildung 2.2 illustriert Beispiele für verschiedene Abweichungen mit gepunkteten Pfeilen. Bleibt das System unter Beobachtung (SuO, engl.: *System under Observation*) im selben Zustand ($q_t = q_s$), so handelt es sich um eine Abweichung mit einem *überflüssigen* Ereignis (Abbildung 2.2a). Eine Abweichung mit einem *veränderten* Ereignis (Abbildung 2.2b) liegt vor, wenn das SuO einen Zustand einnimmt, der mit einem Zustandsübergang, aber durch ein anderes Ereignis, erreichbar ist ($\exists e : \delta(q_s, e) \mapsto q_t$). In einer Abweichung wurde ein Ereignis *ausgelassen* (Abbildung 2.2c), wenn der neue Zustand über einen Zwischenzustand



(a) überflüssig *(b)* verändernd

(c) auslassend *(d)* zufällig

Abb. 2.2: Beispiele für die verschiedenen Arten von Abweichungen symbolisiert durch die gepunktete Pfeile.

erreicht werden kann, der einen Zustandsübergang mit dem beobachteten Ereignis zu q_t hat ($\exists e : \delta(q_s, e) \mapsto q_i \wedge \delta(q_i, \chi) \mapsto q_t$). Wird der neue Zustand beliebig aus der Menge der Zustände gewählt ($q_t \in \mathbb{S}$), wird die Abweichung als *zufällig* bezeichnet (Abbildung 2.2d). Ein Fehlermodell kann eine oder mehrere dieser (oder anderer) Gruppen von Abweichungen enthalten. Da keine Annahme darüber gemacht wird, mit welchen internen Zuständen das System realisiert ist, sondern das beobachtbare Verhalten mit dem erwarteten abgeglichen wird, werden keine Abweichungen mit Bezug auf zusätzliche oder fehlende Zustände betrachtet. Während erwartetes Verhalten als deterministisch angenommen wird, können die Transitionen der Abweichungen auch nicht deterministisch sein.

2.2.5 Verhaltensbewertung mit Automaten

Die Definition von A sieht keine explizite Unterscheidung zwischen erwarteten und unerwarteten Eingaben vor. Um eine Bewertung von Verhalten zu ermöglichen, kann festgelegt werden, dass nur erwartetes Verhalten modelliert wird. Implizit bestimmt sich damit unerwartetes Verhalten über das nicht modellierte.

Ein Pfad ist eine Abfolge von Zustandsübergängen, bei denen der nächste Zustandsübergang von dem Zielzustand des Vorherigen ausgeht. Ein Trace w ist *gültig*, wenn es einen Pfad durch den Zustandsautomaten A ab dem initialen Zustand s_0 beschreibt, d. h. wenn die enthaltenen Ereignisse die Zustandsübergänge in dieser Reihenfolge auslösen. Jede Teilsequenz eines gültigen Traces ist *erwartetes Verhalten*, d. h. es existieren mindestens ein Präfix v_p und ein Suffix v_s, sodass $v_p v v_s = w$. Mit anderen Worten, die Sequenz v von Ereignissen ist in einem gültigen Trace w enthalten und daher ist v erwartetes Verhalten, das beobachtet werden kann, indem man einem Pfad in dem Zustandsautomaten folgt. Wenn das Verhalten *unerwartet* ist, enthält es mindestens ein Ereignis a_j mit

$\langle q_j, a_j \rangle \notin \mathrm{dom}(\delta)$. Ein *ungültiges* Trace $\bar{w}$ enthält unerwartete Verhaltensweisen und widerspricht damit der Spezifikation. Das SuO ist *konform* zur Spezifikation, wenn es sich in $s \in \mathbb{S}$ befindet und $e \in \mathbb{E}^s$ ausgibt. Andernfalls *weicht* es *ab* und *verletzt* die Spezifikation. Jede Abfolge von Eingaben, die durch den Start in Zustand s_0 und das Folgen eines Pfades durch den Automaten erreicht werden kann, ist ein gültiges Trace. Für Abbildung 2.1 ist dies z. B. *join,ack,info,info,info,leave,ack*. Jede Untersequenz davon ist erwartetes Verhalten. Für ein ungültiges Trace gibt es keinen passenden Pfad durch den Automaten. Wenn beispielsweise die Sequenz des vorherigen Beispiels um *info* erweitert wird, ist das Trace ungültig und jede Untersequenz die *leave,ack,info* enthält, ist ein unerwartetes Verhalten.

Um Teile einer Beschreibung explizit als unerwartet zu kennzeichnen, kann die Ausgabe eines Zustandsautomaten verwendet werden. Die bisher vorgestellte Definition für Automaten bietet lediglich die Möglichkeit zu der Beschreibung von Verhalten, aber noch keine für die Ausgabe eines Automaten. Es wird diesbezüglich zwischen zwei Klassen unterschieden: *Akzeptoren* und *Transduktoren*. Für Akzeptoren wird eine Teilmenge ihrer Zustände als *akzeptierend* gekennzeichnet. Sie akzeptieren einen Trace genau dann, wenn sie sich am Ende in einem dieser Zustände befinden. Transduktoren arbeiten hingegen als Transformatoren und erzeugen eine Ausgabesequenz aus der Eingabesequenz. Dies ermöglicht eine Bewertung der Eingabe nach jedem Schritt. In dieser Arbeit werden deshalb Transduktoren vom Typ *Mealy-Automat* [Mea55] verwendet. Dabei hängt die Ausgabe von Zustand und Eingabe ab. Dazu wird die Definition des Automaten A auf ein 6-Tupel erweitert: $\mathrm{M} = \langle \mathbb{E}, \mathbb{S}, s_0, \delta, \mathbb{D}, \gamma \rangle$.

- $\mathbb{D}$ ist die Menge möglicher Ausgaben und
- $\gamma : \mathbb{S} \times \mathbb{E} \to \mathbb{D}$ definiert die Ausgabe für jeden Zustandsübergang.

Die Bedeutung der Ausgabe kann, je nach Anwendungszweck des Automaten, unterschiedlich sein.

Für die Verhaltensbewertung wird die Menge der Ausgaben $\mathbb{D}$ auf eine Menge mit möglichen Bewertungen, den sog. Verdikten, festgelegt. Die Zusammenstellung der Verdikte unterscheidet sich mit Bezug auf die Art des Verstoßes, den die Ausgabe berichten soll. Üblicherweise werden entweder ungültige Traces, unerwartetes Verhalten oder Abweichungen untersucht. Mindestens muss dazu zwischen akzeptiert ($\top$) und abgelehnt ($\bot$) als Ausgabe unterschieden werden. Die Ausgaberelation γ wird damit zur Bewertungsrelation und kann für jeden Zustandsübergang ein Verdikt angeben. Die Erweiterung der Bewertungsrelation für Sequenzen ist die Aneinanderreihung der einzelnen Ausgaben für jedes Ereignis (2.12). Damit auch eine Bewertung einer Menge von Zuständen Q für ein Ereignis erfolgen kann,

wird angenommen, dass einzelne Verdikte mit einer Operation $\oplus$ kombiniert werden können. Die Definition kann beliebig sein, aber typischerweise handelt es sich dabei um die Supremum- oder Infimum-Operatoren, wenn die Verdikte einen Verband bilden. Die erweiterte Bewertungsrelation ist dann definiert durch Gleichung 2.13.

$$\gamma(q_1, a_1 \dots a_i) = \gamma(q_1, a_1) \dots \gamma(q_i, a_i) \tag{2.12}$$

$$\gamma(Q, a_i) = \bigoplus_{q \in Q} \gamma(q, a_i) \tag{2.13}$$

2.2.6 Ungewisser Zustand

Der aktuelle Zustand des SuO ist üblicherweise nicht vollständig auslesbar. Eine der Herausforderungen bei der Überprüfung ist es deshalb, mit diesem Nicht-Wissen richtig umzugehen. Durch eine Abbildung der Beobachtungen eines Traces auf die Spezifikation können Rückschlüsse auf den sichtbaren Zustand des Systems getroffen werden. Allgemein wird zwischen dem Problem der Ungewissheit des initialen und des aktuellen Zustands unterschieden [LY96; San05]. Ersteres befasst sich mit dem Wissen, das über den ersten Zustand verfügbar ist, also den Zustand vor einer Beobachtung. Das andere betrifft den Zustand danach.

Das Problem der Ungewissheit des initialen Zustands (bezogen auf einen Mealy-Automaten) wird als eine Abbildung $\pi(v)$ einer Eingabesequenz $v \in \mathbb{E}^*$ auf eine Partition der Zustände (2.14) dargestellt [LY96; San05]. Zwei Zustände s, t sind genau dann in dem gleichen Block B_i, wenn $\gamma(s, v) = \gamma(t, v)$. Für eine bestimmte Ausgabe ist bekannt, dass der initiale Zustand in B_1 enthalten ist, für eine andere in B_2, usw. Die Ungewissheit des aktuellen Zustands wird formalisiert als eine Abbildung $\sigma(v)$ einer gegebenen Eingabesequenz $v \in \mathbb{E}^*$ auf die Mengen möglicher finaler Zustände je unterschiedlicher Ausgabesequenz (2.15). Mit jeder weiteren Eingabe entwickelt sie sich weiter. Die Ungewissheit des aktuellen Zustands für $\sigma(vy)$ kann inkrementell berechnet werden, indem $\delta(\cdot, y)$ auf jeden der Blöcke in $\sigma(v)$ angewendet wird. Sollten die Zustände eines Blocks unterschiedliche Ausgaben für y produzieren, wird er in neue Blöcke für jede der Ausgaben aufgespalten. Im Gegensatz zu $\pi(v)$ handelt es sich nicht um eine Partitionierung der Zustände, sondern ein Zustand kann in $\sigma(v)$ in mehreren Blöcken auftreten.

$$\pi(v) = \{B_1, B_2, \dots, B_r\} \subset \mathcal{P}(\mathbb{S}) \tag{2.14}$$

$$\sigma(v) = \{\delta(B_i, v) \mid B_i \in \pi(v)\} \subset \mathcal{P}(\mathbb{S}) \tag{2.15}$$

Anhand dieser Definition des (nicht) Bekannten, kann untersucht werden, wie sich das Wissen über den aktuellen Zustand bei Erweiterung der Sequenz verändert. Es wird schrittweise präzisiert [San05; LY96]. Dabei werden Trennsequenzen (engl.: *separation sequence*) und Zusammenführungssequenzen (engl.: *merging sequence*) unterschieden.

Eine **Trennsequenz** macht mindestens zwei initiale Zustände s, t durch unterschiedliche Ausgaben unterscheidbar, also $\gamma(s, v) \neq \gamma(t, v)$. Damit liegen diese Zustände in verschiedenen Blöcken von $\pi(v)$. Ein minimierter Mealy-Automat hat mindestens eine Leitsequenz (engl.: *homing sequence*), die ihn in einen Zustand überführt, der durch die Beobachtung seiner Ausgabe bestimmt werden kann. Denn v ist eine Leitsequenz, wenn jeder Block in $\sigma(v)$ genau ein Element enthält. Eine solche Sequenz kann durch Aneinanderreihung von Trennsequenzen für jedes Paar von Zuständen in polynomialer Zeit aufgestellt werden. Für jedes Paar muss eine Trennsequenz existieren, sonst wären beide Zustände identisch und der Automat nicht minimal.

Eine **Zusammenführungssequenz** reduziert die Größe eines Blocks, indem mindestens zwei Zustände im gleichen Block den gleichen Zustand nach einem Zustandsübergang einnehmen. Dies reduziert die Ungewissheit über den aktuellen Zustand, denn es bleiben weniger Möglichkeiten übrig. Für die genaue Identifizierung des initialen Zustands sind diese Sequenzen jedoch ungeeignet, da auch eine genauere Unterscheidung des Ursprungs verhindert wird. Eine Synchronisierungssequenz (engl.: *synchronizing sequence*) überführt einen Automaten in einen bestimmten Zustand, ohne das der initiale Zustand oder die Ausgabe des Automaten beachtet werden muss. Eine solche Sequenz v liegt vor, wenn $\delta(s_i, v) = \delta(s_j, v)$ für alle Zustände $s_i, s_j \in \mathbb{S}$. Wenn sie existiert, kann sie durch Aneinanderreihung von Zusammenführungssequenzen erstellt werden. Die Überprüfung ihrer Existenz sowie ihre Konstruktion sind in polynomialer Zeit durchführbar. Das Problem die kürzeste Synchronisierungssequenz zu finden, ist jedoch NP-vollständig, wie Eppstein [Epp90] gezeigt hat. Deshalb sind Heuristiken, mit denen möglichst kurze Synchronisierungssequenzen effizient gefunden werden können, Gegenstand der Forschung [z. B. KKY18; Kar+16; RS15; Alt+17].

Um die Synchronisierungssequenzen eines Automaten zu finden, kann ein Synchronisierungsbaum (engl.: *synchronizing tree*) verwendet werden [San05]. Dabei handelt es sich um einen Baum mit einem Wurzelknoten dessen Kanten mit den Eingaben und die Knoten mit Mengen von Zuständen beschriftet sind. Jeder Knoten, der kein Blatt ist, hat genau $|\mathbb{E}|$ Kinder und die zu diesen führenden Kanten haben verschiedene Beschriftungen. Die Knoten werden mit $\delta(\mathbb{S}, v)$ beschriftet, wobei v die Sequenz von Eingaben der Kanten auf dem Pfad vom Wurzelknoten aus sind. Bei einem Knoten handelt es sich um ein Blatt, wenn seine Beschriftung nur aus einem (oder keinem) Zustand besteht oder die gleiche Beschriftung bereits (mit kleinerer Tiefe) im Baum existiert.

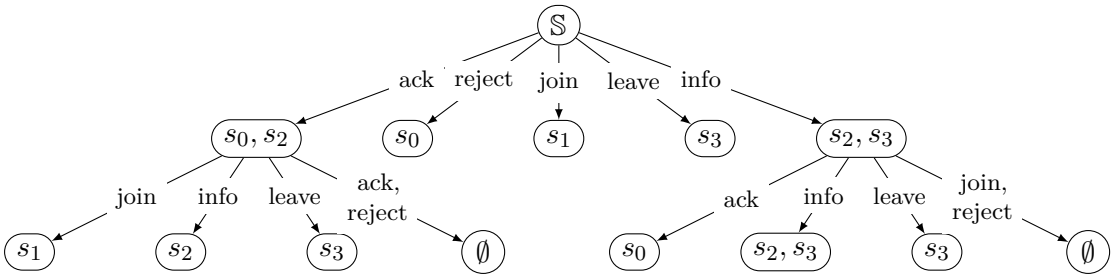


Abb. 2.3: Synchronisierungsbaum für den Automaten aus Abschnitt 2.2.1.

Der Synchronisierungsbaum für das Beispiel aus Abschnitt 2.2.1 ist in Abbildung 2.3 dargestellt. Eingaben, bei denen es immer nur einen möglichen Zustand als Nachfolger gibt, werden als eindeutig bezeichnet. In dem dargestellten Baum sind sie als Blätter auf der ersten Ebene zu erkennen. Wie in der Abbildung sichtbar ist, sind in dem Beispiel die drei Eingaben *reject*, *join* und *leave* eindeutig. Für die Eingaben *ack* und *info* sind erst weitere Eingaben notwendig um den Zustand genau zu bestimmen. Die mit $\emptyset$ bezeichneten Knoten besagen, das es keinen Zustand gibt, der nach einer Sequenz von Eingaben mit erwartetem Verhalten aktiv sein kann. Sie können daher nur durch unerwartetes Verhalten erreicht werden.

2.3 Verifikation zur Laufzeit

Ziel dieser Arbeit ist es, alle in der Interaktion erkennbaren Abweichungen der Ausführung eines Systems von seiner Spezifikation zu finden. Unter Laufzeitüberprüfung (RV, engl.: *Runtime Verification*), auch Verifikation zur Laufzeit, wird im weitesten Sinne die Studie der Methoden verstanden, mit denen das dynamische Verhalten eines datenverarbeitenden Systems analysiert werden können [Fal+18]. Die gebräuchlichste Analyseform ist die Überprüfung, ob ein bestimmter Lauf eines Systems einer bestimmten Spezifikation entspricht. Die Methoden im Themengebiet sind zwar in den letzten Jahren gereift, es fehlt aber noch an einem einheitlichen Sprachgebrauch. Deshalb schlagen Falcone u. a. [Fal+18] eine Taxonomie für RV vor. Sie unterscheiden die Kategorien Spezifikation, Monitor, Verwendung, Reaktion und Trace. Diese sehen sie auch als Hauptbestandteile jedes RV Werkzeugs. Zudem betrachten sie die Interferenz und Anwendungsgebiete der Werkzeuge. Jede der Kategorien wird noch feiner unterteilt. Diese Unterteilung wird im Folgenden zusammengefasst wiedergegeben. Eine Übersicht der Taxonomie ist in Abbildung 2.4 dargestellt.

Die *Spezifikation* wird darin unterschieden, ob sie **implizit** oder **explizit** angegeben wird. Bei einer impliziten Spezifikation wird davon ausgegangen, dass es für die zu überprüfenden Aspekte des erwarteten Verhaltens ein allgemeines Verständnis gibt, wie beispielsweise

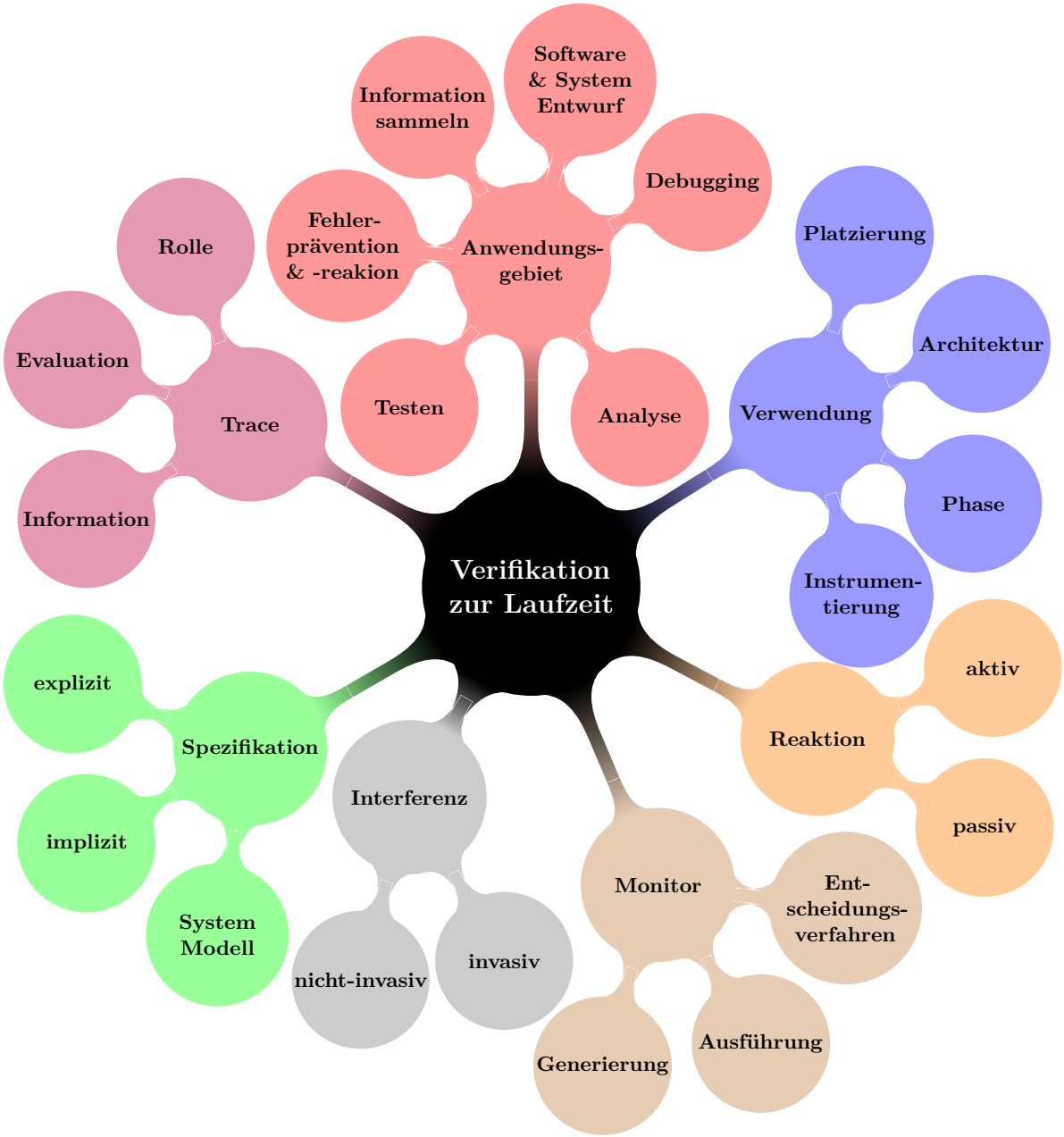


Abb. 2.4: Taxonomie der Verifikation zur Laufzeit nach Falcone u. a. [Fal+18].

korrekter Speicherzugriff oder Atomarität von Operationen. Bei einer expliziten Spezifikation gibt der Benutzer die funktionalen und/oder nicht-funktionalen Anforderungen an. Er definiert damit eine Abbildung des Traces auf die Domäne der **Ausgabe**. Die Spezifikation folgt dabei normalerweise einem bestimmten **Paradigma** und beschreibt die Abbildung **operational** (z. B. mit einem endlichen Automaten) oder **deklarativ** (z. B. mit einer Formel). Im einfachen Fall wird ein einzelnes **Verdikt** mit einer Ausführung des Systems assoziiert. Die Ausgabe kann **Zeugen** beinhalten, das sind beispielsweise Belegungen von freien Variablen der Spezifikation, mit denen ein Verstoß beobachtet werden kann. Bei kontinuierlich neuen Ausgaben spricht man von einem **Streaming** der Bewertungen. Bedingungen an das Zeitverhalten können sich auf **logische** oder **physische Zeiten** beziehen. Logische Zeiten betrachten die relative Ordnung von Ereignissen und es wird unterscheiden, ob diese Ordnung **total** (z. B. bei Beobachtung eines monolithischen Systems) oder **partial** (z. B. bei einem verteilten System) ist. Physische Zeiten befassen sich mit der Menge an realer Zeit, die während der Ausführung des Systems vergeht. Die Zeitinformation kann hier **diskret** oder **stetig** sein. Teilweise wird Zeit auch als Teil der Daten betrachtet. Generell unterscheiden sich verschiedene Sprachen für Spezifikationen in den behandelten **Daten** und den unterstützten **Operationen** auf ihnen.

Der *Monitor* ist die zentrale Komponente eines RV Ansatzes. Damit ist die Komponente gemeint, die neben dem beobachteten System ausgeführt wird, um es zu überprüfen. Das **Entscheidungsverfahren** kann entweder **analytisch**, beispielsweise eine Anfrage und Überprüfung von Datensätzen einer Datenbank, oder **operational** sein. Operationale Entscheidungsverfahren können auf **Automaten** oder **Regelsystemen** basieren. Das Entscheidungsverfahren arbeitet mit einem Objekt, das oft auch selbst als Monitor bezeichnet wird. Dieses kann **explizit** aus der Spezifikation **generiert** werden, z. B. ein Automat aus einer LTL-Formel, oder **implizit** existieren. Die **Ausführung** des Monitors kann entweder **direkt** sein, wenn beispielsweise ausführbarer Code generiert wurde, oder sie **interpretiert** die Spezifikation oder andere Informationen zur Laufzeit. Ein Monitor ist **intakt**, wenn er keine inkorrekten Ausgaben erzeugt und **vollständig**, wenn er immer eine Ausgabe liefert.

Mit *Verwendung* wird auf die Aspekte eingegangen, wie ein Monitor implementiert sowie organisiert ist und wie er Informationen aus dem beobachteten System entgegennimmt, sofern das geschieht. So beschreibt die **Phase**, ob der Monitor **offline**, d. h. nach der Ausführung des SuO, oder **online** eingesetzt wird. In der online Phase, wenn der Monitor gleichzeitig mit dem SuO ausgeführt wird, kann er **synchron** oder **asynchron** angebunden sein, entsprechend abhängig davon, ob das System die Ausführung pausiert und auf die Auswertung durch den Monitor wartet oder nicht. Die **Platzierung** eines Monitors ist **integriert**, wenn er sich den Adressbereich im Speicher mit dem SuO teilt, sonst ist sie **abgegrenzt**. Prinzipiell handelt es sich dabei um verschiedene Vorgehensweisen bei der

Instrumentierung. Während integrierte Werkzeuge diese implizit mitbringen, da sie selbst die notwendigen Ereignisse erfassen, bieten abgegrenzte Werkzeuge normalerweise eine Schnittstelle an, über die sie Ereignisse entgegennehmen. Die Instrumentierung kann auf **Software** oder **Hardware** Ebene erfolgen. Abschließend kann die Architektur eines Monitors **zentral** (z. B. ein Monolith) oder **dezentral** (z. B. verschiedene kommunizierende Komponenten) sein.

Die *Reaktion* eines Monitors wird als **passiv** bezeichnet, wenn er die Ausführung des SuO nicht oder nur minimal beeinflusst. Typischerweise liefert er dann auf Basis seiner Beobachtungen die **Ausgabe der Spezifikation** (z. B. Verdikte) zurück. Zudem kann er eine **Erklärung** (z. B. einen Zeugen) und **Statistiken** zu der Ausgabe bereitstellen. Die Reaktion ist **aktiv**, wenn in die Ausführung des Systems eingegriffen wird. Beispielsweise kann ein **erzwingender** Monitor versuchen zu verhindern, dass Verstöße gegen ein Merkmal auftreten, indem er das System dazu zwingt, entsprechend der Spezifikation zu agieren. Auch nachdem ein Verstoß auftritt, kann der Monitor reagieren, indem er entsprechende **Wiederherstellungsroutinen** einbringt, vorhandene **Ausnahmebehandlungen** des Systems anstößt oder das System auf einen korrekten Zustand **zurücksetzt**.

Ein *Trace* kann in zwei verschiedenen **Rollen** in Bezug zu einem RV Ansatz stehen. Ein **beobachtetes** Trace bezeichnet das Objekt, das aus dem SuO ausgelesen wird. Umgekehrt ist das **Modell** des Traces das mathematische Objekt, das Teil der Semantik des Spezifikationsformalismus ist. Dieses Modell kann **unendlich** sein (wie in LTL), während das beobachtete Trace immer **endlich** ist. Das Modell muss die Ideen von Zeit und Daten aus der Spezifikation widerspiegeln. Ein beobachtetes Trace kann **ereignisgesteuert** oder **zeitbasiert** erzeugt werden. Im ersten Fall bekommt der Monitor Informationen, wenn etwas von Interesse passiert, im zweiten Fall werden die Informationen mehr oder weniger regelmäßig abgerufen. Das Trace ist **genau**, wenn es alle relevanten Vorkommen der Informationen enthält, sonst, d. h. wenn Informationen fehlen können, ist es **ungenau**. Eine **Information** kann aus dem internen **Zustand**, Benachrichtigungen über das Auftreten bestimmter **Ereignisse**, den **Ein- und Ausgaben** des Systems oder zeitkontinuierlichen **Signalen** bestehen. Letztere können über geschlossene Ausdrücke oder durch diskrete Messwerte erfasst werden. Die vom Monitor erhaltenen Informationen repräsentieren eine **Evaluation** des Systemzustands. Sie hat Bezug zu einem bestimmten **Punkt** (temporal oder im Programm) oder einem **Intervall**.

Die *Interferenz* bestimmt, in wieweit der Monitor das beobachtete System beeinflusst. Theoretisch kann zwischen **invasiv** und **nicht-invasiv** unterschieden werden. Praktisch ist eine absolut nicht-invasive Beobachtung aber kaum zu realisieren. Die Störung des SuO kann beispielsweise aus einem induzierten zusätzlichen Aufwand (bezogen auf Zeit oder Speicher) oder einem veränderten *Scheduling* des Systems bestehen. Der Effekt hängt

stark von der verwendeten Instrumentierung und der Verzahnung mit dem SuO ab. Eine Interferenz besteht auch, wenn der Monitor aktiv in die Steuerung des Systems eingreift (siehe Reaktion).

Das *Anwendungsgebiet* wurde in die Taxonomie aufgenommen, da es einen großen Einfluss auf andere Aspekte eines RV Werkzeugs haben kann. Die folgenden (nicht vollständigen) Kategorien wurden gefunden: Ein RV Werkzeug kann helfen **Informationen** über das System zu **sammeln**, dies schließt beispielsweise Statistiken und die Visualisierung der Ausführung (z. B. mit Traces, Graphen oder Diagrammen) ein. Es kann eine **Analyse** des Systems ausführen, beispielsweise um statische Analyseverfahren zu ergänzen und Belange zu bewerten, wie **Sicherheit**, **Datenschutz** oder **Lebendigkeit**. Weiter kann es beim Finden von Defekten oder der Lokalisierung von Fehlern in Systemen helfen und damit Techniken des **Debugging** und **Testen** ergänzen. Durch Ausnutzung der bereits genannten Techniken kann RV dabei helfen, das generelle Problem der **Fehlerprävention und -reaktion** zu adressieren. Es bietet Möglichkeiten Fehler zu erkennen, diese einzugrenzen, sich von ihnen zu erholen und das System zu reparieren.

Laut Falcone u. a. [Fal+18] wurde die Taxonomie auf der Grundlage von Informationen zu 50 Werkzeugen aufgestellt. Sie beschreiben damit 20 zum Zeitpunkt der Veröffentlichung ihrer Arbeit aktuelle RV Werkzeuge. Viele der nicht in dem Vergleich berücksichtigten Werkzeuge kämen aus der Verarbeitung von Datenströmen oder seien eher von historischem Interesse da sie nicht weiter entwickelt werden. Sie merken an, dass die aufgestellte Taxonomie implizite Ansätze vernachlässige. Für viele Werkzeuge wird angegeben, dass diese sowohl online als auch offline eingesetzt werden können. Nur ein Werkzeug wurde von ihnen als rein offline klassifiziert.

2.4 Zusammenfassung

In diesem Kapitel wurden die Grundlagen vorgestellt, auf denen der präsentierte Ansatz zur modellbasierten Beschreibung und Überprüfung des Verhaltens vernetzter eingebetteter Systeme aufbaut. Mit der Vorstellung der Grundzüge von modellbasierter Entwicklung wurde die Verwendung von Modellen motiviert. Die Modellierungsmethodik, die in Kapitel 4 vorgestellt wird, wurde mit den vorgestellten Modellierungssprachen realisiert. Die Automatentheorie beschreibt die formalen Grundlagen für die dort definierte Verhaltensbeschreibung und die in Kapitel 5 vorgestellte Resumption. Die Taxonomie der Verifikation zur Laufzeit von Falcone u. a. [Fal+18] bietet einen Überblick, wie das dynamische Verhalten eines datenverarbeitenden Systems analysiert werden kann. In Kapitel 3 werden existierende Ansätze dazu betrachtet.

Kapitel 3

3 Absicherung interaktiver Software-intensiver Systeme

Verschiedene Forschungsfelder verfolgen unterschiedliche Ansätze, die Unterschiede zwischen dem Verhalten eines Systems und seiner Spezifikation zu identifizieren. Dieses Kapitel stellt die Anforderungen an den in dieser Arbeit erarbeiteten Ansatz vor, um anschließend einen Überblick zu geben, in wieweit existierende Ansätze diese bereits erfüllen.

3.1 Anforderungen an den Ansatz

Diese Arbeit betrachtet die Aufgabe, wie die erkennbaren Unterschiede zwischen der Ausführung eines System unter Beobachtung (SuO, engl.: *System under Observation*) und seiner Spezifikation mit einem einzigen Monitor gefunden werden können. Dazu wird ein Monitor, wie in Abbildung 3.1 dargestellt, parallel zum SuO ausgeführt. Der Monitor vergleicht und bewertet die beobachteten Interaktionen mit dem hinterlegten Modell. Die Kommunikation kann über einen Hardware-Bus, getrennte Kanäle, eine Middleware oder über andere Wege erfolgen, solange der Monitor diese beobachten kann.

Da das Problem dem Themengebiet Laufzeitüberprüfung (RV, engl.: *Runtime Verification*) zugeordnet werden kann, wird im Folgenden die in Abschnitt 2.3 vorgestellte Taxonomie für eine genauere Beschreibung der Aufgabe zu Hilfe genommen. Die Anforderungen leiten sich aus den in Abschnitt 1.2 definierten Herausforderungen ab.

Der Anwendungszweck des Ansatzes besteht im Test und Debugging, da diese beiden Aktivitäten normalerweise im Rahmen der Integration verschiedener Software-Komponenten anfallen. Ziel ist dabei nicht, dass der Ansatz selbstständig mit Testfällen das SuO stimuliert. Vielmehr soll er als Testorakel [SWH11] dienen, das Interaktivität unterstützt. Während ein Testfall darauf fokussiert vorbestimmte Reaktionen zu überprüfen, die durch

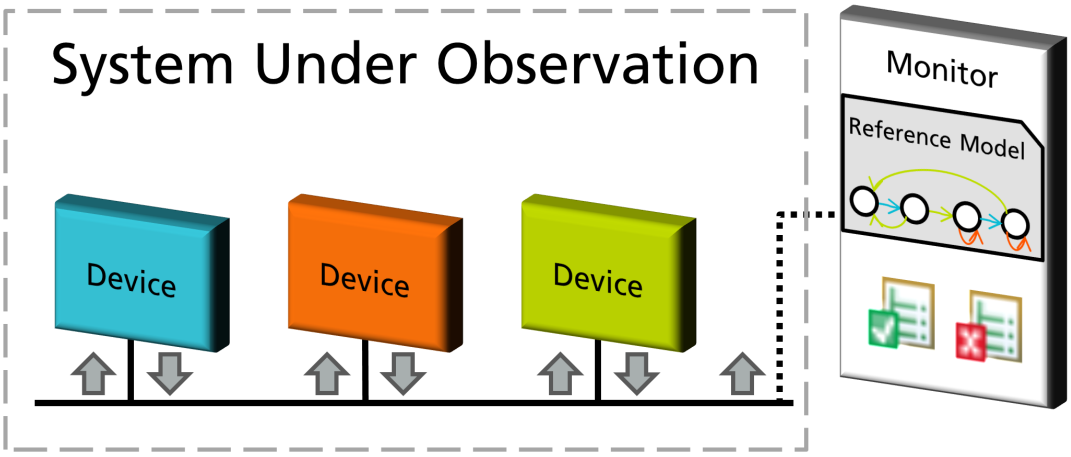


Abb. 3.1: Monitor der die Kommunikation eines Systems überwacht.

seinen Stimulus ausgelöst werden, soll der Ansatz das Verhalten auch in realen, komplexen Betriebssituationen mit der ihm vorliegenden Spezifikation vergleichen. So kann das System ebenfalls überwacht werden, wenn es manuell bedient wird. Der Ansatz kann damit auch bei der Erkennung von Fehlerursachen unterstützen, wenn er erkennbare Abweichungen zur Spezifikation aufzeigt, da somit das untersuchte Problem leichter eingegrenzt werden kann. Um die kontinuierliche Weiterentwicklung solcher Systeme zu unterstützen, sollte der Ansatz auch in das System integriert werden können. Damit kann das System selbst Abweichungen im Verhalten erkennen, beispielsweise durch eine unbeabsichtigte Veränderung der Konfiguration, und geeignete Gegenmaßnahmen können angestoßen werden.

Die Spezifikation muss explizit sein, da das Verhalten von der realisierten Funktionalität abhängt und es davon kein allgemeines Verständnis gibt. Verschiedene Sprachen können für die Spezifikation eines Monitors verwendet werden [DGR04]. In der Literatur finden sich verschiedene Formalismen dazu, beispielsweise lineare temporale Logik (engl.: *Linear Temporal Logic*, LTL) [BLS11]. Solch eine Beschreibung kann auch mit Zuständen und Transitionen zwischen diesen beschrieben werden [FHR13], also mit einem (endlichen) Automaten. Zustandsautomaten sind verständlich, weit verbreitet, lernbar, diagnostizierbar, geeignet für die Verifikation und leicht zu bearbeiten, insbesondere im Vergleich mit anderen Formalisierungen wie Hidden-Markov-Model, Petri-Netzen oder Regel-basierten Systemen [MNE15]. Deswegen wird eine operationale Beschreibung mit Automaten bevorzugt. Bei der Entwicklung eingebetteter Systeme werden Zustandsautomaten häufig zur Beschreibung von Kommunikationsprotokollen und Interaktionen zwischen Komponenten verwendet. Zur Erfassung der unabhängig voneinander auftretenden Ereignisse in verteilten Systemen, muss der Formalismus die Möglichkeit bieten eine partielle Ordnung zu beschreiben. Da eingebettete Systeme typischerweise Zeitschranken einhalten müssen, ist auch eine Definition der physischen Zeit notwendig. Die Daten sind z. B. im automobilen Infotainment häufig

strukturiert oder komplexer, wie beispielsweise eine Liste von Liedern, die neben den Titeln auch Informationen über die Laufzeit, den Interpreten und häufig auch ein Bild enthält. Die Operationen auf den Daten müssen ermöglichen, einzelne Elemente dieser strukturierten Daten zu vergleichen. Zudem werden einfache arithmetische Operationen auf zahlenbasierten Datentypen benötigt. Da in dieser Arbeit angenommen wird, dass die Spezifikation nur das Soll-Verhalten beschreibt, müssen keine Verdikte hinterlegt werden. Diese ergeben sich implizit. Prinzipiell sollte der Ansatz aber auch mit expliziten Verdikten in der Spezifikation verwendet werden können.

Das Modell zur Beschreibung des Traces muss nicht zwingend unendlich sein, da das SuO zu jedem Zeitpunkt immer nur endlich viele Schritte absolviert hat und nur diese Beobachtungen die Basis für die Bewertung sind. Da das System ständig neue Schritte produziert, muss das Modell vielmehr mit einer kontinuierlichen Erweiterung zurechtkommen. Es wird angenommen, dass das beobachtete Trace ereignisgesteuert und genau aufgezeichnet wird. Ereignisse sind dabei die Ein- und Ausgaben der einzelnen Komponenten des Systems, also die Nachrichten zwischen ihnen. Die Evaluation findet jeweils zum Zeitpunkt des Ereignisses statt. Da alle Abweichungen gefunden werden sollen, erfolgt die Ausgabe als Streaming, wobei jedes Verdikt immer auf das Intervall seit der letzten gemeldeten Abweichung Bezug nimmt.

Der betrachtete Monitor ist passiv und gibt eine Ausgabe entsprechend der Spezifikation zurück. Er zeigt, ob ein beobachtetes Trace gültig ist oder nicht. Das bildet die Grundlage, um weitere Statistiken über die Ausführung zu sammeln und um aktive Reaktionen auf Fehler zu realisieren. Dabei sind Interferenzen, soweit möglich, zu minimieren. Deshalb liegt der Fokus der Überprüfung auf der Kommunikation, da diese häufig (nahezu) nicht-invasiv abgehört werden kann.

3.2 Dynamische Software-Testverfahren

Unter dynamischen Software-Testverfahren werden Prüfmethoden zusammengefasst, die darauf abzielen, Fehlverhalten von Software in deren Ausführung festzustellen. Dies bedeutet, dass insbesondere die Abhängigkeit von dynamischen Laufzeitparametern berücksichtigt wird. Beispiele dafür sind variierende Eingabeparameter, bestimmte Interaktionen mit dem Benutzer oder die verwendete Laufzeitumgebung. Sie grenzen sich von statischen Verfahren ab, bei denen die Software nicht ausgeführt wird, sondern direkt die Programmstruktur oder der Quellcode auf mögliche Fehler untersucht wird. Zwar sind dynamische Analyseverfahren häufig nicht vollständig und können deshalb Fehler übersehen, die absichtliche Unvollständigkeit hilft aber dabei, die Grenzen von statischen Analysen zu überwinden [FHR13], wie z. B. Zustandsexplosion. Im Folgenden werden dynamischen Test-Verfahren

in zwei Kategorien eingeteilt. *Out-of-Service* Verfahren liefern erst nach der abgeschlossenen Ausführung des Systems eine Bewertung oder benötigen dedizierte Instanzen des Systems, die sie zum Zweck der Analyse manipulieren. *In-Service* Verfahren ermöglichen hingegen eine direkte Rückmeldung zu einem System im normalen Betrieb. Dieser Überblick kann, im Rahmen dieser Arbeit, aufgrund der Vielzahl an Verfahren in diesem Bereich nicht vollständig sein. Die Auswahl der konkreten Ansätze für die verschiedenen Verfahren basiert auf der Relevanz für den vorgestellten Ansatz. Sie dient dazu, diesen vom aktuellen Stand der Technik abzugrenzen.

3.2.1 Out-of-Service Verfahren

Out-of-Service Verfahren – wie der Name nahelegt – sind nicht darauf ausgelegt die Ausführung während des normalen Betriebs zu überprüfen. Sie können zu diesem Zeitpunkt nicht eingesetzt werden, da noch nicht alle notwendigen Informationen vorliegen oder weil sie zur Auswertung in den Ablauf des Systems eingreifen müssten.

Conformance Checking vergleicht aufgezeichnete Ereignisprotokolle eines Prozesses mit dem dazugehörigen Prozessmodell, um aufzudecken, wo der reale Prozess vom modellierten abweicht [vdAAvD12]. Dabei werden alle beobachteten Eingaben, Ausgaben und Zwischenergebnisse des Prozesses als Ereignisse des Traces betrachtet. Es wird hauptsächlich offline eingesetzt, nachdem das SuO seine Ausführung beendet hat. Dies liegt u. a. daran, dass die verwendeten Techniken des Data Minings zum Abgleich zwischen Modell und Ausführung sehr rechenintensiv sind. Zudem können einige der Techniken nur dann effizient verwendet werden, wenn bereits die vollständigen Protokolle vorliegen.

Cook, He und Ma [CHM01] schlagen einen Algorithmus vor, um einen minimalen Satz an Änderungen wie Einfügungen, Ersetzungen oder Löschungen von Ereignissen an bestimmten Stellen zu finden, die ein gegebenes Trace in ein genau dem Modell entsprechendes transformiert. Als Maß für die Differenz eines beobachteten Ereignisprotokolls von seiner Prozessbeschreibung schlagen sie die *minimale Anzahl der Änderungen* an dem Trace vor. Allgemeiner können für verschiedene Änderungsoperationen unterschiedliche Werte als Kosten festgelegt und die Kosten aller Änderungsoperationen aggregiert werden. Reger [Reg15] nimmt den Ursprung eines Ereignisses in die Analyse auf, um sinnvolle Änderungen des Traces zu finden, die für den gleichen Ursprung konsistent sind und das Trace korrigieren. Er beschreibt Trace, Änderungsoperationen und Zustandsautomaten mit gewichteten Transduktoren. Transduktoren sind zusammensetzbare algorithmische Transformationen, d. h. Automaten, die eine Eingangssequenz lesen und das Ergebnis der Transformation als Ausgangssequenz schreiben. Jeder Schritt eines gewichteten Transduktors ist mit einem Kostenaufwand verbunden. Wie Allauzen und Mohri [AM08] gezeigt haben, kann eine dreiseitige Komposition der drei Transduktoren ohne ein großes Zwischenergebnis

durchgeführt werden. Jeder Pfad in der Zusammensetzung stellt eine Möglichkeit dar, die Ablaufverfolgung so zu bearbeiten, dass sie einem Pfad in dem Zustandsautomaten entspricht. Dieser zusammengesetzte Transduktor kann dann nach dem niedrigsten Gesamtkostenaufwand durchsucht werden. Die vollständige Suche nach den minimalen Änderungen des Traces hat eine Rechenkomplexität, die aktuell erfordert, dass sie offline ausgeführt wird.

Auch wenn Allauzen und Mohri [AM09] gezeigt haben, dass ein so zusammengesetzter Transduktor auf linearem Platz berechnet werden kann, ist die Berechnung nach wie vor teurer in Bezug auf die Rechenzeit. Daher haben Cook und Wolf [CW99] vorgeschlagen, den Suchraum einzuschränken (engl.: *prune*). Sie senken die Komplexität der Suche, indem sie Teile des Suchraums verwerfen, die wenig vielversprechend aussehen. Sie argumentieren, dass zwar die Identifizierung des minimalen Ziels durch die Einschränkung nicht mehr garantiert wird, dies aber oft vernachlässigbare Auswirkungen auf das Ergebnis hat und gleichzeitig der Suchraum drastisch reduziert wird. Sie schlagen vor, Einschränkungen auf Basis der Kosten (engl.: *cost pruning*) und der Position (engl.: *position pruning*) vorzunehmen. Ersteres eliminiert Knoten aus der Suche, die voraussichtlich zu höheren als den erwarteten Gesamtkosten führen werden, während letzteres Knoten entfernt, die τ_{prune} Schritte in der Ablaufverfolgung hinter dem aktuell besten Knoten sind.

Die Algorithmen wurden zwar nicht ursprünglich dafür entwickelt zur Laufzeit ausgeführt zu werden, aber bei Verwendung von Einschränkungen auf Basis der Position im Trace sind zur Laufzeit, mit einem Offset von τ_{prune} Schritten, alle Informationen zur Durchführung der Suche nach den Änderungen verfügbar. Der Algorithmus, ohne Einschränkung des Suchraums, wird daher auch in der Evaluation in Abschnitt 7.2.3 als Vergleich herangezogen.

Conformance Testing verfolgt das gleiche Ziel wie *Conformance Checking*, stimuliert das SuO aber aktiv, um möglichst effizient Abweichungen zu finden oder eine notwendige Abdeckung an überprüftem Verhalten zu erreichen. Deshalb wird üblicherweise von einem System unter Test (SuT, engl.: *System under Test*) gesprochen. Es gibt für verbreitete Standards komplette Sammlungen von Testfällen [TTCN3] mit denen die Konformität eines SuTs gezielt überprüft werden kann. Ein umgebender [HHS16] oder eingebetteter [KBK16] Testkontext stimuliert das SuT mit Eingangssequenzen aus Testfällen. Die Reaktionen des SuTs können dann mit dem erwarteten Verhalten verglichen werden. Im Fall von modellbasiertem Testen (engl.: *model-based testing*) wird das SuT mit einem Modell des gültigen Verhalten verglichen [PL05]. Anhand des Modells können zudem auch die Testfälle abgeleitet werden. Um das System zwischen den Tests in einen definierten Ausgangszustand zu bringen, können die in Abschnitt 2.2.6 vorgestellten Leit- und Synchronisationssequenzen helfen.

3.2.2 In-Service Verfahren

In-Service Verfahren bewerten das Verhalten eines Systems während des normalen Betriebs. Sie sollten deshalb nicht aktiv in die Abläufe des Systems eingreifen, zumindest nicht rein zum Zweck der Beobachtung. Da sie in der online Phase, also zur Laufzeit ausgeführt werden, können sie dabei helfen, dass ein System seinen eigenen Zustand erkennen und auf ungewollte Zustände reagieren kann.

Unter Laufzeitüberprüfung (RV, engl.: *Runtime Verification*) wird das Themengebiet verstanden, dass sich mit Methoden zur Überprüfung von konkreten Läufen eines *Systems unter Beobachtung* (SuO) befasst. Eine Taxonomie von [Fal+18] für RV wurde bereits in Abschnitt 2.3 vorgestellt. In diesem Abschnitt ist der Fokus auf konkrete Ansätze und der Vergleich mit anderen Themengebieten gelegt. Auch wenn nicht strikt erforderlich, sind hier insbesondere solche Ansätze von Interesse, die eine Auswertung bereits zur Laufzeit ermöglichen. Dazu wird zunächst betrachtet, wie RV in der Literatur beschrieben wird. Im Allgemeinen wird dazu das ausgeführte System durch einen Monitor beobachtet. Leucker und Schallhart [LS09] beschreiben RV als passives Testen mit einem Monitor, der überprüft, ob ein bestimmter Lauf des SuO ein bestimmtes Korrektheitsmerkmal (engl.: *correctness property*) erfüllt oder verletzt. Der Monitor nutzt dazu eine Spezifikation des Merkmals und prüft, ob diese von dem SuO eingehalten wird. Ein Merkmal ist dabei eine Partition der Traces des Systems und teilt diese in zwei Mengen oder eine detailliertere Klassifizierung auf.

Ein Monitor wird von Delgado, Gates und Roach [DGR04] in zwei Komponenten aufgeteilt: Beobachter (engl.: *Observer*) und Analyst (engl.: *Analyzer*). Der Beobachter verfolgt die Zustände und Zustandsänderungen des SuO. Ein Zustand kann dabei eine beliebige Kombination der Informationen sein, die aus dem SuO ausgelesen werden können. Interessante Beobachtungen gibt er an den Analysten weiter. Dieser wertet aus, ob es sich dabei um erwartetes Verhalten handelt. Diese Auswertung gibt der Monitor nach außen weiter, damit diese protokolliert oder darauf reagiert werden kann. Der Ablauf wird in Abbildung 3.2 veranschaulicht. Der Einfluss durch die Überprüfung auf das SuO wird durch dieses Vorgehen minimiert, da sich der Einfluss auf das reine Beobachten beschränkt. Da das SuO in seiner normalen Funktionsweise beobachtet wird, entfällt die Notwendigkeit einen Testkontext zu erstellen. Zudem eignet sich das Verfahren auch sehr gut für die Überwachung von produktiv eingesetzten Anlagen.

Es gibt eine Vielzahl von Ansätzen und RV Werkzeugen [Fal+18], die auf bestimmte Problemstellungen eingehen. Wenn es sich beispielsweise bei den Abweichungen um Lücken in der Beobachtung handelt, kann ein *Hidden Markov Model* verwendet werden, um die Zustände zur Überprüfung abzuschätzen [Sto+12]. Viele RV Werkzeuge, wie z. B. TraceMatches [All+05] oder JavaMOP [Mer+11], sehen einen Vorverarbeitungsschritt der

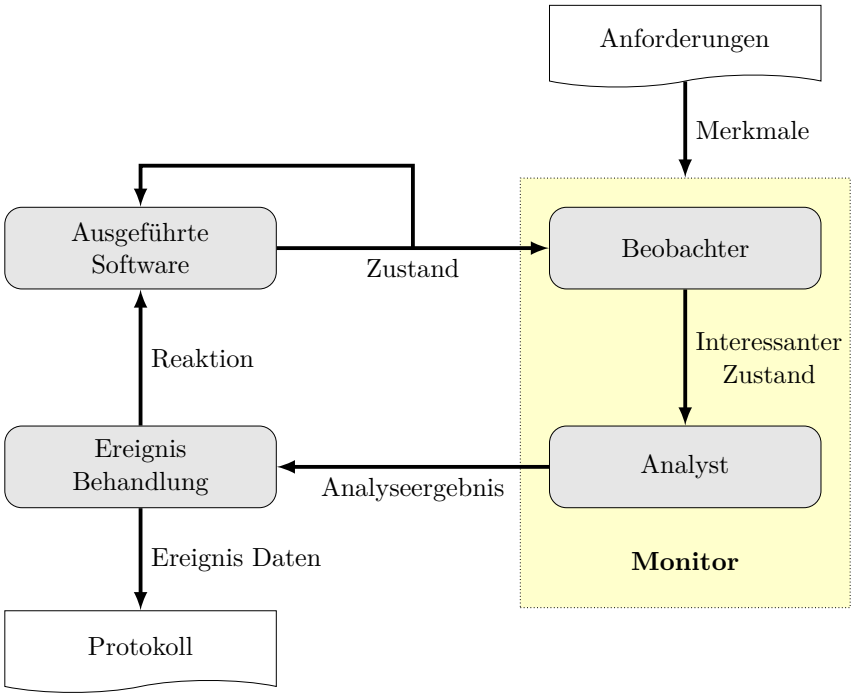


Abb. 3.2: Aufbau eines Monitors nach Delgado, Gates und Roach [DGR04].

Eingabedaten vor und filtern sie, bevor sie an eine Instanz des Monitors übergeben werden. Durch solche Methoden zum Schneiden von Traces mit Parametern (engl.: *parametric trace slicing or tracecuts*) sieht jeder Monitor nur noch die für ihn relevanten Ereignisse [All+05; CK13]. Diese Methoden dienen dazu ein Trace aufzuspalten, wenn verschiedene Instanzen eines Merkmals mehrmals in der Ausführung auftauchen und jede Instanz unabhängig betrachtet werden soll.

Cimatti, Tian und Tonetta beschreiben eine Erweiterung von RV um Rücksetzen [CTT19]. Dabei wird der Bezugspunkt einer in LTL beschriebenen Prüflogik auf das aktuelle Ereignis gesetzt, die Historie bleibt aber erhalten. Neben der Prüflogik beziehen sie auch Annahmen in die Überprüfung mit ein. Durch Erhalt der Historie können die Annahmen überprüft und gleichzeitig ihr zusätzliches Wissen genutzt werden. Einige zu überprüfende Merkmale werden dadurch vorhersagbar und andere, sonst nicht-beobachtbare, können beobachtet werden. In ihrem Beispiel verwenden sie als Annahme, dass ein bestimmtes Merkmal nur einmal erfüllt sein kann und prüfen, ob es nie eintreten wird. Der positive Fall kann normalerweise nicht beobachtet werden. Konnte die Prüfung aber durch Beobachtung des Merkmals bereits widerlegt werden, kann die Prüflogik nach dem Rücksetzen das nicht Auftreten vorhersagen, indem sie die Annahme verwendet. Wird das Merkmal dennoch erneut erfüllt, kann in diesem Fall eine Verletzung der Annahme erkannt werden. Wenn die Annahme verletzt wird, kann die Überprüfung aber auch nach Rücksetzen nicht fortgesetzt bzw. nur die verletzte Annahme gemeldet werden.

Das Themengebiet komplexe Ereignis-Verarbeitung (CEP, engl.: *Complex Event Processing*) befasst sich nicht direkt mit Testverfahren für Software. Eckert und Bry [EB09] beschreiben es als „[...] ein Sammelbegriff für Methoden, Techniken und Werkzeuge, um Ereignisse zu verarbeiten *während sie passieren*, also kontinuierlich und zeitnah. CEP leitet aus Ereignissen höheres, wertvolles Wissen in Form von sog. komplexen Ereignissen, d.h. Situationen die sich nur als Kombination mehrerer Ereignisse erkennen lassen, ab." Dazu definiert es abstrakte Konzepte, wie virtuelle Ereignisse, die nicht in der physischen Welt auftreten, aber modelliert, simuliert oder sich vorgestellt werden können [LS11]. CEP beschäftigt sich mit der Extraktion von Informationen aus verschiedenen Datenströmen zur Laufzeit.

Typische Anwendungsgebiete für CEP sind die Überwachung von Geschäftsprozessen, die Auswertung von Sensor Netzwerken oder die Analyse von Marktdaten [EB09]. Nach [Hal16] formuliert es die Verarbeitung der Ereignisse eines Traces als ein Datenbank-Problem. Ein Trace wird dabei als eine dynamische Datenquelle betrachtet, auf der Anfragen (engl.: *queries*) ausgeführt werden, um Ergebnisse zu extrahieren. Häufig werden solche Anfragen in SQL oder einem davon abgeleiteten Dialekt beschrieben. Teilweise wird die Fähigkeit dazu auch als notwendige Anforderung für solche Werkzeuge gesehen [SÇZ05]. Im Gegensatz zu klassischen Datenbanken wird die Anfrage nicht nur einmal ausgeführt, sondern kontinuierlich auf neu eintreffende Daten angewendet. Um den Datenstrom als Relation betrachten zu können, werden verschiedene Fensteroperationen wie z. B. *alle Ereignisse der letzten Stunde* verwendet [LS11].

BeepBeep 3 [Hal16] ist ein Werkzeug zur Ereignis-Verarbeitung, das versucht, die Themengebiete RV und CEP zu kombinieren. Es besteht aus modularen Prozessoren, die, unter Berücksichtigung ihrer Schnittstellen, frei kombiniert werden können, um eine Anfrage zu formulieren. Die Prozessoren bilden dabei eine Art Pipeline, durch die der Datenstrom bei der Verarbeitung fließt. Die Pipeline kann entweder direkt über (Java-)Code aufgebaut oder über BeepBeeps eigene Anfragesprache für Ereignisströme *eSQL* (engl.: *Event Stream Query Language*) definiert werden. BeepBeep bietet Prozessoren für Standardoperationen des CEP an, wie Fensteroperationen und Aggregatoren. Zudem verfügt es über Paletten mit weiteren Prozessoren, die beispielsweise alle Operationen des in der Verifikation zur Laufzeit häufig verwendeten Formalismus LTL nutzbar machen oder die Verwendung eines (Moore-)Automaten ermöglichen. Dadurch kann das Werkzeug ein breites Band an Ausdrücken auswerten.

3.3 Fazit

Dieses Kapitel hat den in dieser Arbeit verfolgten Ansatz zur Absicherung interaktiver Software-intensiver Systeme in die RV Taxonomie eingeordnet, um existierende Ansätze mit diesen Anforderungen vergleichen zu können. Das Ziel besteht darin, alle Unterschiede zwischen SuO und Spezifikation durch automatische Erzeugung eines Monitors zur Laufzeit zu finden. Hierbei wird angenommen, dass die Spezifikation das positive Soll-Verhalten beschreibt. Anschließend wurden existierende Verfahren für den dynamischen Software-Test betrachtet. Dazu wurde zwischen *Out-of-Service* und *In-Service* Verfahren unterschieden. Mit *Out-of-Service* Verfahren können bereits alle Unterschiede aufgedeckt werden, allerdings können diese per Definition nicht als Grundlage für Reaktionen zur Laufzeit, wie in Ziel 2 gefordert, verwendet werden. Die betrachteten *In-Service* Verfahren können mehrere Abweichungen entdecken, benötigen dazu aber eine entsprechend (manuell) aufgespaltene Spezifikation des erwarteten Verhaltens, eine Definition zum Schneiden von Traces oder eine explizite Beschreibung der Abweichungen. Die Ableitung solcher Merkmale aus der Verhaltensbeschreibung [HMF14] oder aus laufenden Systemen und deren Traces [CW99; RS93; BJR06; DGP15; RBR15] ist im Allgemeinen aber nicht trivial oder benötigt ein bereits korrekt funktionierendes System.

In dieser Arbeit wird daher betrachtet, wie Monitore, die auf einer Beschreibung des erwarteten Verhaltens in Form von Zustandsautomaten basieren, erweitert werden können, um alle erkennbaren Unterschiede zum beobachteten Verhalten zur Laufzeit melden zu können. Dieser Ansatz kann als komplementär zu bestehenden Ansätzen gesehen werden, da er eine Möglichkeit zur Wiederaufnahme der Überprüfung bietet, die auf beliebige, mit Automaten beschreibbare, Merkmale angewendet werden kann. Im nächsten Kapitel wird dazu zunächst die verwendete Modellierung des Soll-Verhaltens vorgestellt. Die Erweiterung der Automaten wird dann in Kapitel 5 hergeleitet und in Abschnitt 7.2 evaluiert. Dazu wird untersucht, wie sich verschiedene Annahmen über mögliche Abweichungen auf die Genauigkeit und Vollständigkeit der erzeugten Monitore auswirken.

Kapitel 4

4 Modellierung des Soll-Verhaltens

Eine Erfassung und Beschreibung des erwarteten Verhaltens ist Voraussetzung und Grundlage für jede Überprüfung eines Systems. Nur wenn definiert ist, welches Verhalten erwartet wird, kann dies auch verifiziert werden. Wie bereits in den vorherigen Kapiteln beschrieben, gibt es verschiedene Möglichkeiten eine Spezifikation anzugeben. Die verfügbaren Sprachen unterscheiden sich darin, welche Aspekte des Systems sie erfassen und wie diese dargestellt werden.

In diesem Kapitel wird eine Modellierungsmethodik vorgestellt, die im Rahmen verschiedener Forschungsprojekte unter Beteiligung des Autors dieser Arbeit entstanden ist, sowie weiterentwickelt und erprobt wurde. Beispielsweise hat sich eines der Forschungsprojekte mit dem Aufbau einer Werkzeugplattform zur Absicherung von Infotainment- und Fahrerassistenzfunktionen beschäftigt, ein anderes mit dem Aufbau einer Plattform zur automatisierten Überwachung eingebetteter Systeme.

Der in diesem Kapitel vorgestellte Ansatz zur Modellierung wurde bereits in wissenschaftlichen Veröffentlichungen [Pra+13; Dra+13; DPW15; DW17] beschrieben. Besonderer Fokus liegt hier auf einer Beschreibung der Konzepte, ihres Zusammenhangs und ihrer Formalisierung. Die entwickelte Methodik wurde vom Autor erweitert, um die Anforderungen zu erfüllen, die im nächsten Abschnitt beschrieben werden. Danach wird eine Übersicht über die verschiedenen Schichten des Modells gegeben, die anschließend im einzelnen genauer erläutert werden.

Dieses Kapitel geht hauptsächlich auf die Semantik der Modellierung ein. Die zur Realisierung des Prototyps verwendete konkrete und abstrakte Syntax wird in Kapitel 6 vorgestellt.

4.1 Anforderungen

In diesem Abschnitt werden die Anforderungen an die Modellierungsmethodik und damit erstellte Modelle zusammengefasst, die sich aus der Zielsetzung (vgl. Abschnitt 1.3) ergeben. Die entstandene Modellierungsmethodik zur Erfüllung dieser Anforderungen wird in den folgenden Abschnitten beschrieben.

In einem verteilten System können einzelne Komponenten als eine Art Black-Box vorliegen, d. h. ohne eine Beschreibung des internen Verhaltens, beispielsweise um IP-Rechte zu schützen. Damit diese Komponenten mit anderen interagieren können, sind die Schnittstellen, über die sie kommunizieren, definiert. Deshalb setzt die Absicherung bzw. Überwachung an dieser Stelle an und die Modellierung der Spezifikation erfasst das Kommunikationsverhalten. Dies hat zudem den Vorteil, dass Kommunikation häufig mit wenig Aufwand und Einfluss auf das beobachtete System instrumentiert werden kann.

Eine Übersicht der Komponenten eines Systems soll die Struktur ihrer Kommunikationsbeziehungen darstellen. Für jede der Komponenten werden dort die verwendeten Schnittstellen und die Kommunikationspartner benannt. Das Verhalten jeder Kommunikationsbeziehung soll genauer beschrieben werden können. Ebenso muss Expertenwissen über ein komplexeres Zusammenspiel mehrerer Komponenten erfassbar sein. Dazu ermöglicht die Modellierungsmethodik eine abstrakte Beschreibung des Kommunikationsverhaltens, in der die folgenden (Un-)Abhängigkeiten zwischen verschiedenen Nachrichten dargestellt werden können: Reihenfolge, Zeit-Eigenschaften oder Beschränkungen, sowie funktionale Beziehungen der Daten zweier Nachrichten. Die Abhängigkeiten können an einen bestimmten Kontext der Kommunikation gebunden sein. Letztendlich muss mit dem Modell eine automatisierte Überwachung eines laufenden Systems hinsichtlich der Einhaltung des modellierten Verhaltens ermöglicht werden können.

Die Modellierung soll für die Betrachtung und Verwendung durch verschiedene Benutzer geeignet sein. Dies bedeutet, dass die grundlegende Funktionsweise eines beschriebenen Systems ohne viel Vorwissen ersichtlich ist, das Verhalten aber in der gewünschten Genauigkeit mit dem Modell eindeutig beschrieben werden kann. Dazu soll eine grafische Darstellung des Verhaltens in Form von Automaten angeboten werden. Automaten sind verständlich, weit verbreitet, lernbar, diagnostizierbar, geeignet für die Verifikation und leicht zu bearbeiten, insbesondere im Vergleich mit anderen Formalisierungen wie Hidden-Markov-Model, Petri-Netzen oder Regel-basierten Systemen [MNE15].

Da diese Anforderungen auch in die Anforderungen an den Ansatz dieser Arbeit eingeflossen sind, decken sie sich mit den bereits in Abschnitt 3.1 für die Definition einer Spezifikation aufgestellten. Die Anforderungen können entsprechend der Taxonomie der Verifikation zur Laufzeit wie folgt zusammengefasst werden: Die Spezifikation muss explizit sein und eine

operationale Beschreibung mit Automaten ist zu bevorzugen. Logische und physische Zeit muss respektive mit partieller Ordnung und Zeitschranken beschrieben werden können. Vergleiche zwischen einzelnen Elementen von strukturierten oder komplexeren Daten müssen möglich sein, sowie einfache arithmetische Operationen auf Zahlen-basierten Datentypen. Die Spezifikation beschreibt das Soll-Verhalten, eine Erweiterung um explizite Verdikte sollte aber möglich sein.

4.2 Schichten zur Abstraktion

Der Fokus der Modellierungsmethodik liegt darauf, das dynamische Kommunikationsverhalten, also die Interaktionen zwischen verschiedenen Komponenten in dem System unter Beobachtung (SuO, engl.: *System under Observation*) zu beschreiben. Um einen präzisen Bezug zwischen realem (physischem) und modelliertem Verhalten herstellen zu können, muss dabei, wie in Herausforderung 1 erörtert, auch mit den Details der verwendeten Technologien umgegangen werden. Damit dennoch die eigentliche Verhaltensbeschreibung von diesen Details entkoppelt wird, ist das Modell in verschiedene Schichten aufgeteilt. Sie abstrahieren von den physischen Gegebenheiten und erlauben so eine kompakte Beschreibung des Verhaltens. Neben der Reduktion technischer Details in der Verhaltensbeschreibung führt dies auch zu einer leichteren Wiederverwendbarkeit und Erweiterbarkeit der einzelnen Schichten. Die Modellierungsmethodik besteht aus den Schichten **Verhalten**, **Ereignisse**, **Schnittstelle** und **Topologie**. Die verschiedenen Schichten sind jeweils auf die Beschreibung von bestimmten Aspekten des Systemverhaltens ausgelegt und ergänzen sich gegenseitig. Ihr Zusammenspiel wird in Abbildung 4.1 illustriert. Eine detailliertere Beschreibung der einzelnen Schichten folgt in den nächsten Abschnitten.

Die Topologie beschreibt die Komponenten des SuO und die Kommunikationsbeziehungen zwischen ihnen. Jede Komponente bietet eine oder mehrere Schnittstellen an oder benötigt sie von anderen Komponenten. Auch eine beliebige Mischung von beidem ist möglich. Eine Schnittstelle bestimmt die Elemente der Interaktionen zwischen den Komponenten. Allgemein handelt es sich dabei um die Nachrichtentypen mit ihren Parametern, die eine Komponente verschicken und empfangen kann. Die Ereignisse geben Nachrichten mit bestimmten Parameterbelegungen eine semantische Bezeichnung. Dabei bildet der Parameterbereich, der nicht genauer unterschieden wird, die so bezeichnete *Äquivalenzklasse* des Ereignisses. Ereignisse werden in der Beschreibung des Verhaltens verwendet, um erlaubte Reihenfolgen der Interaktionen zu definieren. Dabei können auch Ereignisse von verschiedenen Schnittstellen in einem Verhalten verwendet werden. Ein solches Referenz-Modell des Verhaltens beschreibt im Kern das gültige Verhalten und schließt nur kritische oder beispielhafte Abweichungen ein.

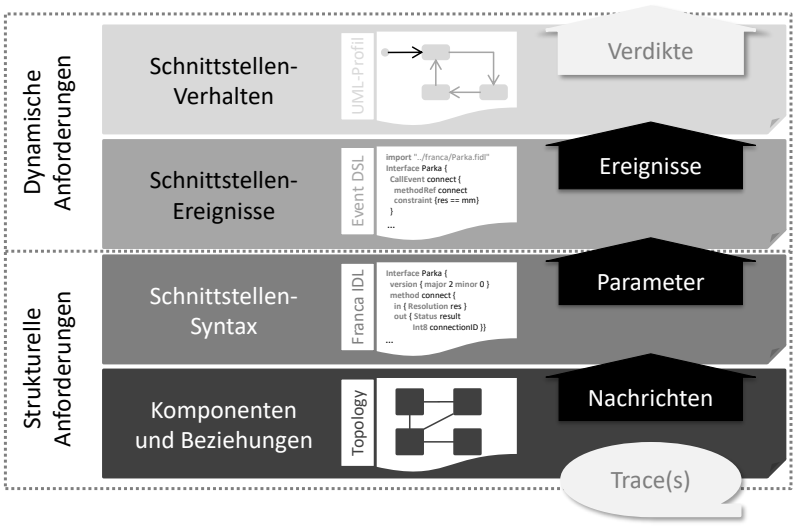


Abb. 4.1: Übersicht der Schichten in der Modellierung

Ein Modell der Interaktionen ergibt sich aus einer Kombination der Elemente aller Schichten. Es ist ein Tupel $\langle \mathcal{O}, \mathbb{I}, \Psi, A \rangle$. $\mathcal{O}$ bestimmt dabei den Ort innerhalb der Topologie, der mit dem Modell betrachtet wird. Dies kann beispielsweise eine bestimmte Kommunikationsbeziehung oder eine Menge von Komponenten sein. $\mathbb{I}$ ist die Menge der relevanten Schnittstellendefinitionen und Ψ die Menge der entsprechenden Abbildungen auf Ereignisse. Die Interaktionen sind mit diesen Ereignissen dann in dem Automaten A spezifiziert. Dabei wird lediglich das positive Soll-Verhalten angegeben. Ausnahmen, negatives Verhalten oder weitere Bewertungen des Verhaltens können beschrieben werden, wenn die Bewertungsrelation der Automaten explizit angegeben wird (vgl. Abschnitt 2.2.5).

4.3 Topologie

Die Topologie beschreibt die logische Struktur des Systems. Sie definiert, welche Komponenten das System enthält und wie diese miteinander verbunden sind. Sie trifft aber keine Aussagen über deren Verhalten. Mit ihr wird ermöglicht, den *Ort*, der beschrieben bzw. beobachtet wird, genauer zu bestimmen. Ein solcher Ort ist ein zusammenhängender Teil der Topologie, also eine Gruppe von Komponenten, die über Kommunikationsbeziehungen verbunden sind. Dies ist beispielsweise eine Kommunikationsbeziehung, eine Komponente mit ihren Kommunikationspartnern oder eine beliebige Gruppe von Komponenten. In Abbildung 4.2 werden die Elemente der Topologie illustriert und der folgende Absatz definiert sie formal.

Ein *Ort* wird beschrieben durch ein Tupel $\mathcal{O} = \left\langle \mathbb{C}^{\mathcal{O}}, \Upsilon^{\mathcal{O}} \right\rangle$. Dabei sei $\mathbb{C}^{\mathcal{O}} \subseteq \mathbb{C}$ eine (Teil-)Menge der Komponenten und $\Upsilon^{\mathcal{O}} \subseteq \Upsilon$ eine Menge von Kommunikationsbeziehungen

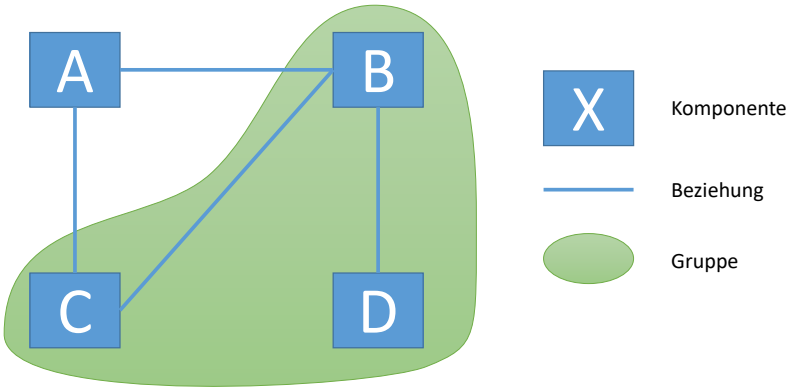


Abb. 4.2: Illustration der Elemente der Topologie

zwischen diesen. Die Relation $\Upsilon^{\mathcal{O}} \subseteq \mathbb{C}^{\mathcal{O}} \times \mathbb{C}^{\mathcal{O}}$ enthält genau dann $\langle c_1, c_2 \rangle$, ein Paar von Komponenten, wenn eine Kommunikationsbeziehung zwischen c_1 und c_2 besteht. Eine Richtung der Kommunikationsbeziehung wird dabei nicht berücksichtigt, d. h. die Relation ist symmetrisch. Der Ort $\langle \mathbb{C}, \Upsilon \rangle$ enthält alle Komponenten und Kommunikationsbeziehungen der Topologie.

Eine *Komponente* ist ein Bestandteil eines Systems. Sie interagiert über Schnittstellen mit dem restlichen System, d. h. den anderen Komponenten des Systems. Im Rahmen dieser Arbeit wird eine Komponente als unteilbar angesehen. Zudem wird ihre Implementierung als unbekannt angenommen. Das bedeutet, dass die Modellierung keine weitere Detaillierung des inneren Aufbaus einer Komponente vorsieht oder benötigt. Beschrieben wird lediglich das erwartete Verhalten der Interaktionen mit anderen Komponenten des Systems. Prinzipiell kann aber eine Komponente als System betrachtet und dann genauer beschrieben werden. Eine Überprüfung von Systemen, die aus Systemen aufgebaut sind, geht aber über den Rahmen dieser Arbeit hinaus und wird nicht weiter betrachtet.

Jede Komponente $c \in \mathbb{C}$ kann Schnittstellen benötigen und anbieten. Diese Mengen werden entsprechend mit $\mathbb{I}^{\overleftarrow{c}}$ und $\mathbb{I}^{\overrightarrow{c}}$ bezeichnet. Für die weitere Betrachtung sind ebenso die daraus abgeleiteten Ereignisse relevant, die eine Komponente über ihre Schnittstellen entgegennimmt und auslöst. Sie werden mit $\mathbb{E}^{\overleftarrow{c}}$ und $\mathbb{E}^{\overrightarrow{c}}$ referenziert.

Eine *Kommunikationsbeziehung* beschreibt die Möglichkeit der Interaktion zwischen den beiden durch sie verbundenen Komponenten. Sie ist eine logische Verbindung, d. h. die verbundenen Komponenten sind in der Lage Nachrichten auszutauschen. Es bedeutet nicht zwangsläufig, dass im realen System eine direkte physische Verbindung zwischen den beiden Komponenten besteht, beispielsweise in Form eines Kabels. Da in dieser Arbeit eine Überprüfung der dynamischen Aspekte von Interaktionen im Vordergrund steht, wird zur Vereinfachung der Beschreibung angenommen, dass eine Komponente jede kompatible Schnittstelle in ihre Kommunikationsbeziehungen einbringt. Genau die Schnittstellen

$I \in \mathbb{I}^{c_1 \cap c_2}$ sind kompatibel mit einer Kommunikationsbeziehung $\langle c_1, c_2 \rangle$. Bei $\mathbb{I}^{c_1 \cap c_2}$ handelt es sich um die Schnittmenge der angebotenen und benötigten Schnittstellen der beteiligten Komponenten, definiert in Gleichung 4.1.

$$\mathbb{I}^{c_1 \cap c_2} = \left(\mathbb{I}^{\overrightarrow{c_1}} \cap \mathbb{I}^{\overleftarrow{c_2}}\right) \cup \left(\mathbb{I}^{\overrightarrow{c_2}} \cap \mathbb{I}^{\overleftarrow{c_1}}\right) \tag{4.1}$$

4.4 Schnittstellenbeschreibung

Eine Komponente kommuniziert mit ihrer Umgebung über bestimmte Schnittstellen. Eine *Schnittstelle* definiert die Struktur der Einheiten mit denen eine Kommunikation möglich ist. Die *Schnittstellenbeschreibung* legt den Fokus auf eine logische Beschreibung dieser *Nachrichten*. Sie definiert, welche *Nachrichtentypen* in dem System auftauchen können, aber nicht, wie diese Bit-für-Bit aufgebaut sind. In verteilten Systemen werden Nachrichten häufig entsprechend bestimmter Muster gebildet, wie beispielsweise der Aufruf einer Methode und Rückgabe des Ergebnisses. Eine Schnittstellenbeschreibungssprache (IDL, engl.: *Interface Definition Language*), wie z.B. Franca IDL [Franca], beschreibt deswegen häufig nicht direkt einzelne Nachrichtentypen, sondern höhere Konzepte; beispielsweise die verfügbaren *Methoden*, *Broadcasts* und *Attribute* der Schnittstelle, sowie deren Parameter. Die möglichen Nachrichtentypen können daraus abgeleitet werden. Eine Komponente, die eine Schnittstelle benötigt, wird im Folgenden **Client** genannt. Eine Komponente, die diese Schnittstelle anbietet, wird als **Server** bezeichnet. Eine Komponente kann gleichzeitig Server und Client für verschiedene und auch für die gleiche Schnittstelle sein. Ein Client kann über die Schnittstelle Methoden aufrufen, Broadcasts und Antworten auf Methodenaufrufe empfangen, sowie Attribute abfragen und manipulieren. Der Server stellt die entsprechende Gegenstelle zur Verfügung.

Ziel der Modellierungsmethodik dieser Arbeit ist, den Einfluss von technischen Details auf die Modellierung der Applikationslogik zu reduzieren (vgl. Ziel 1). Je nach Kommunikationsmedium variieren die verfügbaren Mechanismen zur Realisierung der Nachrichten. Die Beschreibung von Nachrichtentypen mit höheren Konzepten fördert daher eine Trennung der Applikationslogik von solchen Details. Aus der Schnittstellenbeschreibung kann z. B. Code für das konkrete Medium generiert werden, der den Aufruf dieser Mechanismen kapselt. So wird auch bei der Implementierung eine Unabhängigkeit von dem gewählten Medium garantiert. Stellt das Kommunikationsmedium die gewünschten Mechanismen nicht selbst zur Verfügung, können sie normalerweise mit anderen Nachrichten nachgebaut werden. Zudem ist das Wissen über den Zusammenhang bestimmter Nachrichten und Nachrichtentypen wichtig. Beispielsweise ist eine Antwort immer einem bestimmten vorangegangenem Aufruf zugeordnet. Ohne diesen Zusammenhang kann kein generischer

Mechanismus verwendet werden, der dem Aufrufer die richtige Antwort zustellt oder dies überprüft. Um dennoch jeden Nachrichtentyp individuell mit einer IDL zu definieren, kann in diesem Fall die Beschreibung beispielsweise auf Methodenaufrufe ohne Antwort beschränkt werden.

Es seien $\mathbb{M}$ die Menge aller Nachrichten und $\mathbb{P}$ die aller Parameter. Eine *Schnittstelle* wird in dieser Arbeit als ein Tupel $I = \left\langle \mathbb{M}^{\overleftarrow{I}}, \mathbb{M}^{\overrightarrow{I}}, \succ^I, \mathbb{P}^I \right\rangle$ betrachtet. Es wird unterschieden, welche Nachrichtentypen Client ($\mathbb{M}^{\overleftarrow{I}} \subset \mathbb{M}$) und Server ($\mathbb{M}^{\overrightarrow{I}} \subset \mathbb{M}$) jeweils verschicken können. Die Relation $\succ$ gibt einen Zusammenhang mit einem anderen Nachrichtentyp an. Da in dieser Arbeit eine eindeutige Zuordnung von Nachrichtentypen zu Schnittstellen angenommen wird, kann eine einheitliche Relation $\succ$ gebildet und die Notation vereinfacht werden (4.2). Beispielsweise setzt eine Antwort des Servers normalerweise eine Anfrage des Clients voraus (4.3). Die Relation $\mathbb{P}^I$ ordnet jeder Nachricht $m \in \mathbb{M}^{\overleftarrow{I}} \cup \mathbb{M}^{\overrightarrow{I}}$ eine Menge $\mathbb{P}^m \subset \mathbb{P}$ zu. Sie beschreibt, welche Parameter mit dieser Nachricht übertragen werden.

$$\succ = \bigcup_{I \in \mathbb{I}} \succ^I \tag{4.2}$$

$$m_{\text{Anfrage}} \in \mathbb{M}^{\overleftarrow{I}}, m_{\text{Antwort}} \in \mathbb{M}^{\overrightarrow{I}} : \qquad m_{\text{Anfrage}} \succ m_{\text{Antwort}} \tag{4.3}$$

Unabhängig davon, wie die Schnittstelle mit der IDL beschrieben wird, werden damit Nachrichtentypen der Schnittstelle zugeordnet. Wie der zusammenhängende Nachrichten zugeordnet werden hängt von der Realisierung der Schnittstelle ab und ist deshalb nicht Inhalt der logischen Beschreibung. Komplexere und optionale Zusammenhänge zwischen Nachrichten werden ebenfalls nicht über die Schnittstellenbeschreibung erfasst, sondern in der Verhaltensbeschreibung.

4.5 Ereignisbeschreibung

Während die Schnittstellenbeschreibung definiert, welche Nachrichtentypen an einer Schnittstelle auftreten und deren Parameter, betrachtet die *Ereignisbeschreibung* die Parameterwerte einer Nachricht. Ein *Ereignis* beschreibt die Menge semantisch äquivalenten Belegungen der Parameter einer Nachricht. Es dient als *Äquivalenzklasse* zur Beschreibung der Interpretation der Inhalte einer Nachricht. Dies bezieht sich insbesondere darauf, welchen Einfluss das Ereignis auf den Kontrollfluss hat, um als Grundlage für die Modellierung des Verhaltens zu dienen. Die Bedeutung kann sich beispielsweise aus direkten Steuerbefehlen ergeben, die in den Parametern einer Nachricht statt als eigene Nachrichtentypen kodiert sind. Damit bleibt die statisch definierte Schnittstellenbeschreibung identisch, das dynamische Verhalten

kann aber auch von den Parametern der Nachricht abhängen und durch hinzunehmen von neuen Parameterwerten um neue Aktionen erweitert werden. Durch Definition von Ereignissen für gültige Wertebereiche und Kombinationen der Parameterwerte werden die Parameter bekannter Aktionen detaillierter überprüft. Die Integrität hat Auswirkungen auf den Kontrollfluss, spätestens wenn entsprechende Ausnahmebehandlungen erwartet werden. Mit Ereignissen kann die Beschreibung der Nachrichtentypen um Bereiche der Parameter verfeinert werden. Ereignisse sind weniger dafür gedacht, funktionale Beziehungen zwischen den Parametern verschiedener Nachrichten zu prüfen. Zum Beispiel sollten Ereignisse nicht prüfen, ob eine bestimmte angeforderte Lautstärke auch angewendet wurde, sondern definieren, wie eine gültige Anfrage und Antwort aussehen. Im Fall der Lautstärke könnte der Wert vermutlich auch über entsprechend fein definierte Ereignisse abgebildet werden, eine Beschreibung mit den in Abschnitt 4.6.2 vorgestellten Kontexten ist jedoch kompakter.

4.5.1 Hierarchie von Ereignissen

Die Größe der Äquivalenzklassen bestimmt, wie detailliert das Verhalten in der Verhaltensbeschreibung unterschieden werden kann. Um hier einen Kompromiss aus Strukturierung und Flexibilität bei der Beschreibung zu ermöglichen, sind die Ereignisse in einer Hierarchie angeordnet. Ein Wurzel-Ereignis für einen Nachrichtentyp hat keine Einschränkungen für die Belegung der Parameter. Die jeweils nächste Ebene an Elementen partitioniert die möglichen Parameterbelegungen mit Bedingungen. Dies kann wiederholt werden, bis der gewünschte Detaillierungsgrad erreicht ist. Das übergeordnete Ereignis schließt immer alle ihm untergeordneten mit ein. Diese Anordnung ermöglicht zudem eine effiziente Auswertung und Zuordnung eines Ereignisses zu einer Nachricht.

Eine *Ereignisbeschreibung* definiert die Teilmenge der Ereignisse $\mathbb{E}^m \subset \mathbb{E}$, die einem Nachrichtentyp m zugeordnet sind. Die Teilmengen für alle Nachrichtentypen ergeben eine Partition aller Ereignisse. Damit ist ein Ereignis immer genau einem Nachrichtentyp zugeordnet. Die Hierarchie der Ereignisse ist durch eine Relation $e_1 \triangleright e_2$ beschrieben, die e_2 als Spezialisierung von e_1 kennzeichnet. Ein Ereignis kann immer nur ein anderes spezialisieren, es kann aber verschiedene Spezialisierungen eines Ereignisses geben. Hierdurch ergibt sich ein Baum der Ereignisse in $\mathbb{E}^m$, wie durch ein Beispiel in Abbildung 4.3 dargestellt wird. Um Beziehungen zwischen Ereignissen über mehrere Hierarchieebenen hinweg zu beschreiben, wird die reflexiv-transitive Hülle der Relation $\triangleright$ verwendet. Sie wird mit $\overset{*}{\triangleright}$ bezeichnet (4.4). Damit beschreibt $e_1 \overset{*}{\triangleright} e_3$, dass e_1 alle Nachrichten abdeckt, die auch durch e_3 repräsentiert werden. Sind die beiden Ereignisse unterschiedlich, wird e_1 als *allgemeiner* und e_2 als *spezieller* mit Bezug auf das jeweils andere Ereignis bezeichnet. Je m gibt es genau ein sog. Wurzel-Ereignis, das allgemeiner als alle anderen Ereignisse in $\mathbb{E}^m$ ist.

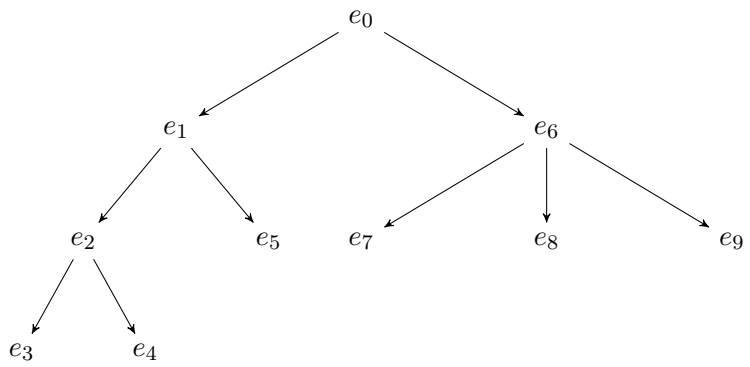


Abb. 4.3: Beispiel der Hierarchie von Ereignissen für einen Nachrichtentyp.

Ein sog. Blatt-Ereignis hat keine weitere Spezialisierung, im Beispiel in Abbildung 4.3 sind dies e_3, e_4, e_5, e_7, e_8 und e_9. Die Relation zwischen Blatt und Partitionierung der Parameterbelegungen wird mit ψ^m bezeichnet.

$$e_1 \overset{*}{\triangleright} e_3 \quad \Longleftrightarrow \quad e_1 = e_3 \ \vee \ e_1 \triangleright e_3 \ \vee \ (e_1 \overset{*}{\triangleright} e_2 \wedge e_2 \triangleright e_3) \tag{4.4}$$

4.5.2 Abbildung von Nachrichten auf Ereignisse

Ein Ereignis repräsentiert eine Äquivalenzklasse von Nachrichten. Diese Betrachtungsweise vereinfacht die weitere Auswertung von Nachrichtensequenzen, da dort nicht mehr einzelne Parameter betrachtet werden müssen. Dieser Abschnitt behandelt die Herausforderungen bei der Abbildung von Nachrichten auf Ereignisse.

Bei der Zuordnung eines Ereignisses zu einer beobachteten Nachricht sind verschiedene Aspekte zu beachten. Zunächst muss die Nachricht erfasst und ihre Parameter müssen auswertbar gemacht werden. Durch Interpretation der Schnittstellenbeschreibung oder mit daraus erzeugtem Code kann eine einheitliche Form zum Zugriff auf diese Daten bereitgestellt werden. Häufig können diese Daten auch direkt durch eine entsprechend konfigurierte Middleware oder Laufzeit-Bibliothek geliefert werden. Üblicherweise enthalten sie neben den Werten für die Parameter auch sog. Meta-Daten wie Zeitstempel, Nachrichtentyp, Sender und Empfänger. Für die Zuordnung der Ereignisse werden die Parameter gegen die Bedingungen der möglichen Ereignisse des Nachrichtentyps evaluiert. Für eine effiziente Auswertung kann auch die hierarchische Struktur der Ereignisse ausgenutzt werden. Die Abbildung erfolgt immer auf das spezifischste Ereignis, also ein Blatt der Hierarchie. Da die Blätter eine Partition des Parameterraums der Nachricht bilden, ist diese Zuordnung eindeutig.

Prinzipiell kann die Abbildung statisch oder zustandsbehaftet sein. Im zweiten Fall haben frühere Nachrichten einen Einfluss auf die Auswertung der aktuellen Nachricht, im statischen Fall nicht. Da der in dieser Arbeit vorgestellte Ansatz eine explizite Verhaltensmodellierung vorsieht, soll aber eine Beschreibung von Verhalten in der Ereignisbeschreibung vermieden werden. Ein häufiger Anwendungsfall, bei dem solch ein Zustand in der Ereignisauswertung hilfreich wäre, ist die Erkennung und Zuordnung des richtigen Kommunikationskontexts mit zugehöriger Belegung von Variablen in diesem Kontext. Diese Kommunikationskontexte werden, wie in Abschnitt 4.6.2 vorgestellt wird, deshalb gesondert modelliert und auf eine zustandsbehaftete Abbildung der Ereignisse wird verzichtet. Die hier betrachtete Abbildung von Nachrichten auf Ereignisse ist somit statisch. Dies bringt weitere Möglichkeiten zu Effizienzsteigerung mit, wie beispielsweise durch eine parallelisierte Auswertung.

$\mathcal{N}$ sei die Menge aller beobachtbaren Nachrichten und $\mathbb{V}$ die aller möglichen Parameterwerte. Eine *Nachricht* $\langle u, w\rangle = n \in \mathcal{N}$ enthält neben der Werte-Sequenz $w \in \mathbb{V}^*$ auch eine Sequenz an Meta-Daten u. Für jeden *Parameter* $p \in \mathbb{P}^m$ beinhaltet w an einer bestimmten Stelle i einen Wert $v_i \in \mathbb{V}^p$. Die Meta-Daten sind analog aufgebaut. Prinzipiell können sie wie Parameter behandelt werden. Die Unterscheidung ist allein dem unterschiedlichen Ort der Definition und Wert-Bildung geschuldet. Während Parameter von der Anwendung bestimmt werden, stammen die Meta-Daten von der Instrumentierung. In den Meta-Daten ist beispielsweise der Nachrichtentyp m enthalten. Die Abbildung von Nachrichten auf Ereignisse ist eine Funktion $\Psi : \mathcal{N} \mapsto \mathbb{E}$. Da die Ereignisse für jeden Nachrichtentyp unabhängig voneinander sind, kann sie aus deren Abbildungen zusammengesetzt werden (4.5). Die Abbildung für einen Nachrichtentyp, beschrieben durch $\psi^m : \mathbb{V}^* \mapsto \mathbb{E}$, wird bei Definition der Ereignisbeschreibung festgelegt, indem die Blätter der Ereignishierarchie eine Partition des Parameterraums beschreiben.

$$\Psi(\langle \ldots m \ldots, w\rangle) = \psi^m(w) \tag{4.5}$$

Die hier gewählte Darstellungsform kann den Eindruck erwecken, dass Werte atomar sind. Dies ist explizit nicht der Fall. Der Wertebereich eines Parameters kann also selbst auch strukturiert sein oder beispielsweise Listen von Elementen enthalten. Auf eine detaillierte Beschreibung wurde hier zur besseren Nachvollziehbarkeit verzichtet, da dies keine Auswirkungen auf die beschriebene Semantik hat. Die Implementierung der Verarbeitung der Parameter ist davon aber betroffen und muss beispielsweise Operatoren für den Zugriff auf untergeordnete Elemente bereitstellen.

4.6 Verhaltensbeschreibung

Die *Verhaltensbeschreibung* verbindet die Bausteine der anderen Schichten, insbesondere die Ereignisse. Ihre Aufgabe ist dabei sowohl eine logische Ordnung zwischen verschiedenen Ereignissen zu schaffen, als auch an deren Kombination geknüpfte nicht-funktionale Bedingungen, wie Zeitverhalten, zu definieren.

Entsprechend ihrem Namen definiert die Verhaltensbeschreibung das an einem bestimmten Ort $\mathcal{O}$ erwartete Verhalten. Der Ansatz dieser Arbeit sieht dabei vor, dass mögliche Abweichungen nicht explizit modelliert werden müssen. Sie können aber beschrieben werden, etwa als Beispiele oder um kritische Situationen hervorzuheben. Im Gegenzug bedeutet dies, dass das positive Soll-Verhalten bei $\mathcal{O}$ vollständig beschrieben werden muss, da jedes nicht beschriebene Verhalten als Abweichung interpretiert wird. Vollständig bedeutet dabei alle Vorkommen zu definieren, wann die Ereignisse der Schnittstellen $\bigcup_{\langle c_1,c_2\rangle\in\Upsilon^{\mathcal{O}}} \mathbb{I}^{c_1\cap c_2}$ erwartet werden. Ist in der Beschreibung erwartetes Verhalten nicht enthalten, wird dies bei Auftreten sonst als Abweichung erkannt. Diese Eigenschaft kann aber auch verwendet werden, um schrittweise das Soll-Verhalten zu erfassen, beispielsweise indem es automatisch gelernt und nach Freigabe in die Verhaltensbeschreibung integriert wird. Ebenfalls können andere Artefakte, beispielsweise Sequenzen aus der Spezifikation, zur Initialisierung einer Verhaltensbeschreibung verwendet werden. Erkannte Abweichungen sind dann dahingehend zu untersuchen, ob es sich um eine Lücke in der Beschreibung des Soll-Verhaltens handelt oder um Fehler des SuO. Da das positive Verhalten beschrieben wird, kann dieses auch für andere Zwecke weiterverwendet werden, beispielsweise eine Restbus-Simulation oder für die Generierung von Testfällen. Wie mit dieser Beschreibung alle erkennbaren Abweichungen des Ist-Verhaltens entdeckt werden können, wird in Kapitel 5 genauer beschrieben.

4.6.1 Sender und Empfänger

Der in dieser Arbeit verfolgte Ansatz zur Beschreibung von Soll-Verhalten verwendet eine erweiterte Interpretation der in Abschnitt 2.2.1 vorgestellten Zustandsautomaten $A = \langle \mathbb{B}, \mathbb{S}, s_0, \delta \rangle$. Dieser kann auch durch einen hierarchischen Zustandsautomaten definiert sein. Im Folgenden werden die Erweiterungen vorgestellt.

Der Automat kann das Zusammenspiel der Ereignisse von mehreren Kommunikationsbeziehungen kombiniert beschreiben. Selbst bei nur einem Paar von Komponenten kann es vorkommen, dass beide eine Schnittstelle sowohl anbieten als auch benötigen. In all diesen Fällen ist es notwendig, dass Sender und Empfänger der Nachricht berücksichtigt werden, um Verwechslungen zu vermeiden. Dies ist mit konstantem Aufwand in die Auswertung

der Ereignisse integrierbar. Um Sender und Empfänger auch überprüfen zu können, wird Ψ, die Abbildung von Nachrichten auf Ereignisse, erweitert, sodass sie den Sender sn und Empfänger rc aus den Meta-Daten extrahiert (4.6). Die Menge der Eingaben des Automaten ändert sich damit zu $\mathbb{B} \subseteq \mathbb{C} \times \mathbb{C} \times \mathbb{E}$ und eine Eingabe ist ein 3-Tupel $\langle sn, rc, e \rangle$. Entsprechend werden die Elemente der Menge von Beschriftungen $\mathbb{B}$ ebenfalls um diese beiden Werte erweitert. Eine Beschriftung ist damit das gleiche 3-Tupel $\langle sn, rc, e \rangle$. Das Verhalten des einfachen Automaten bleibt unverändert, bis auf die substituierte Menge an Eingaben.

$$\Psi(\langle \ldots m, sn, rc, \ldots, w \rangle) = \langle sn, rc, \psi^m(w) \rangle \tag{4.6}$$

Ein alternativer Ansatz besteht darin, Sender und Empfänger mit Kontexten zu erfassen. Diese werden im nächsten Abschnitt vorgestellt.

4.6.2 Kontexte

Der Unterschied zwischen korrektem und unerwartetem Verhalten kann von der Kompatibilität der Parameter zweier Nachrichten abhängen. Beispielsweise, wenn eine Antwort einen bestimmten Wert aus der zugehörigen Anfrage wiederholen soll. Allgemeiner formuliert, ist es von Interesse, bestimmte Nachrichten anhand ihrer Merkmale zu identifizieren, z. B. mit einer Kennzahl der Transaktion, und vom restlichen Trace isoliert zu betrachten. Die Abbildung der Nachrichten auf Ereignisse ist statisch, wie in Abschnitt 4.5.2 beschrieben. Prinzipiell kann dieser Zusammenhang auch mit (unendlich) vielen statischen Ereignissen modelliert werden. Dazu wird allerdings ein Satz Ereignisse pro möglichem Wert benötigt. Zudem muss die richtige Sequenz mit allen gültigen Varianten beschrieben werden. Deren Anzahl multipliziert sich für jede weitere Beziehung zwischen Parametern. Kontexte erlauben diese Zusammenhänge ergonomischer zu erfassen und ermöglichen eine speichereffiziente Darstellung mit nur kleinem Einfluss auf die Laufzeitkomplexität. Mit Kontexten kann, analog dem Schneiden von Traces mit Parametern [z. B. All+05; Mer+11], jeweils ein bestimmter Teil eines Traces gefiltert und untersucht werden. Im Unterschied zu den dort beschriebenen Ansätzen wird hier die Kontexterkennung strukturell von der restlichen Verhaltensüberprüfung getrennt.

Ein Kontext ist ein bestimmter Zusammenhang zwischen verschiedenen Nachrichten. Dazu legt er die Bedingung fest, wann eine Nachricht in ihn fällt, also für ihn relevant ist. Dies funktioniert ähnlich der Zuordnung von Nachrichten zu Ereignissen, allerdings ist diese Abbildung zustandsbehaftet und die Kontexte bilden keine Partition. Eine Nachricht kann also mehreren Kontexten zugeordnet sein, sie ist mit diesen *kompatibel*. Jeder Kontext hat seinen eigenen Zustand und dieser bleibt während der Auswertung einer Nachricht unverändert. Nur wenn ein Kontext *getroffen* wird, werden die entsprechenden Aktionen

anschließend auf seinen Zustand angewendet. Für einen Treffer muss eine kompatible Nachricht vom Kontext erwartet werden. Wann sie erwartet wird, ist durch die Verhaltensbeschreibung vorgegeben. Dies ist beispielsweise notwendig, um auch eine sequentielle Selektion des Kontexts zu ermöglichen. Das bedeutet aber, dass die Kontext-Auswertung synchron mit der Verhaltensbewertung ablaufen muss. Da Kontexte einen eigenen Zustand besitzen, benötigen sie aber ohnehin eine synchrone Auswertung. Beispiele für Aktionen auf einem Zustand sind das Merken oder Vergessen von Parametern. Da Kontexte nicht an eine Schnittstelle gebunden sind und auch als Verbindung zwischen statischer Ereignis- und dynamischer Verhaltensbeschreibung dienen, können hier auch die Meta-Daten mit in den Bedingungen und Aktionen verwendet werden.

Ein *Kontext* $\xi \in \Xi$ wird bestimmt durch eine Funktion $\lambda^\xi : \mathcal{K}^\xi \times \mathcal{N} \mapsto \mathcal{K}^\xi \cup \{\bot\}$ und hat vor der i-ten Nachricht den Zustand $k_i^\xi \in \mathcal{K}^\xi \subseteq \mathcal{K}$. Wobei Ξ, $\mathcal{N}$ und $\mathcal{K}$ entsprechend die Mengen aller Kontexte, Nachrichten und Kontextzustände sind. Der Wert $\bot$ als Rückgabe bedeutet, dass die Nachricht nicht mit dem Kontext *kompatibel* ist. Sonst wird ein neuer Zustand des Kontexts zurückgegeben. Dieser wird nur verwendet, wenn der Kontext *getroffen*, also von der Verhaltensbeschreibung erwartet wurde. Die Menge der kompatiblen Kontexte in Schritt i wird kurz mit Ξ_i^λ angegeben (4.7). Der ξ zugeordnete Zustand k_{i+1}^ξ nach der i-ten Nachricht n_i ergibt sich somit induktiv aus dem vorherigen Zustand k_i^ξ, dem Resultat von $\lambda^\xi(k_i^\xi, n_i)$ und der Treffer-Abfrage des Kontexts in der Verhaltensbeschreibung (4.8). Letztere wird mit der Funktion $\Lambda : \mathbb{S} \times \mathbb{E} \times \mathcal{P}(\Xi) \mapsto \mathcal{P}(\Xi)$ beschrieben, die zurückgibt, welche der kompatiblen Kontexte im Zustand $q_i \in \mathbb{S}$ von dem Ereignis $a_i = \Psi(n_i)$ getroffen werden. Der Vollständigkeit halber sei erwähnt, dass dies prinzipiell auch bereits mit der Kompatibilität entschieden werden kann. Diese Erweiterung von λ^ξ, spezifisch für einen Automaten A, wird bezeichnet als λ_A^ξ. Ein Kontext ist also ein Automat $\xi = \left\langle \mathcal{N}, \mathcal{K}^\xi, k_0^\xi, \lambda_A^\xi \right\rangle$. Durch die Aufspaltung entfällt allerdings die Notwendigkeit Teile von A nachzubauen.

$$\Xi_i^\lambda = \{\xi \in \Xi \mid \lambda^\xi(k_i^\xi, n_i) \neq \bot\} \tag{4.7}$$

$$k_{i+1}^\xi = \begin{cases} \lambda^\xi(k_i, n_i), & \text{if } \xi \in \Xi_i^\lambda \cap \Lambda(q_i, \Psi(n_i), \Xi_i^\lambda) \\ k_i^\xi, & \text{sonst} \end{cases} \tag{4.8}$$

Zur Integration in einen Automaten $A = \langle \mathbb{B}, \mathbb{S}, s_0, \delta \rangle$ werden die Beschriftungen erweitert zu $\mathbb{B} \subseteq \mathbb{E} \times \mathcal{P}(\Xi)$. Sei $ct = \{\xi_1, \xi_2, \ldots\}$ eine Menge von Kontexten aus Ξ, dann ist $\langle e, ct \rangle$ die Beschriftung einer Transition, die das Ereignis e in den Kontexten ct erwartet. Dabei müssen alle angegebenen Kontexte kompatibel sein und genau diese werden dann getroffen, d. h. die Rückgabe von Λ wird durch die Beschriftung der ausgeführten Transition bestimmt (4.9).

Eine Konfiguration $K = \langle as, cs \rangle$ enthält neben der Menge der aktiven Zustände $as \subseteq \mathbb{S}$ auch die Kontextzustände $cs \subseteq \xi \mapsto \mathcal{K}$.

$$\Lambda(q_i, a_i, \Xi_i^\lambda) = \{\xi \in ct \subseteq \Xi_i^\lambda \mid \langle q_i, \langle a_i, ct \rangle \rangle \in \mathrm{dom}(\delta)\} \tag{4.9}$$

Rein deklarativ kann leicht definiert werden, dass für jede Nachricht die Menge aller möglichen kompatiblen Kontexte bestimmt wird; ebenso deren Kompatibilität mit allen folgenden Nachrichten. Wie von Allan u. a. [All+05] bereits bei der Verwendung freier Variablen für das Schneiden von Traces erkannt wurde, gibt dies wenig Hilfestellung bei der praktischen Umsetzung. Es gäbe einfach zu viele mögliche Kontexte, die gleichzeitig verfolgt werden müssten, aber nie getroffen würden. Dies ist allerdings auch nicht notwendig, da nur die Kontexte betrachtet werden müssen, die zur Verhaltensüberwachung notwendig sind. Für einen beliebigen Zustand sind das die von seinen ausgehenden Transitionen erwarteten Kontexte. Zur Abbildung von Kontexten auf einen einfachen Automaten müssten die Zustände und Transitionen für jede mögliche Kombination der Kontextzustände kopiert werden. Sofern es keine Begrenzung dieser Kombinationen gibt, kann dies im Allgemeinen nicht mehr mit einem *endlichen* Automaten beschrieben werden.

Die Zuordnung von Sender und Empfänger kann durch Definition von Kontexten für die Rollen jeder Komponenten und Verwendung an den Transitionen erfolgen. Allgemein bieten Kontexte ein generisches Modell zum Umgang mit Daten der Interaktionen. Solange eine Implementierung auf Kontexte abgebildet werden kann, schränkt das nicht die Wiederverwendung existierender Mechanismen bei der Verarbeitung ein; beispielsweise für die Zuordnung einer Anfrage zur passenden Antwort, da viele Middleware-Bibliotheken diese Zuordnung bereits erledigen oder leicht ermöglichen. Entsprechend können dann die Parameter der Anfrage quasi in die Antwort kopiert und zusätzlich in der Zuordnung eines Ereignisses verwendet werden. Dies erspart eine explizite Konfiguration von Kontexten für diese Standardmechanismen. Damit eine weitergehende Referenz auf dieses Verhältnis in der Verhaltensbeschreibung nicht eingeschränkt wird, sollte eine solche Sonderbehandlung implizit passende Kontexte deklarieren.

4.6.3 Instanzen

Bestimmte Sequenzen des erwarteten Verhaltens innerhalb der Spezifikation können sich wiederholen. Beispielsweise wird zunächst der Aufbau einer Verbindung angefragt, diese dann aufgebaut, und schließlich abgebaut, nachdem sie freigegeben wurde. Diese Sequenz wird für jede weitere Verbindung wiederholt. Sind die Wiederholungen sequentiell, können die Zustände und Transitionen einen zyklischen Graphen bilden. Für jede Verbindung, die gleichzeitig verwaltet wird, muss das Kreuzprodukt dieser Zustände und Transitionen mit

sich selbst gebildet werden, um alle Möglichkeiten darzustellen. In einem hierarchischen Automaten kann dies auch durch identische, parallele Automaten abgebildet werden, von denen jeder eine Verbindung verfolgt. Kontexte können verwendet werden, um die Nachrichten dem richtigen Zustand bzw. Automaten zuzuordnen, beispielsweise durch Unterscheidung der Verbindungen mit einer Kennzahl.

Zur Verallgemeinerung dieses Konzeptes wird es ermöglicht, einen Automaten beliebig häufig zu *instanziieren*. Die *Instanzen* laufen dann parallel zueinander und verfeinern einen vorher bestimmten Automaten. Wird dies realisiert, indem jede Instanz eine eigenständige Kopie des instanziierten Automaten ist, entspricht dies den manuell modellierten, parallelen Automaten. Indem eine Instanz einen Instanz-spezifischen Kontextzustand für einen Kontext erstellt, kann sie diesen Kontext instanziieren, um beispielsweise die Merkmale einer Verbindung zu speichern. Ereignisse können dann dem individuellen Kontext und damit der Instanz zugeordnet werden. Sofern keine Obergrenze für die maximale Anzahl an Instanzen festgelegt wird, kann der enthaltende Automat potenziell unendlich viele Zustände haben. Voraussetzung ist aber die Verwendung von unendlich vielen, verschiedenen Kontexten. Zwei Instanzen mit äquivalenten Kontexten sind identisch, sobald sie den gleichen aktiven Zustand haben und es muss nur noch eine der Instanzen betrachtet werden. Da die Anzahl der Zustände für das Verhalten begrenzt ist, kann die Differenzierung nur über den Zustand des Kontextes erfolgen. Ist auch die Anzahl der Kontextzustände beschränkt, kann dies auch mit endlich vielen Zuständen als einfacher Automat abgebildet werden.

Für die Formalisierung wird zunächst ein einfacher *Automat mit Instanzen* betrachtet, ohne Kontexte: Er ist definiert durch ein 3-Tupel $AI = \langle \mathbb{A}, A_0, \mathbb{E} \rangle$. $\mathbb{A}$ ist eine Menge sequentieller Automaten, $A_0 \in \mathbb{A}$ ist der initiale Automat und $\mathbb{E}$ ist die Menge der Ereignisse. Die Automaten haben disjunkte Zustandsräume. Die Beschriftung in den Automaten sind Tupel der Form $\langle e, \mathbb{A}_{new} \rangle \in \mathbb{B} = \mathbb{E} \times \mathcal{P}(\mathbb{A})$. Das erste Element e beschreibt das Ereignis, durch das die Transition aktiviert wird. Das zweite Element $\mathbb{A}_{new}$ ist Menge der Automaten, von denen neue Instanzen erzeugt werden. Dazu werden die initialen Zustände in die Konfiguration $K \subseteq \mathbb{S}$, das ist die Menge der aktiven Zustände, aufgenommen.

Auch hier gilt wieder: Durch Abbildung jeder erreichbaren Konfiguration auf jeweils einen Zustand und Verbindung dieser mit den entsprechenden Transitionen, kann dies auf einen einfachen Automaten abgebildet werden. Sofern alle $A \in \mathbb{A}$ eine endliche Anzahl an Zuständen haben, hat auch der einfache Automat endlich viele Zustände. Da es sich bei einer Konfiguration immer um eine Untermenge aller Zustände handelt, ist die Menge aller Konfigurationen eine Untermenge der Potenzmenge aller Zustände und somit durch $2^{|\mathcal{P}(\mathbb{S})|}$ begrenzt.

Die vorgestellten Konzepte, wie Kontexte und Instanzen wurden auf sequentieller Automaten abgebildet und sind als Erweiterungen solcher beschrieben. Eine Kombination der Konzepte ist damit prinzipiell bereits möglich und wird im Folgenden für Instanzen und Kontexte demonstriert. Sie kann offensichtlich ebenso auf einen einfachen Automaten abgebildet werden kann.

Ein *Automat mit Kontexten und Instanzen* ist ein 4-Tupel $KI = \langle \mathbb{A}, A_0, \mathbb{E}, \Xi \rangle$. $\mathbb{A}$ ist eine Menge sequentieller Automaten mit Kontexten, $A_0 \in \mathbb{A}$ ist der initiale Automat, $\mathbb{E}$ ist die Menge der Ereignisse und Ξ ist die Menge der Kontexte. Die Automaten haben disjunkte Zustandsräume. Es sei $ct = \{\xi_1, \xi_2, \ldots\}$ eine Menge von Kontexten aus Ξ. Die Beschriftung der Automaten sind Tupel $\langle e, ct, \mathbb{A}_{new} \rangle \in \mathbb{B} = \mathbb{E} \times \mathcal{P}(\Xi) \times \mathcal{P}(\mathbb{A})$, damit bei Ausführung der Transition für e in den Kontexten ct neue Instanzen der Automaten $\mathbb{A}_{new}$ erzeugt werden. Eine Konfiguration $K = \langle as, cs \rangle$ enthält neben der Menge der aktiven Zustände $as \subseteq \mathbb{S}$ auch die Kontextzustände $cs \subseteq \xi \mapsto \mathcal{K}$.

Wie bereits dargestellt wurde, kann es hilfreich sein, dass zwei Instanzen eines Automaten unterschiedliche Kontextzustände verwenden. Dies kann mit der beschriebenen Formalisierung realisiert werden, indem ein Kontext, der instanziierte Kontextzustände verwendet, alle Instanzen eindeutig durchnummeriert. Sein Kontextzustand ist dann eine Sequenz der instanziierten Kontextzustände sortiert entsprechend der Nummerierung. Die so identifizierbaren Kontextzustände können dann wie in Abschnitt 4.6.2 angegeben ausgewertet und aktualisiert werden. Wie bereits dort für einfache Kontexte beschrieben, kann dies auch in einem Automaten beschrieben werden. Dazu wird allerdings eine (ggf. vereinfachte) Kopie des Automaten benötigt, um zu erkennen, welche Instanz getroffen wurde. Da es sich hierbei mehr um eine Optimierung der Implementierung handelt und dies analog zu den Instanzen von Automaten funktioniert, wird im Rahmen dieser Arbeit auf eine genauere Beschreibung der Formalisierung von Kontextinstanzen verzichtet. Der interessierte Leser sei hier auf die Formalisierungen zum Schneiden von Traces mit Parametern [All+05] und von selbst-replizierenden Automaten [Nas16] (engl.: *self-replicating Automata*) hingewiesen, in denen freie Variablen zu einer mehrfachen Auswertung eines Automaten führen.

4.6.4 Zeitverhalten

Die Verhaltensbeschreibung soll auch einen Bezug zu Zeit und damit verbundenen Eigenschaften und Beschränkungen ermöglichen. Die logische Zeit, also in welcher Reihenfolge Nachrichten erwartet werden, wird bereits über die Pfade im Automaten angegeben. Unabhängige Sequenzen können mit parallelen Automaten eines hierarchischen Automaten beschrieben werden. Die physische Zeit nimmt hingegen Bezug darauf, wie viel Zeit auf einer (realen) Uhr verstreicht, bis etwas geschieht.

Ein verbreiteter Ansatz zur Beschreibung von Zeit in Automaten sind sog. Zeit-behaftete Automaten [AD94; Bak+18; WA19]. Sie besitzen eine Menge an Uhren auf denen Bedingungen für die Transitionen formuliert werden können. Eine ausgeführte Transition setzt eine bestimmte Untermenge der Uhren zurück. Dies bietet eine solide Grundlage, um Zeitverhalten beschreiben und überprüfen zu können. Die Zeit wird in einer Nachricht typischerweise als Zeitstempel in den Meta-Daten hinterlegt. Zeit ist damit ein weiteres Datum, das wie andere Daten verarbeitet werden kann. Entsprechend kann ein Zeit-behafteter Automat mit seinen Bedingungen auch mit den bereits beschriebenen Kontexten ausgedrückt werden.

Zeit-behaftete Automaten sind ein Ansatz auf niedriger Ebene. Bereits Alur und Dill [AD94] haben das festgestellt und schlagen vor, dass Abbildungen aus Sprachen mit mehr Struktur sinnvoll wären. Ein Ansatz dazu ist die Zeitverhalten-erweiterte Beschreibungssprache (TADL, engl.: *Timing Augmented Description Language*) [TIM09]. Entsprechend widmet sich dieser Abschnitt mehr der praktischen Verwendbarkeit und untersucht dazu, wie die Zeit-Bedingungen der TADL in der vorgestellten Modellierung ausgedrückt werden können. Da bereits Syntax und Semantik der Zeit-Bedingungen vorgegeben sind, steht die Integration in die Modellierung im Vordergrund. Die TADL erwartet, dass die Modellierung beobachtbare Ereignisse definiert, auf denen die Zeit-Bedingungen formuliert werden. Solch ein Ereignis muss die komplette Situation erfassen, die für die Zeit-Bedingung relevant ist. Dabei reicht es nicht, nur auf die Ereignisse, wie sie in Abschnitt 4.5 beschrieben sind, Bezug zu nehmen, sondern auch der Zustand der Verhaltensbeschreibung kann relevant sein; ebenfalls getroffene Kontexte.

Aber zunächst, bevor der Integration der Zeit-Bedingungen nachgegangen wird, noch ein kurzer Blick in die TADL, um herauszufinden, welche Bedingungen dort formuliert werden. Sie bietet zwei Grundmuster, für die sie jeweils einen Algorithmus zur Auswertung angibt: Zum einen gibt es Bedingungen der Wiederholungsrate (engl.: *repetition rate constraints*) und zum anderen Bedingungen für die Verzögerungen (engl.: *delay constraints*). Alle weiteren Varianten der betrachteten Bedingungen können durch bestimmte Parameter mit den Grundmustern nachgebildet und damit durch die Algorithmen bewertet werden.

Die Parameter der Wiederholungsrate bestehen aus einer Unter- sowie Obergrenze für den Abstand zwischen zwei Ereignissen, der Schrittweite und einem sog. Jitter, das ist die erlaubte Abweichung von einem idealen Zeitpunkt für jedes Auftreten. Die Schrittweite gibt an, das wievielte folgende Ereignis betrachtet wird. Eine Schrittweite von 1 bezeichnet direkt aufeinanderfolgende Ereignisse. Der Algorithmus zur Auswertung betrachtet die Sequenz der Bereiche, in denen die idealen Zeitpunkte liegen können, die im Allgemeinen mit zunehmenden Beobachtungen kleiner werden.

Die Parameter der Verzögerung bestehen aus einer Ereigniskette und der Breite, Unter- sowie Obergrenze für das Zeitfenster. Die Ereigniskette beinhaltet einen Stimulus und eine Reaktion. Sie kann durch weitere Ereignisketten verfeinert werden, die sequentiell oder parallel angeordnet sein können. Die Ereigniskette dient dazu, Stimuli mit den richtigen Reaktionen zu verknüpfen, die dann in dem Zeitfenster liegen müssen. Für die Auswertung wird überprüft, ob die relevanten Reaktionen für jeden Stimulus im Zeitfenster liegen. Der genaue Algorithmus ist, wie der für die Wiederholungsrate, in dem Projektbericht von TIMMO [TIM09] beschrieben und wird hier nicht wiederholt.

Kontexte können die notwendigen Daten erfassen und darauf die Bedingungen formulieren. Insbesondere für die Wiederholungsrate bietet sich an, dies über die Kompatibilitätsfunktion λ eines dedizierten Kontext auszuwerten. Ein Ereignis ist nur dann mit dem Kontext kompatibel, wenn er zu dem erwarteten Bereich passt, in diesem Fall dem Zeitfenster. Wird der Kontext getroffen, werden die Bereichsgrenzen (für das nächste Ereignis) entsprechend aktualisiert. Damit kann beispielsweise auch modelliert werden, dass eine Wiederholung nur in bestimmten Systemzuständen erwartet wird. Die Struktur der Ereignisketten erinnert an den Aufbau eines hierarchischen Automaten. Deshalb kann eine Verzögerung auch mit der Zeit in Verbindung gebracht werden, die ein Zustand aktiv ist. Da dies häufiger benötigt wird, werden Zustände mit einer Zeitschranke für die maximale Aktivierungsdauer versehen und nach deren Ablauf wird ein Ereignis ausgelöst.

Eine Bedingung für die Verzögerung besteht dann im Allgemeinen aus zwei Zuständen. Der erste Zustand wird durch den Stimulus betreten, der die Untergrenze als Zeitschranke hat. Durch das ausgelöste Ereignis wechselt er in den zweiten Zustand, der die Reaktion erwartet. Dieser hat als maximale Aktivierungsdauer die Differenz aus Unter- und Obergrenze. Komplexere Ereignisketten können durch Verfeinerung des ersten Zustands realisiert werden. Insbesondere bei Verwendung von diskreten Zeitschritten sollte noch unterschieden werden, ob die Aktivierungsdauer zu Beginn oder am Ende des Zeitschritts erfolgt. Denn würde man sich für eine Variante entscheiden und die Zeitschwelle um $\pm\epsilon$ anpassen, um die andere Priorisierung auszudrücken, hätte dies Auswirkungen auf die Startzeit der Aktivierungsdauer des nächsten Zustands. Durch eine Wahlmöglichkeit je Zustand wird dies gelöst.

Durch die Integration von TADL in die Modellierung ist eine strukturierte Beschreibung von Zeit-Bedingungen möglich. Es wurde nicht untersucht, ob TADL und Zeit-behaftete Automaten die gleiche Mächtigkeit haben. Sofern notwendig, ist es aber möglich, letztere mit Kontexten zu emulieren. Da die Verwendbarkeit der Modellierung zu einer statischen Verifizierung der Spezifikation nicht im Fokus der Arbeit liegt, wurde auf eine Abbildung der gewählten Beschreibung von Zeitverhalten auf Zeit-behaftete Automaten verzichtet.

4.7 Fazit

Dieses Kapitel hat ein neues Referenz-Modell zur Beschreibung des Soll-Verhaltens eines verteilten eingebetteten Systems vorgestellt, das als Grundlage für die Überprüfung dieses Systems dienen kann. Hierbei handelt es sich um eine Kombination, Anpassung und Erweiterung bestehender Ansätze, um eine Technologie-unabhängige Beschreibung (siehe Herausforderung 1), insbesondere auch der Interaktionen an Schnittstellen, zu ermöglichen. Die Beschränkung auf präzise definierte Elemente resultiert in einer klaren Semantik für die Beschreibung (siehe Herausforderung 2). Der Ansatz lindert durch Kontexte und Instanzen eine drohende Zustandsexplosion (siehe Herausforderung 3) und ermöglicht die Beschreibung nicht-funktionaler Eigenschaften wie Zeit (siehe Herausforderung 4). Er begegnet damit den in Ziel 1 beschriebenen Herausforderungen.

Die Aufteilung des Modells in Schichten gibt eine mögliche Antwort auf die Frage, wie Soll-Verhalten wiederverwendbar beschrieben werden kann. Zum einen können Artefakte einzelner Schichten direkt wiederverwendet werden. Zum anderen schafft die Aufteilung den Raum, die Semantik einzelner Schichten anzupassen, beispielsweise können Ereignisse mit oder ohne Kontexte ausgewertet werden.

Das Modell stellt dabei die Schnittstellen des Systems in den Vordergrund, da ein Kommunikationskanal häufig einfacher überwacht werden kann, als die Verarbeitung innerhalb einer Komponente. Zudem ist so kein detaillierter Einblick in einzelne Komponenten notwendig, aber bei Bedarf auch möglich. Die Modellierung beschreibt die Topologie des Systems, die verfügbaren Nachrichten von Schnittstellen, sowie Ereignisse basierend auf deren Parametern und definiert mit diesen Bausteinen in der Verhaltensbeschreibung gültige Sequenzen. Durch die eingeführten Erweiterungen für Instanzen und Kontexte können neben den sequentiellen Abhängigkeiten des Kontrollflusses auch die Zusammenhänge von Daten der Nachrichten kompakt beschrieben werden. Instanziierte Zustände kapseln weitgehend unabhängige, identische Abläufe und erleichtern so, den einzelnen Ablauf zu erkennen. Kontexte bieten das Potenzial nahezu beliebige Auswertungen in die Überprüfung einzubauen, insbesondere solche, die besser in einer anderen Form als mit einem Automaten abgebildet werden. Bedingungen an das Zeit-Verhalten des Systems können ebenfalls mit Kontexten und über die maximale Aktivierungsdauer jedes Zustands beschrieben werden. Dies ist in vielen Fällen einfacher und kompakter, als beispielsweise mit rücksetzbaren Uhren, wie sie in Zeit-behaftete Automaten verwendet werden.

Eine Beschreibung mit positivem Soll-Verhalten vereinfacht zudem die Generierung von weiteren Artefakten, beispielsweise für eine Restbus Simulation oder um die Abdeckung einer Test-Sammlung zu prüfen.

Da die Erweiterungen auf sequentielle Automaten zurückgeführt wurden, ist eine Überprüfung mit bekannten Verfahren zur Laufzeitüberprüfung mit Automaten möglich. Wie alle erkennbaren Abweichungen gefunden werden können, auch wenn nur das Soll-Verhalten beschrieben wurde, wird im nächsten Kapitel untersucht. Einzelne, bekannte Abweichungen können in die Modellierung mit aufgenommen werden. Im Rahmen dieser Arbeit wird die Überprüfung bekannter Abweichungen aber nicht genauer betrachtet und wie Soll-Verhalten behandelt. Dies ist damit begründet, dass während der Überprüfung für solche Abweichungen lediglich ein anderes Verdikt ausgegeben wird, sie sonst aber wie Soll-Verhalten behandelt werden können.

Kapitel 5

5 Resumption - Wiederaufnahme der Überprüfung

Dieses Kapitel stellt die neue, vom Autor erarbeitete Methodik zur automatisierten Erweiterung von Monitoren um *Resumption* vor. Dazu wird untersucht, ob und wie ein Monitor auf Basis der Spezifikation des beobachteten Systems erkennen kann, wann das System sich nicht entsprechend der Spezifikation verhält. Ein robustes System wird, solange es sich um keinen schwerwiegenden Fehler handelt, auch nach einer Abweichung fortfahren. Der Monitor sollte diese Abweichung melden, ebenso wie alle weiteren, die danach auftreten. Die erarbeitete Methodik zur Erweiterung mit Resumption soll einem Monitor ermöglichen, allein auf Grundlage der Beschreibung des gewünschten Soll-Verhaltens, alle erkennbaren Abweichungen so genau wie möglich zu melden.

Das Erstellen einer vollständigen Beschreibung des Soll-Verhaltens, insbesondere im Rahmen der Spezifikation, ist häufig nicht wirtschaftlich. Die Methodik soll daher *gutmütig* mit unvollständigen Beschreibungen umgehen. Das bedeutet im Detail, dass korrektes und beschriebenes Verhalten immer akzeptiert werden sollte, unabhängig davon, ob auch anderes Verhalten beschrieben wurde oder nicht. Im Falle von nicht beschriebenem Verhalten sollte dies wie unerwartetes Verhalten behandelt und gemeldet werden – in der vorgestellten Modellierung wird unerwartetes Verhalten auch nicht explizit beschrieben (siehe Kapitel 4). Der Monitor sollte möglichst schnell seine Überprüfung fortsetzen, also sobald das Verhalten wieder im Monitor hinterlegt ist, um weitere Abweichungen melden zu können. Wie und mit welchen Annahmen erkannt werden kann, ob das beobachtete Verhalten wieder beschrieben ist, und welche Auswirkungen dies auf die Erkennung von Abweichungen hat, wird in diesem Kapitel untersucht.

Dieses Kapitel baut die Untersuchung von Resumption mit den folgenden Schritten auf: Zunächst wird die Problemstellung theoretisch betrachtet. Um eine anschauliche Abstraktion der Problematik zu gewähren, wird die Verifikation zur Laufzeit auf Basis einer Soll-Beschreibung als Spiel zwischen zwei Spielern beschrieben. Anhand dieser Abstraktion wird

untersucht, wie der Monitor unerwartetes Verhalten effizient erkennen und wie er den Zustand des Systems, in Bezug auf die Beschreibung, bestimmen kann. Da die Abweichungen und nicht das unerwartete Verhalten vom Monitor erkannt werden sollen, wird anschließend betrachtet, wie der Monitor bei Beobachtung von unerwartetem Verhalten die erkennbaren Abweichungen des Systems eingrenzen kann. Dabei wird erschlossen, welche Abweichungen erkennbar sind. Auf Basis der theoretischen Erkenntnisse wird dann die Methodik zur Erweiterung eines Monitors mit Resumption formal beschrieben. Die Grundlagen der Methodik für einfache Automaten wurden bereits in [DWB18] veröffentlicht. Die dort beschriebene Theorie ist auf Resumption für sequentielle Automaten beschränkt.

Die Modellierung beschreibt Verhalten mit hierarchischen Automaten, die Kontexte, Instanzen und Zeitschranken verwenden (vgl. Kapitel 4). Zusätzlich wird in diesem Kapitel daher betrachtet, wie Resumption auf die vorgestellten Modellierungskonzepte angewendet wird. Wie dort bereits gezeigt wurde, lassen sich diese Modellierungskonzepte auf sequentielle Automaten abbilden. Resumption kann damit auf das Resultat dieser Abbildungen angewendet werden. Eine naive Anwendung ist aber häufig ineffizient, da die Abbildungen zu einer Explosion des Zustandsraums führen. Daher wird der Einfluss der verschiedenen Modellierungskonzepte auf Resumption untersucht.

5.1 Resumption für einfache Automaten

Ein Referenz-Modell, wie es in Kapitel 4 vorgestellt wurde, definiert, wie sich ein System unter Beobachtung (SuO, engl.: *System under Observation*) verhalten soll. Da dort das erwartete Verhalten definiert wird, entspricht jede Transition im Zustandsautomaten des Modells einer erwarteten Beobachtung. Systeme können sich in bestimmten Situationen aber anderes verhalten, als durch das Referenz-Modell erwartet werden würde. Dies ist auch die Motivation dafür, sie zur Laufzeit zu beobachten. Folglich ist in der Soll-Beschreibung nicht definiert, wie das System nach einer Abweichung weiterarbeiten soll. Der Zustand des SuO wird undefiniert. Wenn die Abweichung nicht kritisch war und das SuO robust implementiert wurde, wird es aber oft in der Lage sein, korrekt weiterzuarbeiten, also nach der Abweichung wieder erwartetes Verhalten zeigen.

Betritt ein Monitor nach einer erkannten Abweichung einen finalen Zustand, wie das Beispiel in Abbildung 5.1 zeigt, kann er nur die erste Abweichung finden. In der Abbildung wird '*' als Kurzschreibweise für alle im Ausgangszustand nicht durch andere Transitionen gebundenen Ereignisse verwendet. Im Falle eines Referenz-Modells sind genau die unerwarteten Ereignisse ungebunden. Der erweiterte Automat betritt den ablehnenden Zustand $s_\perp$ also immer nach einem Verstoß gegen die Referenz. Ein Monitor, der einen solchen Automaten verwendet, kann ungültige Läufe identifizieren, nicht aber alle individuellen

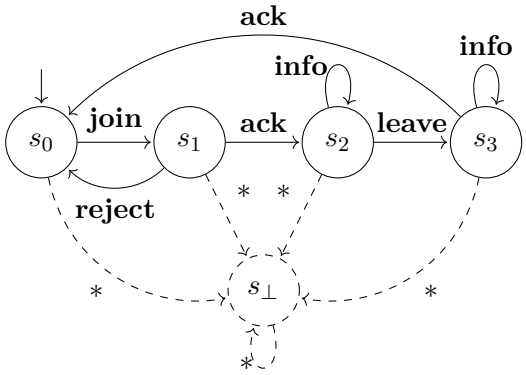


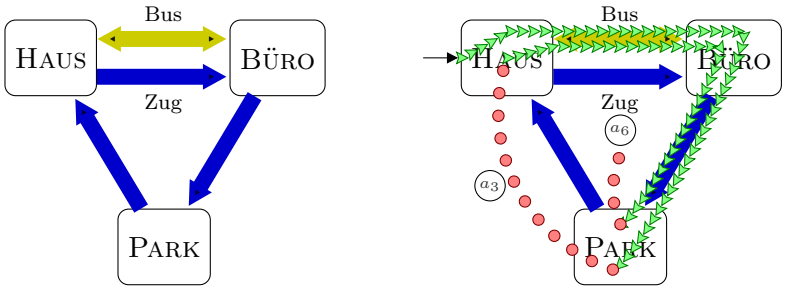
Abb. 5.1: Erweiterung des Automaten aus Abbildung 2.1 zur Erkennung ungültiger Traces.

Abweichungen, sollte es mehr als eine geben. Verschiedene Techniken können auf einen Monitor angewendet werden, damit er mehr als nur die erste Abweichung finden kann. Bisher wird dies in der Regel von Hand durchgeführt und erfordert zusätzliche Designarbeit, z. B. um Transitionen hinzuzufügen oder um die Spezifikation nur zum Zweck der Überprüfung in verschiedene Teile aufzuteilen, die separat und wiederholt überprüft werden können. Um eine automatisierte Lösungsstrategie zur Erweiterung von Monitoren zu entwickeln, die es ihnen ermöglicht, mehr als nur die erste Abweichung zu erkennen, erarbeitet dieser Abschnitt eine abstrakte Sichtweise auf die Problematik.

5.1.1 Verifikation zur Laufzeit als Spiel

Zur Veranschaulichung, welche Informationen dem Monitor verfügbar sind, kann die Laufzeitüberprüfung mit einem Referenz-Modell als Spiel formuliert werden. Im Folgenden spielt Sven die Rolle des SuO und bewegt sich verdeckt auf einem Spielplan, aber hinterlässt dabei eine Spur seiner Bewegungen. Er kann in einem Spielzug auch eine Bewegung ansagen, die auf dem Spielplan eigentlich nicht möglich ist und sich dann an einen beliebigen anderen Ort begeben. Monika, sie spielt den Monitor, muss nach jeder Bewegung bekannt geben, ob Sven die Vorgaben des Spielplans einhält oder dagegen verstößt. Sie kennt lediglich Spur und Spielplan, seinen Aufenthaltsort teilt Sven nicht explizit mit.

Abbildung 5.2a zeigt ein Beispiel für einen Spielplan. Die Felder, dargestellt durch abgerundete Rechtecke, sind Orte, an denen sich Sven aufhalten kann. Ein anderer Aufenthaltsort ist nicht möglich. Die Orte sind durch Verkehrsmittel verbunden. Diese sind durch unterscheidbare Pfeile gekennzeichnet und können nur in deren Richtung genutzt werden, um den Ort zu wechseln. Eine Spur entspricht einer Sequenz aus Verkehrsmitteln in der Reihenfolge, die zum Ortswechsel verwendet wurde. Da der Spielplan auch als Automat betrachtet werden kann, wird die in Abschnitt 2.2.2 vorgestellte Notation für formale Beschreibungen verwendet. Dabei werden Verkehrsmittel mit e_i und Orte auf dem Plan



(a) Beispiel eines Spielplans, der Sven den Wechsel zwischen Orten in Pfeilrichtung erlaubt.

(b) Verdeckte Bewegung von Sven. Dreiecke zeigen konforme, Kreise abweichende Spielzüge.

Abb. 5.2: Beispiel eines Spiels der Verifikation zur Laufzeit, in dem Monika (Monitor) zu erkennen versucht, ob Sven (SuO) sich an die Vorgaben des Spielplans (Spezifikation) hält.

mit s_i bezeichnet. Das Verkehrsmittel in Schritt j ist a_j, ausgehend von Ort q_j. Die Folge der Schritte ergibt eine Spur, allgemein auch als *Trace* bezeichnet.

Abbildung 5.2b stellt den Pfad dar, den Sven in diesem Beispiel gewählt hat. Nur er kennt diesen Pfad. Er hinterlässt dabei die für Monika sichtbare Spur *Bus,Zug,Bus,Bus,Zug,Bus*. Er beginnt im Haus. In den Schritten a_3 und a_6, entschließt er sich von dem Spielplan *abzuweichen* und behauptet den *Bus* zu nehmen, der im Park eigentlich nicht vorhanden ist. Bei a_3 entscheidet er sich dabei zu Haus zu gehen; bei a_6 kann er noch zu einem beliebigen Ort springen. Alle anderen Bewegungen sind mit dem Spielplan *konform*.

Mit Wissen über den aktuellen Aufenthaltsort von Sven ist es für Monika trivial, konformen Bewegungen anhand von Spielplan und Spur zu folgen. Sie kann seinen nächsten Ort ableiten, in dem sie das Ziel des von Sven bekannt gegebenen Verkehrsmittels für seinen bisherigen Aufenthaltsort nachschlägt. Diese Abbildung von Ort und Verkehrsmittel zu einem Zielort entspricht der Transitionsrelation des zugrundeliegenden Automaten und wird deshalb kurz mit δ bezeichnet. Die erste Abweichung kann von ihr leicht identifiziert werden, denn dann ist der Spielzug von Sven nicht in δ enthalten. Dies zeigt beispielhaft, wie Monitore ohne Resumption funktionieren.

Theorem 1. *Monika kann Abweichungen anhand der Transitionsrelation identifizieren* (5.1)*, wenn sie den Aufenthaltsort q_i von Sven kennt.*

$$\gamma(q_i, a_i) \mapsto \begin{cases} \top, & \textit{if } \langle q_i, a_i \rangle \in \mathrm{dom}(\delta) \\ \bot, & \textit{otherwise.} \end{cases} \tag{5.1}$$

Beweis. Wenn Monika bekannt ist, dass Sven sich an Ort q_i befindet und er das Verkehrsmittel a_i benutzt, kann sie dies in δ nachschlagen. Ist a_i am Ort q_i verfügbar, d. h.

$\langle q_i, a_i \rangle \in \text{dom}(\delta)$, dann kann dort SVEN nicht mit a_i abweichen und MONIKA kennt den neuen Aufenthaltsort von SVEN: $\delta(q_i, a_i)$. Ist a_i nicht am Ort q_i verfügbar, muss er Abweichen, wenn er behauptet das Verkehrsmittel zu verwenden. Deshalb kann MONIKA immer bestimmen, ob SVEN abweicht, wenn sie seinen Aufenthaltsort kennt. □

Nach einer Abweichung kann MONIKA allerdings nicht mehr wissen, an welchen Ort sich SVEN bewegt hat. Sie ist sich seines Aufenthaltsorts ungewiss. Sich dieser Ungewissheit bewusst zu werden und sie zu reduzieren, um die Überprüfung wiederaufzunehmen, wird als *Resumption* bezeichnet. Um die Bedeutung von mehreren Abweichungen in einer Sequenz von Verdikten zu definieren, bezieht sich jedes Verdikt auf den Teil der Spur nach der letzten gemeldeten Abweichung. Die folgenden Abschnitte untersuchen das Potenzial von Resumption bei der Erkennung von unerwartetem Verhalten und Abweichungen. Dazu wird zunächst bewiesen, dass alles unerwartete Verhalten mit Resumption erkannt werden kann. Im Anschluss wird demonstriert, wie mit Resumption minimale Teilsequenzen der Spur bestimmt werden können, die zusammen alle und jede für sich exakt eine erkennbare Abweichung enthalten.

5.1.2 Resumption zur Erkennung von unerwartetem Verhalten

Unerwartetes Verhalten ist definiert als jede Sequenz von Ereignissen, die nicht in der Spezifikation bzw. dem Referenz-Modell enthalten ist. Es wird angenommen, dass die interne Implementierung des Verhaltens einer Komponente des SuO unbekannt ist und nur deren Stimuli und Antworten beobachtet werden können (Black-Box-Annahme). Es wird erwartet, dass diese Interaktionen innerhalb des SuO durch einen Automaten $A = \langle \mathbb{E}, \mathbb{S}, s_0, \delta \rangle$ beschrieben werden können. Wenn beobachtete Ereignisse zu einem Pfad durch A passen, wird dieses Verhalten als konform angenommen. Ansonsten, also im Fall eines Verstoßes, ist der neue Zustand des SuO unbekannt, da die Beobachtung die einzige Informationsquelle ist. Im Allgemeinen kann keine Einschränkung für den neuen Zustand gegeben werden. Diese Transition kann also beispielsweise auch nicht-deterministisch sein. Ein Monitor $\mathrm{M} = \langle \mathbb{E}, \mathbb{S}, s_0, \delta, \mathbb{D}, \gamma \rangle$ kennt nur die Spezifikation sowie die Ereignisse aus der Beobachtung und bewertet damit das Verhalten des SuO.

Formaler und auf das vorgestellte Spiel angewendet wird unerwartetes Verhalten definiert als eine Spur, die ein Ereignis a_j mit $\langle q_j, a_j \rangle \notin \text{dom}(\delta)$ enthält. SVEN befindet sich vor Spielzug j an Ort q_j. Niemand außer SVEN kennt q_j, weswegen MONIKA stattdessen die Menge an Orten Q_j^k berücksichtigt, die mit erwartetem Verhalten von einem beliebigen Ort vor Spielzug k bis dahin erreicht werden könnten (5.2). Sind k und j identisch, dann enthält diese Menge alle Orte ($Q_j^j = \mathbb{S}$).

$$Q_j^k = \delta(\mathbb{S}, a_k \dots a_{j-1}) \tag{5.2}$$

Theorem 2. *Eine Spur kann inkrementell aufgeteilt werden, sodass jedes (abgeschlossene) Segment genau ein unerwartetes Verhalten enthält. Je (weiterem) Spielzug kann innerhalb von $O_{Platz}(|A|)$ und $O_{Zeit}(|\mathbb{S}|)$ entschieden werden, ob der nächste Spielzug zu dem gleichen Segment gehört oder zu einem neuen.*

Beweis. Zum Beweis wird die Konstruktion beschrieben. Das i. Segment beginnt bei Spielzug $\rho_i \in \mathbb{N}$ und endet vor $\rho_{i+1} = \rho_i' \in \mathbb{N}$. Eine gültige Segmentierung der Spur liegt vor, wenn die Segmente genau dann aufgeteilt werden, wenn kein konformes Verhalten möglich ist (5.3). Dann enthält die Spur $v_i = a_{\rho_i} \dots a_{\rho_i'}$ immer genau ein unerwartetes Verhalten. Wäre kein unerwartetes Verhalten in v_i, dann würde auch $Q_{\rho_i'}^{\rho_i}$ mindestens einen Ort enthalten und das Segment noch nicht abgeschlossen. Wäre zusätzliches unerwartetes Verhalten in v_i vor Spielzug ρ, mit $\rho_i < \rho < \rho_i'$, dann würde $Q_{\rho}^{\rho_i}$ bereits keine Elemente enthalten und ρ wäre als Minimum gewählt worden. Deshalb muss exakt ein unerwartetes Verhalten in v_i enthalten sein.

$$\rho_0 = 1 \wedge \rho_i' = \min\{\rho \mid \rho > \rho_i \wedge Q_{\rho}^{\rho_i} = \emptyset\} \tag{5.3}$$

Die Platz- und Zeitschranken ergeben sich aus den Schritten dieser Konstruktion. Für jeden Spielzug muss die Menge der Zielorte für alle möglichen Aufenthaltsorte bestimmt werden, also $\delta(Q_{\rho_i'}^{\rho_i}, a_i)$. Dazu muss für jeden der bis zu $|\mathbb{S}|$ Orte in der Transitionsrelation mit Größe $|\delta|$ nachgeschlagen werden. Wird eine perfekte Hash-Funktion verwendet, erfordert jedes Nachschlagen $O_{Zeit}(1)$. Daraus ergibt sich für jeden Spielzug der Zeitaufwand $O_{Zeit}(|\mathbb{S}|)$ und Platzbedarf $O_{Platz}(|\mathbb{S}| + |\delta|) = O_{Platz}(|A|)$. □

Um dies mit Resumption in Bezug zu setzen, wird im Folgenden betrachtet, wie Monika die Bewegungen von Sven überprüfen kann, wenn sie keine Informationen über seinen initialen Aufenthaltsort hat. Monika muss dazu ein Verdikt für alle möglichen Aufenthaltsorte von Sven fällen. Die Transitionsrelation δ wurde bereits erweitert (2.2), um eine Menge von Orten Q auf die Menge aller Zielorte zurückzugeben, die mit dem Verkehrsmittel e von mindestens einem Ort in Q erreicht werden können. δ liefert eine leere Menge zurück, wenn an keinem der möglichen Aufenthaltsorte das Verkehrsmittel benutzt werden kann. Um die Überprüfung fortzusetzen, muss sie mit der Fähigkeit zur Wiederaufnahme, also Resumption, erweitert werden. Das Wissen von Monika über Svens Aufenhaltsort wird zurückgesetzt, wenn sie seinen Spielzug nicht bestätigen kann. Die allgemeinste Annahme ist, dass Sven sich dann an einem beliebigen Ort befindet. Dies wird durch die Transitionsrelation δ^+ (5.4) erfasst.

$$\delta^+(Q,e) = \begin{cases} \delta(Q,e), & \text{if } \gamma(\text{Q,e}) = \top \\ S, & \text{otherwise.} \end{cases} \tag{5.4}$$

Die Bewertungsrelation γ wird auf ähnliche Weise erweitert (2.13). Sie benötigt eine Operation $\oplus$, um die Verdikte für alle möglichen Aufenthaltsorte zu kombinieren. Um unerwartetes Verhalten zu erkennen, kann der Operator $\oplus_{eb}$ (5.5) verwendet werden. Γ bezeichnet die Menge der zusammenzuführenden Verdikte. Solange Q^e nicht leer ist, gibt es mindestens einen Ort in Q mit dem geforderten Verkehrsmittel e und das Verhalten bleibt erwartet. Sonst wird unerwartetes Verhalten erkannt und das nächste Segment der Spur beginnt.

$$\oplus_{eb}(\Gamma) = \begin{cases} \top, & \text{if } \top \in \Gamma \\ \bot, & \text{otherwise} \end{cases} \tag{5.5}$$

Anstatt zur Laufzeit alle Möglichkeiten im Modell nachzuschlagen, kann auch ein deterministischer Automat aufgebaut werden, in dem ein Zustand die Menge aller mit erwartetem Verhalten erreichbaren Orte ist. Da dieser über die Potenzmenge der Orte gebildet wird und die potenziellen Möglichkeiten sich auf dem Spielplan zu bewegen beschreibt, wird er als *Potenzplan* bezeichnet. Er wird im Folgenden genutzt, um das Spiel zu veranschaulichen.

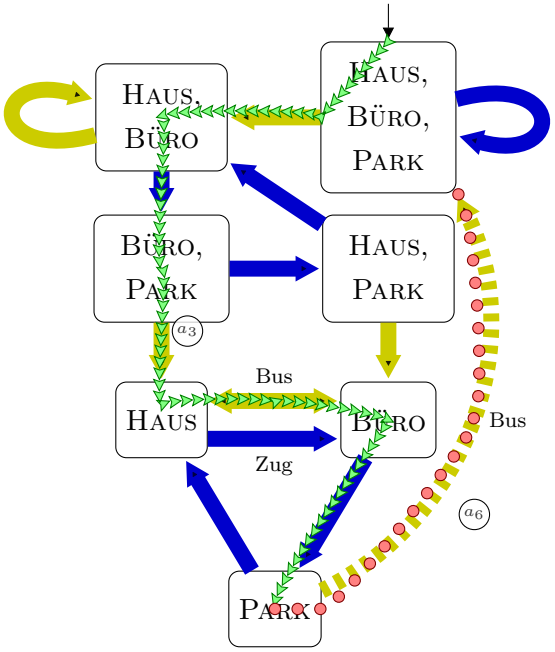


Abb. 5.3: Potenzplan des Beispiels aus Abbildung 5.2a inklusive der Interpretation für unerwartetes Verhalten der Spur aus Monikas Sicht.

Die Bewegungen von Sven werden auf dem Potenzplan $A^{\mathcal{P}} = \left\langle \mathbb{E}, \mathcal{P}(\mathbb{S}), s_{\mathbb{S}}, \delta^{\mathcal{P}} \right\rangle$ abgebildet (siehe Abbildung 5.3). $\mathcal{P}(\mathbb{S})$ ist die Menge aller möglichen Untermengen von $\mathbb{S}$, also ihre Potenzmenge. $\delta^{\mathcal{P}}$ entspricht δ^+, nur dass die Funktion den entsprechenden Zustand für die Menge der Orte zurückgibt statt der Menge selbst. Transitionen, die in A unerwartetes Verhalten waren, haben in $A^{\mathcal{P}}$ den Zustand $s_{\mathbb{S}}$ als Ziel. Dies schließt alle in A undefinierten Transitionen ein. Damit ist $\delta^{\mathcal{P}}$ vollständig und deterministisch. Somit kennt Monika immer den genauen Zustand von Sven in $A^{\mathcal{P}}$ und muss einfach nur seinen Bewegungen folgen (vgl. Theorem 1). Sie beginnt mit der Annahme, dass

Sven sich an einem beliebigen Ort des ursprünglichen Spielplans befindet, d. h. im Zustand $s_{\mathbb{S}} = \{\text{Haus}, \text{Büro}, \text{Park}\}$. Nach dem Spielzug a_3 sind alle andern Möglichkeiten eliminiert und der einzige Ort, an dem sich Sven befinden könnte, ist am Haus. Wie aus Abbildung 5.2b bekannt, ist Sven mit Spielzug a_3 abgewichen und zum Haus gesprungen. Jedoch handelt es sich noch immer um erwartetes Verhalten, denn es gäbe hier keine Abweichung, wenn Sven im Büro angefangen hätte, wie in Abbildung 5.4 dargestellt.

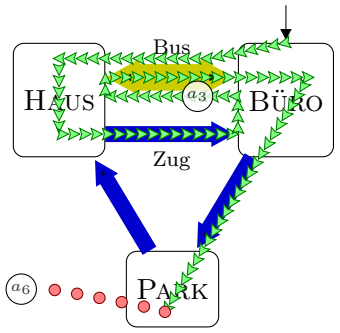


Abb. 5.4: Alternative zur verdeckten Bewegung von Sven, die zu einer identischen Spur wie in Abbildung 5.2b führt. Der neue Startpunkt hat zur Folge, dass keine Abweichung in Spielzug a_3 erfolgt.

Das interne Wissen über die Absichten von Sven spielt bei der Bewertung von erwartetem Verhalten keine Rolle – da keine Annahmen über die nicht beobachtbaren, internen Abläufe des SuO getroffen werden. Die zweite Abweichung bei Spielzug a_6 ist unerwartetes Verhalten, da es nun keine konforme alternative Interpretation mehr gibt. Im Vergleich zu Theorem 2 beträgt hier der Zeitaufwand nur $O_{Zeit}(1)$ für einen Spielzug. Er ist konstant, da nur ein Zustand ausgewertet werden muss, beispielsweise mit einer perfekten Hash-Funktion. Allerdings wird durch die Potenzmenge der Platzbedarf zur Repräsentation der Zustände exponentiell zunehmen; bis zu $O_{Platz}(2^{|\mathbb{S}|})$.

5.1.3 Resumption zur Erkennung von Abweichungen

Der vorherige Abschnitt hat gezeigt, dass jedes unerwartete Verhalten durch reine Beobachtung erkannt werden kann. Dieser Abschnitt untersucht, wie Resumption dabei helfen kann, Abweichungen eines SuO zu erkennen. Eine Abweichung ist ein Ereignis, das im aktuellen Zustand des SuO nicht erwartet wird. Abweichungen stehen also in einem direkten Bezug zum Zustand des SuO. Um das im Sinne des Beispiels zu beschreiben, muss Monika erkennen, ob sich Sven in einem Spielzug zum Abweichen entscheidet. Dies verändert, wie Monika mit der Ungewissheit über den Aufenthaltsort von Sven umgehen kann. In manchen Situationen kann es für sie unmöglich zu entscheiden sein, ob Sven abweicht. Ein Beispiel sind die beiden alternativen Pfade mit identischer Spur in Abbildung 5.2b und Abbildung 5.4. Sie braucht deshalb mindestens ein drittes Verdikt, um festzuhalten, dass die Situation *uneindeutig* ist.

Theorem 3. *In Bezug auf Abweichungen ist eine Bewegung a_i von* Sven *genau dann eindeutig für* Monika*, wenn für alle möglichen Aufenthaltsorte Q von* Sven *die Bewegung definiert oder undefiniert ist* (5.6).

$$\gamma(Q, a_i) \in \{\top, \bot\} \quad \Longleftrightarrow \quad \forall s_1, s_2 \in Q : \gamma(s_1, a_i) = \gamma(s_2, a_i) \tag{5.6}$$

Beweis. Die Implikation von rechts nach links folgt aus den Grenzen für die Bewegung von Sven. Er verhält sich *offensichtlich konform* mit a_i, wenn an allen Orten in Q das Verkehrsmittel verfügbar ist; er kann dann nicht abweichen. Er verhält sich *offensichtlich abweichend* mit a_i, wenn an allen Orten in Q das Verkehrsmittel nicht verfügbar ist; er kann sich dann nicht konform verhalten. In allen anderen Fällen ist die Situation nicht eindeutig, da es mindestens einen Aufenthaltsort $s_1 \in Q$ gibt, der das Verkehrsmittel anbietet und ebenfalls $s_2 \in Q$, an dem es nicht verfügbar ist. Wäre ihr Verdikt $\bot$, könnte sich Sven tatsächlich an s_1 befinden und sich eigentlich konform verhalten; wäre es $\top$, könnte er in Wahrheit gerade an s_2 abweichen. Deshalb muss ihr Verdikt dann $\perp\!\!\!\top$ sein. Ist das Verdikt von Monika $\top$ oder $\bot$ für eine Menge von Orten Q, dann muss das Verdikt für alle einzelnen Orte identisch sein. Damit folgt die Implikation von links nach rechts. □

Basierend darauf kann die Operation $\oplus_d$ (5.7) zur Kombination der Verdikte für die Erkennung von Abweichungen formuliert werden. Identische Verdikte werden zu eben diesem kombiniert und aus unterschiedlichen Verdikten wird ein uneindeutiges.

$$\oplus_d(\Gamma) = \begin{cases} \top, & \text{if } \{\top\} \equiv \Gamma \\ \bot, & \text{if } \{\bot\} \equiv \Gamma \\ \perp\!\!\!\top, & \text{otherwise} \end{cases} \tag{5.7}$$

Dies mag vielversprechend klingen, da es zumindest in manchen Fällen noch eindeutige Verdikte geben kann. Es könnte sogar die Erwartung wecken, dass Monika schließlich die Position von Sven immer weiter eingrenzen können sollte. Jedoch kann Sven jederzeit abweichen, wenn er sich nicht offensichtlich konform verhält. Deshalb ist in all diesen Fällen sein Aufenthaltsort erneut unbekannt und Q_{i+1} wird auf die Menge aller Orte gesetzt. Dies wird durch δ_d^+, einer Alternative zu δ^+ für die Erkennung von Abweichungen, erzwungen.

Daraus ergibt sich, dass Monika ihre Ungewissheit über den Aufenthaltsort von Sven nur reduzieren kann, wenn sie eine Sequenz von offensichtlich konformen Ereignissen beobachtet und durch die Sequenz mögliche Orte eliminiert werden. Bei der Betrachtung von ungewissen Zuständen in Abschnitt 2.2.6 wurden solche Sequenzen bereits vorgestellt und sie werden hier kurz rekapituliert. Sandberg [San05] beschreibt, wie eine *Synchronisations-* oder

Leitsequenz in einem vollständigen und deterministischen Mealy-Automaten innerhalb von $O_{Zeit}(|\mathbb{S}|^3 + |\mathbb{S}|^2 \cdot |\mathbb{E}|)$ gefunden werden kann – wenn eine solche Sequenz existiert. Eine Synchronisationssequenz ist ein Trace $x \in \mathbb{E}^*$, das den betrachteten Automaten, unabhängig von dem Ausgangszustand, in einen bestimmten, bekannten Zustand überführt, d. h. $\left|\delta^+(\mathbb{S}, x)\right| = 1$. Sie besteht aus einer Reihe von *Zusammenführungssequenzen*. Eine Leitsequenz kann auch *Trennsequenzen* enthalten und ergibt einen Trace $x \in \mathbb{E}^*$, das unterschiedliche Ausgaben für verschiedene Zielzustände garantiert (5.8). Ein Automat besitzt nicht immer eine Synchronisationssequenz, beispielsweise gibt es im Automaten zum Beispiel in Abbildung 5.2a keine. Ein minimierter Mealy-Automat hat aber immer eine Leitsequenz [San05].

$$\forall s_1, s_2 \in \mathbb{S}: \quad \delta(s_1, x) \neq \delta(s_2, x) \implies \gamma(s_1, x) \neq \gamma(s_2, x) \tag{5.8}$$

Leitsequenzen können auf die Entdeckung von Abweichungen angewendet werden. A enthält aber keine Ausgabe und ist unvollständig. Ersteres kann durch die Bewertungsrelation γ ersetzt werden und δ_d^+ ergänzt zu einem vollständigen Automaten. Eine Sequenz, die dann eindeutig einen bestimmten Ort identifiziert, wird auch *eindeutige Sequenz* genannt. Die reine Möglichkeit, dass Sven abweichen könnte, negiert aber jeden Gewinn durch Trennsequenzen, denn dies setzt die Suche von Monika stets zurück und sie muss, bereits nach *einer* möglichen Abweichung, erneut alle Orte als Möglichkeit betrachten. Da keine andere Ausgabe verfügbar ist, würden Trennsequenzen nur unter der Annahme funktionieren, dass nur offensichtliche Abweichungen auftreten. Da A im Beispiel keine Zusammenführungssequenz hat, kann $A^{\mathcal{P}}$ für die Erkennung von Abweichung vereinfacht werden, wie in Abbildung 5.5 dargestellt.

Monika weiß, dass Sven immer konform den *Zug* verwenden kann, dies sagt aber nichts über den Aufenthaltsort von Sven aus. Benutzt Sven den *Bus*, könnte er sich im Park befinden und abweichen, oder sich bei Haus und Büro konform verhalten. Daher ist Monika unschlüssig und startet erneut bei $s_{\mathbb{S}}$. Die undefinierten Transitionen könnten auch anders aufgelöst werden, dies würde aber zusätzliche Annahmen über die Abweichungen voraussetzen. Zielführender ist es, das Wissen über unerwartetes Verhalten heranzuziehen, um daraus auf Abweichungen zu schließen.

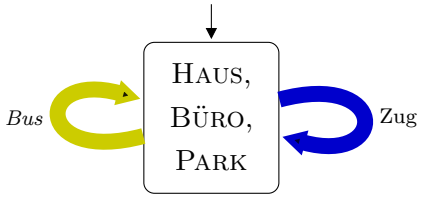


Abb. 5.5: Potenzplan des Beispiels aus Abbildung 5.2a zur Erkennung von Abweichungen aus Monikas Perspektive.

Theorem 4. *Jedes abgeschlossene Segment $a_{\rho_i} \dots a_{\rho'_i}$ des Traces enthält mindestens eine Abweichung.*

Beweis. Für einen Widerspruchsbeweis wird angenommen, dass es keine Abweichung in einem abgeschlossenen Segment des Traces gibt. Wenn es keine Abweichung gibt, kann Sven nur konforme Spielzüge ausgeführt haben. Folglich müsste es mindestens einen Pfad durch A geben, der zu dem Trace passt, also $\delta(\mathbb{S}, a_{\rho_i} \dots a_{\rho'_i}) \neq \emptyset$. Die Ungewissheit über den aktuellen Aufenthaltsort, $Q^{\rho_i}_{\rho'_i}$, müsste dann mindestens einen Ort enthalten. Dies steht allerdings im Widerspruch mit (5.3), das eine leere Menge verlangt, also $Q^{\rho_i}_{\rho'_i} = \emptyset$. Deshalb muss das Segment des Traces mindestens eine Abweichung enthalten. □

Ein Segment des Traces reicht also von einem unerwarteten Verhalten zum Nächsten. Irgendwo dazwischen ist in jedem Segment (mindestens) eine Abweichung. Dies soll im folgenden weiter eingegrenzt werden, um Abweichungen so genau wie möglich im Trace zu bestimmen. Die Segmente sind so konstruiert, dass sie stets mit unerwartetem Verhalten enden. Sie können also auf dieser Seite nicht gekürzt werden. Der Bereich für eine Abweichung kann aber noch am Anfang des Segments eingeschränkt werden, indem der maximale Start-Index gefunden wird, für den das Segment noch unerwartetes Verhalten enthält.

Theorem 5. *Das Trace $a_{\kappa_i} \dots a_{\rho'_i}$ ist der kürzeste Einschluss einer Abweichung vor Schritt ρ'_i, wenn κ_i so gewählt wird, dass das Segment gerade noch das unerwartete Verhalten enthält* (5.9).

$$\kappa_i = \max\{\kappa \mid \rho_i \leq \kappa < \rho'_i \wedge Q^{\kappa}_{\rho'_i} = \emptyset\} \tag{5.9}$$

Beweis. Der Beweis für die Existenz einer Abweichung in dem Trace-Segment bleibt identisch wie für Theorem 4: $Q^{\kappa_i}_{\rho'_i}$ ist nur leer, wenn eine Abweichung auftritt. Es bleibt also der Beweis, dass kein kürzeres Segment die Abweichung einschließt. Dazu betrachten wir die Wahlmöglichkeiten für κ. Im Maximalfall, also wenn $\kappa_i = \rho'_i - 1$, dann weicht Sven offensichtlich mit $a_{\rho'_i}$ ab, da dann kein Ort in $\mathbb{S}$ das Verkehrsmittel anbietet. Da mindestens ein Ereignis notwendig ist, um abzuweichen, kann es keine kürzere Sequenz geben. Die Gleichung 5.9 wählt den größtmöglichen Start-Index κ_i, für den das Segment noch unerwartetes Verhalten und somit eine Abweichung enthält. Der End-Index kann nicht reduziert werden, da dieser bereits als minimal gewählt wurde, wenn das Segment bei ρ_i anfängt (5.3). Wenn das Segment bei einem höheren Index anfängt als ρ_i, kann sich dadurch die Ungewissheit über den aktuellen Zustand (am Ende des Segments) nur erhöhen, also $Q^{\kappa_i}_{\rho'_i} \subseteq Q^{\kappa_i+1}_{\rho'_i}$. Denn das später startende Segment muss mindestens alle Aufenthaltsorte des längeren Segments ebenfalls berücksichtigen, für dieses können aber bereits gewisse Orte ausgeschlossen sein. Wenn κ_i maximal ist, dann ist $Q^{\kappa_i+1}_{\rho'_i}$ nicht mehr

leer. Es kann also auch keinen höheren Index geben, der garantiert eine Abweichung enthält, also für den alle Orte ausgeschlossen werden können. Deshalb ist der kürzeste Einschluss einer Abweichung vor Schritt ρ'_i das Trace-Segment $a_{\kappa_i} \dots a_{\rho'_i}$. □

κ kann durch eine Suche nach unerwartetem Verhalten bestimmt werden, die bei Spielzug ρ'_i beginnt und der Spur in rückwärtiger Richtung folgt. Im Vergleich zur Vorwärtssuche, wie sie in Theorem 2 beschrieben wird, kann ein Schritt rückwärts bis zu $|\mathbb{S}|$ Ausgangsorte haben. Deshalb erhöht sich die für einen Schritt benötigte Zeit auf $O_{Zeit}(|\mathbb{S}|^2)$. Für jeden der bis zu $|\mathbb{S}|$ Zielorte müssen bis zu $|\mathbb{S}|$ in die Menge aufgenommen (und auf Duplikate geprüft) werden. Die Anzahl der Schritte einer Suche ist durch die Länge des Segments begrenzt, das maximal so lange sein kann, wie die gesamte Spur, d. h. $\rho'_i - \kappa_i \leq \rho'_i - \rho_i \leq |v|$. Jedes Segment muss nur einmal durchsucht werden, demnach ist die Länge der Spur auch insgesamt die Grenze. Die Abbildungen der Rückwärtssuche benötigen den Platz $O_{Platz}(|\mathbb{S}|^2 \cdot |\mathbb{E}|)$.

Aus Theorem 2 und Theorem 5 folgt, dass es mindestens eine Abweichung in der Schnittmenge des vorwärts und rückwärts gefundenen unerwarteten Verhaltens gibt. Die Existenz von κ_i impliziert zudem, dass jedes Segment $a_{\rho_i} \dots a_{\kappa_i - 1}$ nur erwartetes Verhalten enthält, selbst wenn Sven dort abweicht. Diese Abweichungen ahmen erwartetes Verhalten so perfekt nach, das sie mit den verfügbaren Informationen von außen nicht erkannt werden können. Dies wird auch an den Möglichkeiten offensichtlich, die Sven in $A^{\mathcal{P}}$ hat.

Theorem 6. *Abweichungen in einem Segment $v_\top = a_{\rho_i} \dots a_{\kappa_i - 1}$ der Spur können nicht (mit den vorhandenen Informationen) erkannt werden.*

Beweis. Für den Widerspruchsbeweis wird angenommen, dass a_χ mit $\rho_i \leq \chi < \kappa_i$ eine erkennbare Abweichung sei. Eine Abweichung ist nur erkennbar, wenn sie keinem erwarteten Verhalten zuzuordnen ist. Es müsste also in $v_\top$ ein unerwartetes Verhalten geben. Das gesamte Segment $v = v_\top v_\bot = a_{\rho_i} \dots a_{\rho'_i}$ enthält nach Theorem 2 genau ein unerwartetes Verhalten. Theorem 5 grenzt dies genauer auf das Segment $v_\bot = a_{\kappa_i} \dots a_{\rho'_i}$ ein, also bei oder nach Spielzug κ_i. Es kann kein weiteres unerwartetes Verhalten vor Spielzug κ_i in v geben, da nur eins in v enthalten ist. Gäbe es ein weiteres unerwartetes Verhalten, wäre v bei der Segmentierung weiter aufgespalten worden. Deshalb wird jede Abweichung, die in $v_\top$ auftreten könnte, als erwartetes Verhalten beobachtet und deshalb nicht erkannt. □

Die Abweichungen können nicht erkannt werden, da es keine Möglichkeit gibt, sie mit den verfügbaren Informationen von erwartetem Verhalten zu unterscheiden. Wenn jedoch weitere oder detailliertere Beobachtungen des Systems verfügbar sind, können diese Abweichungen auch erkennbar werden.

Sofern die Ungewissheit über den aktuellen Zustand auf einen einzelnen Zustand reduziert wird, ohne dass eine (nicht erkennbare) Abweichung auftritt, bedeutet dies ebenfalls, dass der nächste Spielzug, bei dem unerwartetes Verhalten erkannt wird, genau der Abweichung entspricht. Wenn also genügend Spielzüge zwischen zwei Abweichungen liegen, kann ein Monitor mit Resumption exakt die Abweichungen identifizieren.

Theorem 7. *Wenn n nicht überlappende, eindeutige Sequenzen ohne unerwartetes Verhalten beobachtet werden, entspricht der von* Monika *angenommene Ort dem tatsächlichen Aufenthaltsort von* Sven*, wenn dieser vorher nicht mindestens n mal abgewichen ist. Wenn jedoch nicht zumindest eine der eindeutigen Sequenzen eine offensichtlich konforme Synchonisationssequenz ist, können mindestens n Abweichungen aufgetreten sein.*

Beweis. Für den ersten Teil wird gezeigt, dass nicht alle eindeutigen Sequenzen mit nur $n - 1$ Abweichungen getäuscht werden können, also mindestens eine den wahren Ort von Sven verrät. Eine eindeutige Sequenz lässt, ausgehend von allen Orten, an ihrem Ende nur einen möglichen (erwarteten) Aufenthaltsort zu. Wenn Sven nicht innerhalb der Sequenz abweicht, entspricht dies seinem realen Ort. Da die eindeutigen Sequenzen nicht überlappen, gibt es mindestens eine eindeutige Sequenz u, in der keine Abweichung auftritt. Deshalb offenbart u den Aufenthaltsort von Sven. Theorem 1 garantiert die Erkennung von Abweichungen, wenn der Aufenthaltsort bekannt ist. Da eine eindeutige Sequenz kein unerwartetes Verhalten enthalten kann und dies nicht erkannt wurde, muss u nach den Abweichungen von Sven auftreten. Mit einer zusätzlichen Abweichung, also insgesamt n, könnte Sven alle eindeutigen Sequenzen täuschen und sein wahrer Aufenthaltsort bliebe unbekannt.

Für den zweiten Teil wird der Aufbau von eindeutigen Sequenzen betrachtet. Sie sind eindeutig, da sie die Ungewissheit über den aktuellen Zustand $|Q|$ auf eins reduzieren. Wie bei Leitsequenzen kann die Reduktion durch Zusammenführungs- und Trennsequenzen erfolgen. Eine Zusammenführungssequenz nutzt die Struktur des Automaten, beispielsweise wenn zwei Orte mit dem gleichen Verkehrsmittel zu einem Ort führen. Während der Zusammenführungssequenz ist per Definition keine Abweichung möglich, da alle Ausgangsorte das Verkehrsmittel anbieten. Da eine Synchonisationssequenz aus verketteten Zusammenführungssequenzen besteht, ist sie offensichtlich konform und keine Abweichung kann aufgetreten sein, wenn sie beobachtet wurde. Die Beobachtung einer Synchonisationssequenz garantiert damit, dass der wirkliche Aufenthaltsort von Sven bekannt ist. Trennsequenzen funktionieren anders. Sie reduzieren die Ungewissheit des aktuellen Aufenthaltsorts, indem sie Orte eliminieren, an denen Sven abweichen müsste, um die Spur zu erzeugen. Daher gibt es dann mindestens diese Möglichkeit in einer Trennsequenz abzuweichen. Sofern keine der beobachteten eindeutigen Sequenzen eine Synchonisationssequenz

ist, enthält jede mindestens eine eigene Trennsequenz. Wenn es n eindeutige Sequenzen gibt, kann es dann auch mindestens n Abweichungen geben. □

Es mag unwahrscheinlich klingen, mehrere eindeutige Sequenzen vor unerwartetem Verhalten erkennen zu können. Jedoch sind manchmal die seltenen Abweichungen von Interesse. In diesem Fall kann es viele Ereignisse zwischen Abweichungen geben, die zur Wiederaufnahme der Überprüfung verwendet werden können. Protokolle, wie das Beispiel für einen Abonnement-Dienst in Abbildung 2.1, enthalten häufig *eindeutige Ereignisse*. Das sind eindeutige Sequenzen, die aus nur einem Ereignis bestehen. Die Beobachtung von mehreren eindeutigen Ereignissen und (kurzen) eindeutigen Sequenzen kann das Vertrauen erhöhen, dass der Monitor nicht getäuscht wurde. Es kann aber nur die Erkennung von unerwartetem Verhalten garantiert werden.

5.2 Automatisierte Erweiterung von Automaten mit Resumption

Dieser Abschnitt betrachtet, wie ein Automat mit Resumption erweitert und wie dies formal beschrieben werden kann. Jeder Monitor, der auf einer Spezifikation des Soll-Verhaltens basiert, kann mit Resumption erweitert werden. Selbst wenn der Monitor bereits eine vollständige Transitionsrelation besitzt, kann er mit Resumption verbessert werden, wenn er dafür Zustände verwendet, in denen keine weitere Überprüfung stattfindet, wie $s_\perp$ in Abbildung 5.1. Im Folgenden werden tiefgestellte Indices verwendet um die Bestandteile des ursprünglichen Monitors ($\mathcal{L}$), der Erweiterung ($\mathcal{R}$) und des erweiterten Monitors ($\mathcal{E}$) zu unterscheiden. $\mathrm{M}_\mathcal{E}$ entsteht durch Kombination der Mengen und Relationen von $\mathrm{M}_\mathcal{L}$ mit den entsprechenden von $\mathrm{M}_\mathcal{R}$, wobei $\mathrm{M}_\mathcal{L}$ normalerweise im Fall von Widersprüchen bevorzugt wird. Bei Bedarf kann $\delta_\mathcal{R}$ aber $\delta_\mathcal{L}$ für frei wählbare Verdikte, beispielsweise $\perp$, überstimmen, um Resumption zu starten, statt einen Fehlerzustand zu betreten.

Beispiel 1 (Resumption Extension)**.** *Abbildung 5.6 zeigt eine mögliche Erweiterung des Automaten aus Abbildung 2.1 bzw. Abbildung 5.1. Anstatt für unerwartete Ereignisse einen Zustand zu betreten, der nicht mehr verlassen wird, ignoriert der erweiterte Monitor* $\mathrm{M}_\mathcal{E}$ *solche Ereignisse und bleibt im bereits aktiven Zustand.* $\mathrm{M}_\mathcal{E}$ *hat durch die Erweiterung eine vollständige Transitionsrelation und ist in der Lage, die Überwachung wiederaufzunehmen, nachdem er einen Verstoß gemeldet hat. Der ursprüngliche Monitor wurde mit Resumption erweitert.*

Die Erweiterung mit Resumption kann prinzipiell auf eine beliebige Art und Weise erfolgen. Die Verwendung einer sog. Resumption-Strategie ($\mathcal{R}$) zur Definition der Erweiterung ermöglicht diese zu automatisieren und Annahmen über die Rahmenbedingungen zu treffen,

unter denen die Verdikte von $M_{\mathcal{E}}$ korrekt sind. Im Kern besteht eine Resumption-Strategie aus einer Funktion, die ein Ereignis und eine Menge an (möglichen) aktiven Zuständen als Eingabe erhält (5.10). Sie gibt die Menge der Zustände zurück, die als Kandidaten für die Wiederaufnahme in Betracht kommen. Sie definiert damit das berücksichtigte Fehlermodell für den Aufbau von δ^+. Das Beispiel in Abbildung 5.6 wird deshalb nur für bestimmte Fehlermodelle brauchbare Verdikte liefern, wie die Evaluation auch in Abschnitt 7.2.3 bestätigen wird. Eine solche $\mathcal{R}$-basierte Erweiterung mit Resumption kann leicht ausgetauscht werden, um den Monitor an ein anderes Anwendungsszenario mit einem anderen Fehlermodell anzupassen. Die Menge aller potenzieller Kandidaten sei $\mathbb{S}_C = \mathbb{S}_{\mathcal{L}} \cup \{s_{\mathcal{R}}\}$ und $\mathcal{P}(\mathbb{S}_C)$ die Menge aller Untermengen.

$$\mathcal{R} : \mathcal{P}(\mathbb{S}_C) \times \mathbb{E} \to \mathcal{P}(\mathbb{S}_C) \tag{5.10}$$

Mit $\mathcal{R}$ lassen sich die notwendigen zusätzlichen Zustände und Transitionen bestimmen, die zur Erweiterung des ursprünglichen Monitors benötigt werden. Wenn die Mengen $\mathbb{S}_{\mathcal{L}}$ und $\mathbb{E}$ endlich sind, können in einem vorbereitenden Schritt die Zustände $\mathcal{P}(\mathbb{S}_C) \setminus \mathbb{S}_C$ erzeugt werden. $\mathcal{R}$ definiert die Transitionsrelation der Erweiterung. Die Ziele der Transitionen können also durch Auswertung von $\mathcal{R}$ für jede Kombination aus Zustand und Ereignis bestimmt werden. Falls $\mathcal{R}(q, e)$ eine leere Menge zurückgibt, oder nur Zustände, die von denen aus kein Zustand aus $\mathbb{S}_C$ erreicht werden kann, gibt es Situationen in denen eine Wiederaufnahme mit der gewählten Strategie nicht möglich ist. Die Existenz solcher Zustände hängt von $\mathcal{R}$ und der Spezifikation ab. Alle Zustände können entfernt werden, die nicht von einem Zustand in $\mathbb{S}_C$ (indirekt oder direkt) erreicht werden können.

Alternativ kann $\mathcal{R}$ zur Laufzeit als Transitionsrelation während Resumption verwendet werden. Sobald $\mathcal{R}$ einen einzelnen Zustand aus $\mathbb{S}_{\mathcal{L}}$ zurückliefert, kann $M_{\mathcal{L}}$ die Überprüfung in diesem Zustand fortsetzen. Ansonsten werden die Kandidaten parallel verfolgt und nicht konforme entfernt. Sofern $s_{\mathcal{R}}$ ein Kandidat ist, wird $\mathcal{R}$ verwendet, ansonsten $\delta_{\mathcal{L}}$. $\gamma_{\mathcal{R}}$ wird durch (2.13) mit einem geeigneten $\oplus$ bestimmt.

Der vorgestellte Ansatz kann in die Verifikation zur Laufzeit mit Rücksetzen von Cimatti, Tian und Tonetta [CTT19] eingeordnet werden. So kann das Soll-Verhalten als Prüflogik interpretiert werden und die Erweiterung mit Resumption erzeugt die Annahme. Die explizite Angabe von letzterem entfällt mit dem vorgestellten Ansatz. Verletzungen der Annahme könnten, sofern beobachtbar, in das Fehlermodell mit aufgenommen werden. Erzeugt die Erweiterung einen vollständig definierten Automaten als Annahme, der nicht verletzt werden kann, dann ist ein Rücksetzen der Überprüfung unbegrenzt möglich und dies geschieht automatisch nach jeder Abweichung.

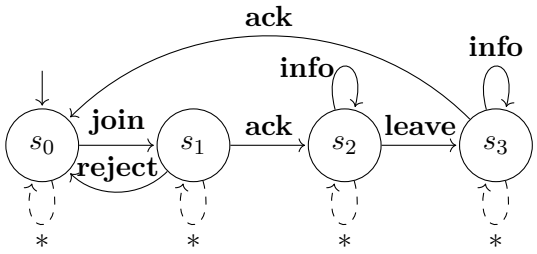


*Abb. 5.6: Erweiterung des Automaten aus Abbildung 2.1 mit einer einfachen Resumption-Strategie. Die hinzugefügten Transitionen sind gestrichelt dargestellt. Beschriftungen für konforme Ereignisse sind **fett**-gedruckt. '*' bezeichnet alle Ereignisse, für die keine andere Transition im Ausgangszustand definiert ist.*

5.3 Resumption für hierarchische Automaten

Dieser Abschnitt betrachtet die Auswirkungen hierarchischer Automaten, wie sie in Abschnitt 2.2.3 vorgestellt werden, auf Resumption. Zunächst eine kurze Rekapitulation, wie hierarchische Automaten in dieser Arbeit definiert sind. Ein solcher Automat $HA = \langle \mathbb{A}, \mathbb{E}, \alpha \rangle$ besteht aus einer Menge von sequentiellen Automaten $\mathbb{A}$, die über eine Kompositionsfunktion α verknüpft sind. Die Automaten werden von α einem Zustand eines anderen Automaten zugeordnet. Wenn dieser Zustand aktiv ist, wird auch der zugeordnete Automat aktiviert und verfeinert damit die Verhaltensbeschreibung des Zustands. Ganz intuitiv kann also der einzelne Zustand durch seine Verfeinerungen ersetzt werden. Dies entspricht der von Mikk, Lakhnechi und Siegel [MLS97] vorgestellten Abbildung auf eine Kripke-Struktur. Der sequentielle Automat kann aber in bestimmten Fällen (mehrfach) exponentiell mehr Zustände zur Beschreibung benötigen [Yan00; vZij04].

Ein naiver Ansatz wäre eine Resumption-Strategie $\mathcal{R}$ auf den flachen Automaten flach(HA) anzuwenden, um $HA_{\mathcal{E}_1} = \text{flach}(HA)_{\mathcal{E}}$ zu bestimmen. Dies führt zu einem gültigen Ergebnis, aber auch zu der bereits erwähnten mehrfach exponentiellen Zunahme in der Anzahl der Zustände. Die Erweiterung mit Resumption unter der Annahme, dass im Fall einer Abweichung ein beliebiger Zustand eingenommen werden kann, erhöht die Anzahl der Zustände erneut exponentiell, da $\mathcal{P}(\text{flach}(HA))$, die Potenzmenge der Zustände des flachen Automaten, den Suchraum bildet. Hierbei handelt es sich letztendlich um die Abbildung eines nicht-deterministischen Automaten auf einen deterministischen. Dabei kann die Anzahl der Zustände exponentiell zunehmen [Yan00].

Die Minimierung eines nicht-deterministischen Automaten ist für sich bereits ein schwieriges Problem [JR91]. Selbst kleine Erweiterungen mit nicht-deterministischem Verhalten, wenn beispielsweise nur eine nicht-deterministische Transition je Zustand erlaubt wird, führen bereits zu NP-vollständiger Komplexität [Mal04]. Die Minimierung von deterministischen, flachen Automaten ist effizient in $O_{Zeit} nlogn$ möglich [Hop71]. Der Automat für die Er-

kennung des ersten unerwarteten Verhalten ist deterministisch, da der Zustand nach einer Abweichung nicht relevant ist und somit zu einem Fehlerzustand zusammengefasst werden kann. Aus dem minimierten Automaten kann der Fehlerzustand wieder entfernt werden, um dann Resumption anzuwenden. Die Minimierung hat keinen Einfluss auf die Korrektheit der Entscheidung, ob ein Trace gültig ist oder nicht, legt aber fest, welchem der ursprünglichen Zustände dieses Verhalten durch Resumption zugeordnet wird. Eine Unterscheidung ist mit den verfügbaren Informationen auch nicht möglich, da die Minimierung genau solche Zustände zusammenfasst, von denen alle ausgehenden Pfade nicht anhand ihres Traces unterschieden werden können. An der potenziell (mehrfach) exponentiellen Zunahme der Anzahl der Zustände durch Resumption ändert dies aber nichts.

Eine weitere Möglichkeit zur Vermeidung zusätzlicher Zustände ist, den hierarchischen Automaten soweit wie möglich in dem mit Resumption erweiterten Automaten zu erhalten. Dazu kann Resumption für den flachen Automaten bestimmt und anschließend die Erweiterung zurück auf den hierarchischen Automaten abgebildet werden. Der Ablauf ist in Algorithmus 1 beschrieben. Eine Minimierung des flachen Automaten in Zeile 3 hilft unnötige Zustände zu vermeiden, wenn der Algorithmus zur Erstellung des flachen Automaten keinen minimalen Automaten garantiert. Die Minimierung dient neben der Reduktion der Zustände auch dazu, eine für Resumption zuträgliche Eindeutigkeit zu gewährleisten. Da die Erweiterung mit Resumption am Ende zurück auf den ursprünglichen Automaten abgebildet wird, kann der Monitor im normalen Betrieb noch immer zwischen den Zuständen unterscheiden; nach unerwartetem Verhalten wird er aber immer den in der Minimierung erhaltenen Zustand wählen. Algorithmus 1 gibt die Erweiterung des Wurzelautomaten mit Resumption zurück.

Algorithmus 1 Resumption für hierarchische Automaten

1: **procedure** $\mathcal{R}_{HA}(HA)$
2: $A_1 \leftarrow \text{flach}(HA)$
3: $A_2 \leftarrow \min(A_1)$
4: $A_3 \leftarrow \mathcal{E}(A_2)$
5: $A_4 \leftarrow \min(A_3)$
6: **return** $\{\langle s, e, s' \rangle \in \delta_{A_4} \mid \langle s, e, s' \rangle \notin \delta_{A_1}\}$

Ein Zustand von A_4 entspricht jeweils einer Menge von Zuständen, die aus der Vereinigungsmenge von Konfigurationen gebildet wird, die *HA* entsprechend eines Traces gerade einnehmen könnte. Um die Erweiterung und den ursprünglichen Automaten zu verbinden, werden die Zustände von A_4, die einer Konfiguration von *HA* entsprechen, durch die Zustände von *HA* ersetzt. Die vom Algorithmus zurückgegebenen Transitionen mit diesen Konfigurationen als Quelle oder Ziel werden entsprechend übersetzt, indem die Zustände des Wurzelautomaten als Quelle und Ziel verwendet und die Unterzustände in *sr* bzw. *td* angegeben werden. Mikk, Lakhnechi und Siegel [MLS97] beschreiben eine Übersetzung

solcher Transitionen allgemein von Statecharts [Har87] in hierarchische Automaten. In Statecharts können solche Transitionen allgemein vorkommen. Sie werden dort als *full compound transitions* bezeichnet und können etwa als *volle Verbund-Übergange* ins Deutsche übersetzt werden.

Dies führt zu der Frage, welche Auswirkungen es hat, wenn eine Resumption-Strategie auf jeden der sequentiellen Automaten von *HA* angewendet wird. Um die Herausforderungen dabei herauszustellen, wird im Folgenden der Aufbau von $HA_{\mathcal{E}_2} = \langle \mathbb{A}_{\mathcal{E}}, \mathbb{E}, \alpha_{\mathcal{E}} \rangle$ betrachtet. Die Menge der sequentiellen Automaten sei zusammengesetzt durch Transformation jedes einzelnen Automaten mit der Resumption-Strategie, d. h. $\mathbb{A}_{\mathcal{E}} = \bigcup_{A \in \mathbb{A}} A_{\mathcal{E}}$. Die Bildung der Kompositionsfunktion $\alpha_{\mathcal{E}}$ gestaltet sich hingegen nicht so trivial. Da nach der vorgestellten Semantik ein Automat (außer dem Wurzelautomat) immer genau einem Zustand $s \in \mathbb{S}$ zugeordnet wird, kann er nicht s und jeden anderen Zustand aus $S = \{s' \in \mathcal{P}(\mathbb{S}) \mid s \in s'\}$ gleichzeitig verfeinern. Zudem würde nach aktueller Semantik der Zustand der Verfeinerung bei einem Zustandsübergang von s' zu s oder einem anderen Zustand aus S zurückgesetzt und führt zu einer langsameren Konvergenz der Wiederaufnahme. Gravierender ist jedoch, dass $HA_{\mathcal{E}_2}$ die impliziten Abhängigkeiten zwischen zwei parallelen Automaten nicht berücksichtigt. Diese ergeben sich dadurch, dass es ausreicht, wenn einer der parallelen Automaten eine Nachricht akzeptiert. Wird dies bei der Resumption nicht berücksichtigt, kann der Automat eigentlich offensichtliche Abweichungen nicht mehr erkennen oder er lehnt gültige Pfade ab.

Die Berücksichtigung der Abweichungen ist aber nicht trivial. Für jede der Abhängigkeiten müssten die Zustände in dem betroffenen Automaten vervielfältigt und die Zustandsübergänge entsprechend dem parallelen Automaten mit geeigneten Bedingungen versehen werden. Dies betrifft auch rekursiv Verfeinerungen der involvierten Zustände. Sofern versteckte Abhängigkeiten nicht vermieden werden können, führt dies letztendlich zum Ausmultiplizieren der parallelen Automaten. Die Umgehung von letzterem war die eigentliche Motivation $HA_{\mathcal{E}_2}$ zu betrachten. Die Abhängigkeiten können nur durch ein spezialisiertes Fehlermodell vermieden werden. Das allgemeine Fehlermodell erlaubt eine komplett neue Konfiguration des Systems nach einer Abweichung. Dadurch sind auch Automaten von Abweichungen betroffen, die von den (nicht) auftretenden Nachrichten selbst keinen Gebrauch machen. Zu einer speziellen Anpassung des Fehlermodells kann allgemein, also ohne Betrachtung des konkreten Anwendungsfalls, nur die Aussage getroffen werden, dass der Monitor nur solange korrekt funktioniert, bis eine Abweichung auftritt, die nicht von seinem Fehlermodell abgedeckt wird. Dies ist dann möglich, da bestimmte Konfigurationen als Kandidaten für Resumption ausgeschlossen werden, wenn eine andere Definition von δ^+ (5.4) verwendet wird. Ohne eine Spezialisierung des Fehlermodells, liegt die Annahme nahe, dass der hierarchische Automat allgemein nicht ohne schlimmstenfalls (mehrfach) exponentielle Zunahme der Zustände mit Resumption versehen werden kann.

5.4 Resumption für Automaten mit Kontexten

Kontexte ermöglichen zur Laufzeit Daten zu hinterlegen, die für weitere Auswertungen erforderlich sind. Dieser Abschnitt betrachtet, was bei der Resumption von Automaten mit Kontexten berücksichtigt werden muss. In Abschnitt 4.6.2 wurde bereits gezeigt, dass ein Kontext als ein Automat betrachtet werden kann, der parallel zu dem Automaten des Kontrollflusses läuft. Entsprechend kann ein Kontext prinzipiell wie parallele Automaten in einem hierarchischen Automaten behandelt werden. Allerdings bedeutet dies ebenfalls, dass im Allgemeinen die Anzahl der Zustände exponentiell zunimmt. Die Zunahme an Zuständen ist durch Kontexte aber extremer, da mit Parametern sehr effizient eine Vielzahl an Zuständen ausgedrückt werden kann. Jede kleine Änderung an einem der Parameter kann zu einem neuen Zustand führen und verändern, ob eine Nachricht mit dem Kontext kompatibel ist oder nicht. Beispielsweise ist ein typischer Anwendungsfall für Kontexte, dass ein Parameter die Kennzahl zur Identifikation einer Verbindung enthält und nur Nachrichten dieser Verbindung sollen von dem Automaten überprüft werden. Der Parameter wird dann zur Laufzeit im Zustand des Kontexts hinterlegt. Genau dann, wenn Nachrichten diesen Wert enthalten, sind sie kompatibel mit dem Kontext.

Für Resumption im allgemeinen Fall bedeutet dies, dass zunächst das Kreuzprodukt aus Zuständen und Kontextzuständen, also jeder Zustand mit allen möglichen Parametrierungen des Kontexts, als Kandidat berücksichtigt werden muss. Eine symbolische Repräsentation und Ausführung [z. B. DAn+18; Lau+19] kann helfen, nicht alle parametrisierten Zustände explizit auflisten zu müssen. Statt mit expliziten Werten arbeiten diese Ansätze mit Bedingungen, die gültige Werte für die Parameter in Abhängigkeit von den Eingaben abstrakt beschreiben. Dabei sind, wie Lauko u. a. [Lau+19] darlegen, neue Herausforderungen zu überwinden. Sie verwenden eine Baumstruktur für Repräsentation der Bedingungen. Damit diese nicht unendlich wachsen, muss erkannt werden können, ob Zweige noch erfüllbar oder identisch sind. Sie verwenden dafür Lösungsverfahren aus der Modulo-Theorie der Erfüllbarkeit (SMT, engl.: *Satisfiability Modulo Theory*), was die Frage aufwirft, ob solch ein Ansatz noch effizient zur Auswertung eines Traces (zur Laufzeit) eingesetzt werden kann. Andererseits haben D'Antoni u. a. [DAn+18] bereits für symbolische Register-Automaten gezeigt, dass bei diesen die Auswertung linear mit der Größe des Inputs wächst. Sie treffen aber keine Aussage über den Einfluss der Größe des Automaten.

Auch für Kontexte kann eine Einschränkung der Kandidaten für Resumption die Größe des resultierenden Automaten reduzieren, wie im folgenden an einem Beispiel geschildert wird. Dies resultiert aber auch darin, dass die Überprüfung diese ausgeschlossenen Situationen nicht erkennt. Beispielsweise kann zu einer Verbindung eine Kennzahl zur Identifikation im Kontext hinterlegt werden, um weitere Nachrichten dieser Verbindung herausfiltern zu können und nur diese zu analysieren. Da in diesem Abschnitt Kontexte im Vordergrund

stehen, sind etwaige weitere Verbindungen nicht von Interesse. Abweichungen können dann zusätzliche oder fehlende Nachrichten mit der korrekten Kennzahl oder solche mit einer anderen Kennzahl sein. Der in Abbildung 5.7 gezeigte Automat hat zwei normale Zustände s_0 und s_1 und einen Kontext, der die Kennzahl speichert. Der Zustand s_1 wurde für die verschiedenen Zustände des Kontext dupliziert. Eine allgemeine Erweiterung mit Resumption ist durch gestrichelte Transitionen und Zustände eingezeichnet.

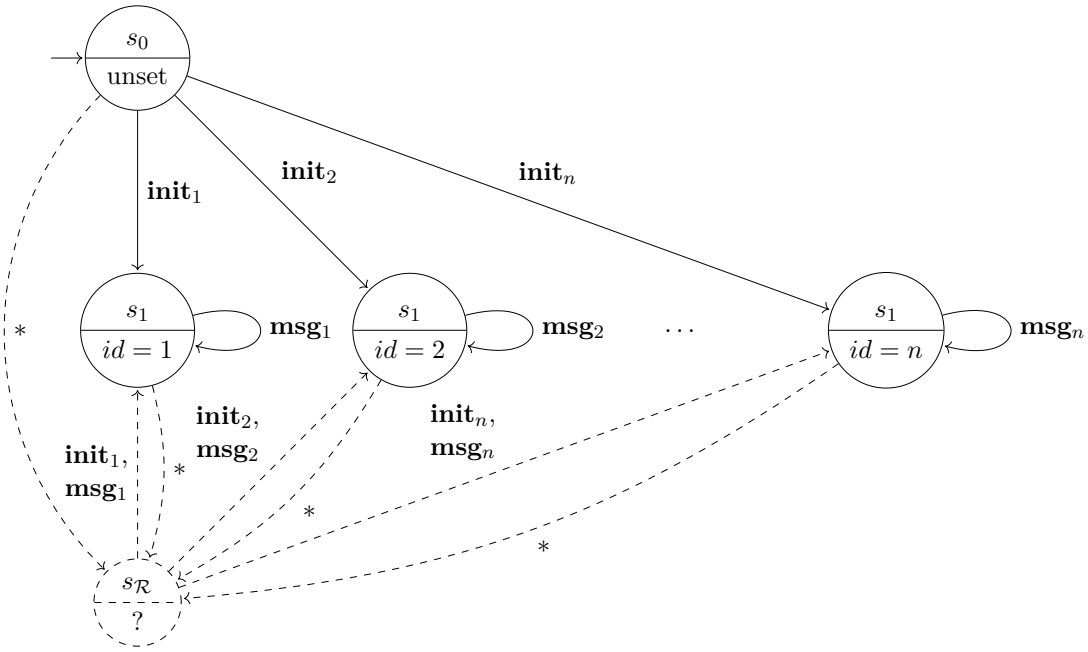


*Abb. 5.7: Beispiel eines Automaten mit Kontext, der mit Resumption erweitert wurde. Die Kontext-Zustände wurden in die Darstellung der Zustände integriert. Der Name des ursprünglichen Zustands ist jeweils in der oberen Hälfte, die Werte von Kontext-Variablen sind in der unteren Hälfte angegeben. Die hinzugefügten Transitionen sind gestrichelt dargestellt. Beschriftungen für konforme Ereignisse sind **fett**-gedruckt. '*' bezeichnet alle Ereignisse, für die keine andere Transition im Ausgangszustand definiert ist.*

In der Abbildung von Kontexten auf einfache Automaten gibt es eine bestimmte Untermenge der Zustände, die den Kontext mit der gewählten Kennzahl repräsentieren. In Abbildung 5.7 ist dies jeweils s_1 mit dem entsprechenden Kontext, allgemein können dies aber mehrere Zustände sein. Nach dem allgemeinen Fehlermodell könnte nach einer Abweichung ein beliebiger Zustand auch außerhalb dieser Untermenge gewählt werden. Damit kann die Kennzahl (und der Zustand) beliebig sein. Ohne eine Annahme muss jeder Wert des Kontexts erst durch erneute Beobachtung wieder herausgefunden werden. Da die Kennzahl in weiteren Nachrichten direkt beobachtet werden kann, sind diese Kandidaten auch vergleichsweise einfach zu bestätigen. Handelt es sich jedoch um Werte, die nicht explizit in der weiteren Kommunikation auftauchen, kann die Bestätigung eines Kandidaten schwer bis unmöglich werden. In einem Fall mit mehr als einem Wert können dazu auch mehrere Nachrichten und Zwischenschritte notwendig sein.

Bei Abweichungen mit korrekter Kennzahl liegt vielleicht intuitiv die Annahme nahe, dass die Kennzahl auch in Zukunft gleich bleibt; bei anderer Kennzahl, dass diese im Folgenden der ursprünglichen oder neuen Kennzahl entspricht. Dies schränkt die Zahl der Kandidaten für die Kennzahl nach einer Abweichung ein. Verursacht die Abweichung einen Wechsel des SuO zu einer anderen Kennzahl, wird der Kontext die eigentlich zugehörigen Nachrichten ignorieren. Werden sie nicht an anderer Stelle im Automaten akzeptiert, wird der Monitor weitere Verstöße melden.

Die in der Evaluation betrachtete Strategie $\mathcal{R}_{\text{wait}}$, die für Resumption den vorherigen Zustand beibehält, also dort *wartet*, besitzt ein vergleichbares Verhalten. In den Ergebnissen der Evaluation in Abschnitt 7.2.3 wird auch experimentell gezeigt, dass *Warten* für Abweichungen, die den Zustand verändern, zu vielen zusätzlich gemeldeten Verstößen führt. Zudem ist die Vollständigkeit zwar hoch, aber nicht perfekt, da es zu Situationen kommen kann, in denen der Monitor so getäuscht wird, dass er eigentlich unerwartetes Verhalten akzeptiert oder erst verzögert bemerkt.

5.5 Resumption für Automaten mit Instanzen

Die Erweiterung von Automaten um Instanziierung ermöglicht einmal spezifiziertes Verhalten in parallelen Handlungssträngen unabhängig zu überprüfen. In diesem Abschnitt wird untersucht, wie sich dies auf Resumption auswirkt. Bei Instanzen handelt sich um einen Sonderfall eines hierarchischen Automaten, bei dem die parallelen Automaten den gleichen Aufbau haben. Prinzipiell kann dies auch explizit modelliert werden, Instanzen sind aber beispielsweise hilfreich, wenn im Vorhinein nicht festgelegt werden kann oder soll, wie viele parallele Abläufe es gibt. Solange der zugrundeliegende Automat endlich viele Zustände hat, bleibt auch der Zustandsraum des Automaten mit Instanzen endlich. Haben zwei Instanzen den gleichen aktiven Zustand, sind sie äquivalent und nur eine der Instanzen wird für die Überprüfung benötigt, da beide das gleiche Verdikt für alle folgenden Ereignisse liefern werden. Entsprechend ist durch eine Instanz für jeden Zustand des Automaten eine Obergrenze an unterscheidbaren Instanzen gegeben.

Im allgemeinen Fall muss für Resumption angenommen werden, dass beliebig viele Instanzen in unterschiedlichen Zuständen nach einer Abweichung aktiv sind. Neue Instanzen können hinzukommen oder bereits aktive Instanzen können in ihrem Zustand verändert oder komplett deaktiviert werden. Eine Möglichkeit damit umzugehen wäre anzunehmen, dass dann alle $|\mathbb{S}|$ unterscheidbare Instanzen aktiv sind. Während zu Beginn nicht bekannt ist, welche Instanzen tatsächlich aktiv sind, können Instanzen schrittweise zusammengefasst werden, wenn sie durch die Ereignisse in identische Zustände überführt werden. Dies erinnert zunächst an Resumption für den einfachen Automaten. Im Fall von Instanzen können aber

nur die Zusammenführungssequenzen verwendet werden. Trennsequenzen finden bei rein positiver Soll-Beschreibung keine Anwendung für Instanzen, da sie prinzipiell wie parallele Automaten interpretiert werden müssen und diese ein gemeinsames Verdikt über $\oplus_{eb}$ bilden; es genügt, wenn einer der Automaten das Ereignis akzeptiert.

Wenn beispielsweise der Automat aus Abbildung 5.7 instanziiert wird (und erst dann die Erweiterung mit Resumption erfolgt), werden nach dem Auftreten von unerwartetem Verhalten `msg` Ereignisse mit beliebiger Kennzahl akzeptiert. Allgemein muss nach der Abweichung angenommen werden, dass alle möglichen Instanzen aktiv sein könnten. Und alle möglichen Instanzen entsprechen allen möglichen Kontextzuständen. Da es in dem Automaten keine Zusammenführungssequenzen gibt, können keine Instanzen eliminiert werden.

In der Praxis sind hier häufig weitere Einschränkungen plausibler (Kombinationen von) Instanzen bekannt, die zur Eliminierung bestimmter Konfigurationen als Kandidaten verwendet werden können. Auch hierbei muss immer berücksichtigt werden, dass dies nur ohne die Gefahr falscher Verdikte geschieht, wenn das Auftreten solcher Konfigurationen nach Abweichungen ausgeschlossen werden kann oder die Konsequenzen als akzeptabel eingestuft werden. Beispielsweise, weil sichergestellt ist, dass neue Instanzen nicht nach einer Abweichung entstehen können oder wenn festgestellt wird, dass die nicht berücksichtigten Abweichungen nur zu zusätzlich gemeldeten Verstößen führen und dies tolerierbar ist oder vielleicht sogar mehr dem gewünschten Fehlermodell entspricht.

Prinzipiell bringt die Kombination von Instanzen und Kontexten eine Zunahme des Konfigurationsraums mit sich, der von Resumption auf Kandidaten hin untersucht werden muss. Allerdings gibt es auch Verwendungen, bei denen die genannte Kombination Resumption vereinfachen kann. Dies ist beispielsweise der Fall, wenn durch den Kontext entschieden werden kann, zu welcher Instanz eine Nachricht gehört. Ganz allgemein, wenn Ereignisse bestimmten Instanzen oder parallelen Automaten zugewiesen werden können, sie also keinen Einfluss auf die übrigen Teile haben, wird auch Resumption nur den jeweils betroffenen Teil verändern. Gilt dies für alle Ereignisse, kann Resumption für die Instanzen unabhängig behandelt werden. Wenn ausgeschlossen werden kann, dass eine Abweichung einer Instanz Einfluss auf die anderen Instanzen hat, oder solche Folgefehler explizit wie weiteres unerwartetes Verhalten berichtet werden sollen, kann Resumption auch nur für den betroffenen Automaten angewendet werden und die anderen parallelen Automaten behalten ihren Zustand.

5.6 Resumption für Automaten mit Zeitverhalten

Zeit-behaftete Automaten berücksichtigen neben dem Ablauf des Kontrollflusses auch dessen Zeitverhalten. Also neben der relativen Sequenz der Ereignisse auch, wie viel Zeit auf einer Uhr vergeht. Da ein Zeit-behafteter Automat über Kontexte nachgebildet werden kann, ist auch eine entsprechende Behandlung mit Resumption möglich. Bei Zeit-behafteten Automaten handelt es sich aber um einen Ansatz auf niedriger Ebene, der für tiefgehende theoretische Betrachtungen interessant ist. In der Praxis kann die Beschreibung von Zeitverhalten nur mit rücksetzbaren Uhren bereits schnell eine benutzbare Komplexität überschreiten. Es hat sich daher eine Abbildungen auf Sprachen mit mehr Struktur bewährt, um wiederkehrende Muster einfacher zu beschreiben.

Für die Modellierung wurden deshalb in Abschnitt 4.6.4 zwei Konzepte vorgestellt, um Zeitverhalten zu beschreiben. Zum einen gibt es Bedingungen der Wiederholungsrate und zum anderen Bedingungen für die Verzögerung. Für die Wiederholungsrate wird vorgeschlagen, sie über die Kompatibilitätsfunktion λ eines dedizierten Kontext zu erkennen. Ereignisse werden dem Kontext zugeordnet, wenn sie zu den vorgegebenen Kriterien für die Wiederholrate gehören. Der Algorithmus zur Überprüfung dieser kann für Resumption einfach durch Zurücksetzen seiner Parameter neu gestartet werden. Es sind auch weitere Varianten denkbar, die beispielsweise das entsprechende Intervall um einen gewissen Anteil vergrößern. Dies geht aber über den Rahmen dieser Arbeit hinaus.

Für die Verzögerung erlauben Zeitschranken an einem Zustand anzugeben, wie lange dieser aktiv sein darf. Hierbei handelt es sich um ein Ereignis, das nur indirekt beobachtet werden kann. Ist der Zustand länger als die festgelegte Zeitschranke aktiv, wird daher ein bestimmtes Ereignis ausgelöst. Im Folgenden wird gezeigt, wie Resumption auf einen Automaten mit Zeitschranken angewendet werden kann. Als Beispiel wird dazu der in Abbildung 5.8 abgebildete Automat betrachtet. Die Überschreitung des in der unteren Hälfte eines Zustands s_i angegebenen Maximums sup_i löst ein Ereignis t_i aus. Im Fall von s_1 und s_2 findet dann ein definierter Zustandsübergang zu s_0 statt, für s_0 ist kein Zustandsübergang definiert.

Durch die Übersetzung der Zeitschranke in ein Ereignis kann Resumption dies wie jedes andere Ereignis betrachten. Wird nun Resumption auf den Automaten angewendet, ergibt sich der in Abbildung 5.9a dargestellte Synchronisierungsbaum. Allerdings können die Ereignisse für Zeitschranken nur indirekt, also durch das Ausbleiben anderer Ereignisse, beobachtet werden. Zudem setzt dies Informationen über den aktuellen Zustand und wann dieser betreten wurde, voraus. Resumption wird aber genau dann benötigt, wenn es eine Ungewissheit über den aktuellen Zustand gibt. Dadurch müssen im Allgemeinen für jeden

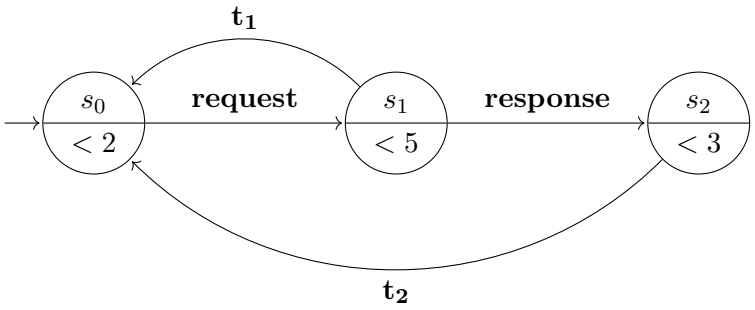


Abb. 5.8: Automat mit Zeitschranken für Zustände.

Zustand mit Zeitschranke auch Unterzustände, entsprechend der möglichen verbleibenden Restzeit, berücksichtigt werden.

Zur Notation einer Menge solcher Unterzustände eines Zustands wird im Folgenden der Operator $\clock : \mathbb{S} \mapsto \mathbb{S}_{sup}$ verwendet. $\mathbb{S}_{sup}$ ist die Menge aller Zustände mit Restzeit. Der Operator $\clock$ liefert für einen Zustand alle Zustände mit gültiger Restzeit zurück. Es sein angemerkt, dass dies unendlich viele sein können, da keine Einschränkung der Genauigkeit erfolgt. Der Operator kann durch ein Intervall beschränkt werden, um nur eine Teilmenge der Zustände mit entsprechenden Restzeiten zu liefern. Zudem wird erlaubt, dass das Intervall über die Länge der Zeitschranke selbst verlängert wird. Die Menge umfasst dann entsprechend auch Unterzustände die erst in der Zukunft betreten werden.

Der zweite Operator $\Delta : \mathbb{S}_{sup} \times \mathbb{R} \mapsto \mathbb{S}_{sup}$ (5.11) liefert die Menge von Unterzuständen, in die die Eingabezustände nach einem Zeitschritt $t \in \mathbb{R}$ unter Berücksichtigung der Zeitschranken übergehen. Da nicht zwingend erforderlich und zwecks Übersichtlichkeit in der Gleichung, wurde auf eine Reduktion der Intervalle auf den Bereich zwischen 0 und der Zeitschranke in Gleichung 5.11 verzichtet. Der Operator ist durch Vereinigung der Teilergebnisse auch für Mengen definiert (5.12).

$$\delta(q_1, t_1) \mapsto q_2 \Rightarrow \clock_{[a+c,b+c)} q_1 \Delta c = \clock_{[a,b)} q_1 \cup \clock_{\left[sup_2 - c, b-a+sup_2-c\right)} q_2 \tag{5.11}$$

$$\clock_{[a,b)} Q \Delta c = \bigcup_{q_i \in Q} \clock_{[a,b)} q_i \Delta c \tag{5.12}$$

Die Menge aller Zustände mit Restzeit ($\clock\mathbb{S}$) für das Beispiel aus Abbildung 5.8, ist die Vereinigung aller Unterzustände mit Restzeit für s_0, s_1 und s_2. Die Unterzustände können durch den Operator unter Angabe des Intervalls entsprechend der Zeitschranke angegeben werden, für s_0 beispielsweise durch $\clock_{[0,2)} s_0$. Der Zeitschritt für die weitere Untersuchung kann beliebig gewählt werden, die Grenzen des Intervalls bieten sich allerdings an, da nur hier eine Veränderung der Möglichkeiten stattfinden kann. Im Allgemeinen sollte auch die Untergrenze geprüft werden, da bereits dann Zustände hinzukommen können. Da von s_0

keine Transition für die Zeitschranke ausgeht, reduziert sich die Menge $\mathbb{S}_{[0,2)}s_0$ nach dem Zeitschritt $\Delta 2$ auf die leere Menge ($\emptyset$). Für s_1 und s_2 kommt hier jeweils s_0 hinzu. Um den Operator Δ auf eine Menge von Zuständen anzuwenden, wird er auf jedes Element einzeln angewendet (5.13). Bei einer Menge von Restzeit-Zuständen kann das Zeitschritt-Argument von den Intervallgrenzen abgezogen werden (5.11). Liegt ein Teil des Intervalls dann unter 0, wird dieser auf den Zustand übertragen, der durch die Transition erreicht wird und dessen Zeitschranke auf die Bereichsgrenzen addiert (5.14). Da es sich um Mengen handelt, können mehrfach auftretende Restzeit-Zustände entsprechend zusammengefasst werden (5.15). Durch wiederholtes Ausführen von Zeitschritten kann ein Synchronisierungsbaum wie in Abbildung 5.9b aufgebaut werden.

$$\mathbb{S}\Delta 2 = \mathbb{S}_{[0,2)}s_0\Delta 2 \cup \mathbb{S}_{[0,5)}s_1\Delta 2 \cup \mathbb{S}_{[0,3)}s_2\Delta 2 \tag{5.13}$$

$$= \emptyset \cup \mathbb{S}_{[0,3)}s_1 \cup \mathbb{S}_{[0,2)}s_0 \cup \mathbb{S}_{[0,1)}s_2 \cup \mathbb{S}_{[0,2)}s_0 \tag{5.14}$$

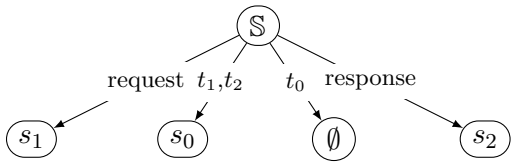
$$= \mathbb{S}_{[0,2)}s_0 \cup \mathbb{S}_{[0,3)}s_1 \cup \mathbb{S}_{[0,1)}s_2 \tag{5.15}$$

Theorem 8. *Gibt es keine gültige Reaktion auf das Ereignis einer Zeitschranke eines Zustands, dann ist für die Beobachtung von gültigem Verhalten seines bei 0 beginnenden Intervalls von Restzeiten äquivalent zur Beobachtung des Zustands mit der oberen Intervallgrenze als Zeitschranke. Insbesondere ist dies der Fall, wenn die obere Intervallgrenze identisch mit der Zeitschranke ist* (5.16).

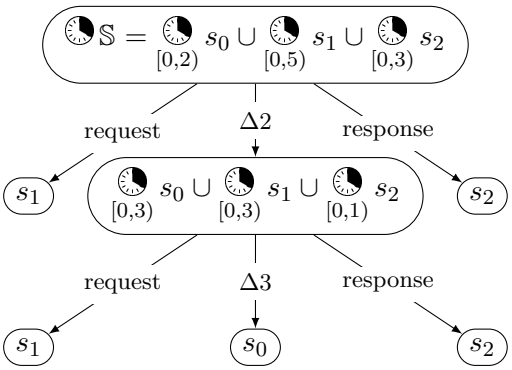
$$t_i \notin \mathbb{E}^{s_i} \Rightarrow \mathbb{S}s_i \equiv s_i \tag{5.16}$$

Beweis. Das gültige Verhalten in einem Zustand wird durch die ausgehenden Transitionen und die Zeitschranke definiert. Er kann nur zusätzliches Verhalten akzeptieren, wenn er weitere Transitionen oder eine längere Zeitschranke besitzt. Gäbe es einen Restzeit-Zustand in diesem Intervall, der anderes gültiges Verhalten zuließe als der Zustand mit Zeitschranke, müsste dieser zusätzliche Transitionen oder eine Zeitschranke außerhalb des Intervalls definieren. Beides wird durch die Konstruktion ausgeschlossen. Aus dem gleichen Grund kann der Zustand kein zusätzliches Verhalten definieren und die Zeitschranke entspricht durch die Konstruktion dem Supremum des Intervalls. □

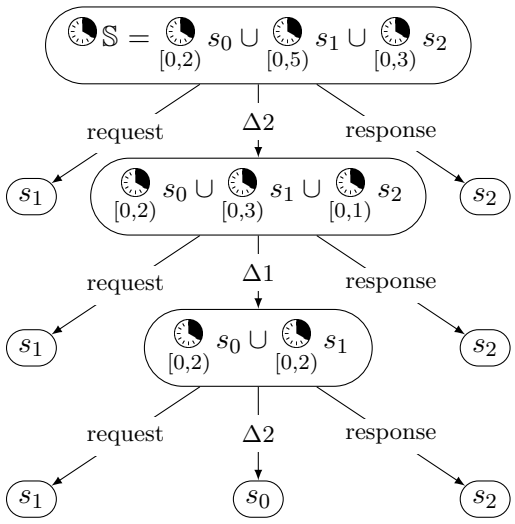
Durch Theorem 8 kann im Synchronisierungsbaum unten kurz s_0 geschrieben werden, da $\mathbb{S}_{[0,2)}s_0 \equiv s_0$. Es fällt weiter auf, dass sich das mögliche beobachtbare Verhalten des Systems in dem Beispiel nach den ersten Zeitschritten nicht verändert, die Transitionen sind äquivalent. In dem Zeitschritt $\Delta 1$ zwischen den beiden mittleren Knoten gehen die Restzeit-Zustände von s_2 alle in s_0 über. Effektiv erweitert sich dadurch die Zeitspanne,



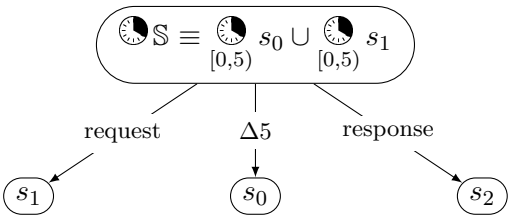
(a) Ohne Berücksichtigung von Restzeit-Zuständen.



(c) Zusammenfassung der mittleren Knoten.



(b) Knoten je ablaufender Zeitschranke.



(d) Minimal mit Restzeit-Zuständen.

Abb. 5.9: Synchronisierungsbaum für den Automaten aus Abbildung 5.8 mit Ereignissen für Zeitschranken ohne Berücksichtigung von Restzeit-Zuständen (a) und mit unterschiedlich zusammengefassten Knoten für die Restzeit-Zustände (b)-(d).

in der die Zeitschranke von s_0 erreicht werden kann. Von s_2 geht nur die Transition für das Ereignis der Zeitschranke aus. Da diese nicht direkt beobachtbar sind, ändert sich auch das beobachtbare gültige Verhalten bei Wegfall des Zustands nicht. Das Ereignis *request* führt zu s_1 und *response* zu s_2. Die beiden mittleren Knoten können deshalb, wie in Abbildung 5.9c, zusammengefasst werden. Mit der gleichen Überlegung kann dies mit dem obersten Knoten kombiniert werden.

In Abbildung 5.9d ist s_2 nicht erwähnt, da es keine Möglichkeit gibt, den Zustand zu erkennen, da das dort gültige Verhalten, *warten*, bereits durch die anderen Zustände abgedeckt wird. Da das Zusammenfassen der Knoten analog zur Minimierung von Automaten funktioniert, wird es hier nicht weiter verallgemeinert.

Theorem 9. *Wenn in einem Synchronisierungsbaum mit Zeitschranken alle gültigen Restzeit-Zustände als Wurzelknoten betrachtet werden, beginnen alle durch Zeitschritte erreichbaren Restzeit-Intervalle bei null.*

Beweis. Dies ergibt sich durch die Definition von Δ (5.11), die keine neuen Lücken entstehen lässt. Eine negative Untergrenze schließt Restzeit-Zustände aus der Vergangenheit in das Intervall ein und kann somit auf 0 für eine äquivalente Betrachtung der aktuellen Möglichkeiten angepasst werden. Da alle gültigen Restzeit-Zustände im Wurzelknoten enthalten sind und in einem Zustand keine Untergrenze festgelegt werden kann, müssen alle Intervalle dort bei null beginnen. Wenn für einen Zustand alle Restzeit-Zustände abgelaufen sind, dann liegt es daran, dass es keinen anderen Zustand mehr gibt, der zu diesem Zustand durch einen Zeitschritt übergeht. Er kann also auch nicht später wieder durch einen Zeitschritt betreten werden. □

Es sei angemerkt, dass wenn nicht alle Restzeit-Zustände im Wurzelknoten enthalten sind, sich die dort enthaltenen Lücken fortsetzen und es Intervalle geben wird, die nicht bei null beginnen. Die in Theorem 9 beschriebene Eigenschaft kann aber verwendet werden, um eine schnellere Konstruktion des in Abbildung 5.9d beschriebenen Synchronisierungsbaums zu erreichen. Ausgehend von $\mathbb{S}$ wird jedes Intervall um die Obergrenze des längsten Intervalls verlängert, das nach der Zeitschranke zu dem zugehörigen Zustand führt. Sollte es einen Zyklus in den Transitionen für Zeitschranken geben, sind diese Intervalle unendlich lang. Für jede Obergrenze in aufsteigender Reihenfolge wird nun überprüft, ob durch Wegfall des entsprechenden Zustands sich das erwartete Verhalten verändert. Ist dies der Fall, wird ein entsprechender Zeitschritt ausgeführt und das Ergebnis als neuer Knoten betrachtet. Der Algorithmus terminiert, wenn in dem aktuellen Knoten alle oberen Intervallgrenzen entweder null oder unendlich sind.

5.7 Fazit

In diesem Kapitel wurde untersucht, wie unerwartetes Verhalten und Abweichungen mit einem Monitor erkannt werden können. Dazu wurde der neue Ansatz zur Wiederaufnahme der Überprüfung, genannt *Resumption*, vorgestellt. Resumption ermöglicht einem Monitor mehr als nur den ersten Verstoß gegen die Spezifikation zu erkennen (siehe Herausforderung 5). Für einfache Automaten wurde gezeigt, dass mit Resumption alles unerwartete Verhalten effizient zur Laufzeit erkannt werden kann. Dazu wurde ein Verfahren zur automatisierten Erweiterung von Automaten mit Resumption vorgestellt. Die Verwendung von Resumption meldet genau dann unerwartetes Verhalten, wenn die Beobachtungen nicht auf den Automaten abgebildet werden können. Dadurch wird die Erkennung robust gegenüber unvollständigen Beschreibungen (siehe Herausforderung 6). Das nicht beschriebene Verhalten wird zwar als unerwartet markiert, der Monitor findet aber selbstständig heraus, wo die Beobachtungen wieder zum Spezifizierten passt und kann so auch danach unerwartetes Verhalten erkennen. Wird die unerwartete Beobachtung später als erwartetes Verhalten eingestuft, hilft die markierte Stelle im Trace, um die Spezifikation entsprechend zu ergänzen. Abweichungen, sofern diese nicht offensichtlich sind und mit unerwartetem Verhalten zusammenfallen, können selbst nicht beobachtet werden, da sie auf dem unbekannten internen Zustand des SuO beruhen. Für jede erkennbare Abweichung kann aber der Bereich des Traces abgegrenzt werden, in dem sie sich befindet. Nicht erkennbar sind Abweichungen, die exakt erwartetes Verhalten nachahmen – das SuO verhält sich entsprechend der Spezifikation, obwohl es intern eine Abweichung gab. Um solche Abweichungen zu erkennen, müsste die Beschreibung verfeinert werden. Umgekehrt sind alle Abweichungen erkennbar, die sich von erwartetem Verhalten unterscheiden. Mit Resumption kann also die Überprüfung fortgesetzt werden, nachdem eine Anomalie festgestellt wurde, um alle erkennbaren Abweichungen zu identifizieren. Der Ansatz ist effizient genug, dass dies für einfache Automaten zur Laufzeit erfolgen kann (Ziel 2) und damit eine Reaktion auf Abweichungen im Betrieb (siehe Herausforderung 7) möglich wird.

Das Verhalten von realen Systemen ist häufig komplexer, als nachvollziehbar mit einfachen Automaten beschrieben werden kann. Deshalb wird die Beschreibung häufig um Konzepte erweitert, wie die in Kapitel 4 Vorgestellten, mit denen bestimmte Verhaltensmuster nicht mehr explizit modelliert werden müssen. Die vorgestellten Konzepte sind auf einfache Automaten abbildbar. Entsprechend kann auch Resumption auf sie, wie auf einfache Automaten, angewendet werden. Dies kann allerdings zu einer (mehrfach) exponentiellen Zunahme an Zuständen führen. Allgemein und ohne Einschränkungen in den berücksichtigten Abweichungen kann diese Zustandsmenge nicht trivial reduziert werden. Daher wurden für die verschiedenen Erweiterungen Rahmenbedingungen betrachtet, unter denen Resumption vereinfacht werden kann.

6 Kapitel 6
Konzepte zur Anwendung des Ansatzes

Dieses Kapitel gibt einen Überblick über Anwendungskonzepte, um die Ansätze aus den vorangegangenen Kapiteln in einem Werkzeug zur Laufzeitüberprüfung zu integrieren. Die prototypische Umsetzung der Ansätze dient der Erfüllung von Ziel 3. Sie bildet die Grundlage für die Überprüfung der Praxistauglichkeit der Ansätze in ausgewählten Anwendungsfällen und zur Evaluierung von Resumption in Kapitel 7. Da viele Konzepte bereits in den vorangegangenen Kapiteln beschrieben wurden, geht dieses Kapitel nur auf die Besonderheiten bei der Implementierung ein. Die prototypische Umsetzung trug einen wesentlichen Bestandteil zum Aufbau der Werkzeugplattform DANA [DW17] bei. In Abbildung 6.1 ist ein Überblick über die Funktionalität der Werkzeugplattform gegeben. Im Folgenden wird nur auf die für diese Arbeit relevanten Teile eingegangen, die, sofern nicht anders angemerkt, vom Autor dieser Arbeit erweitert wurden, um zu den vorgestellten Konzepten zu passen.

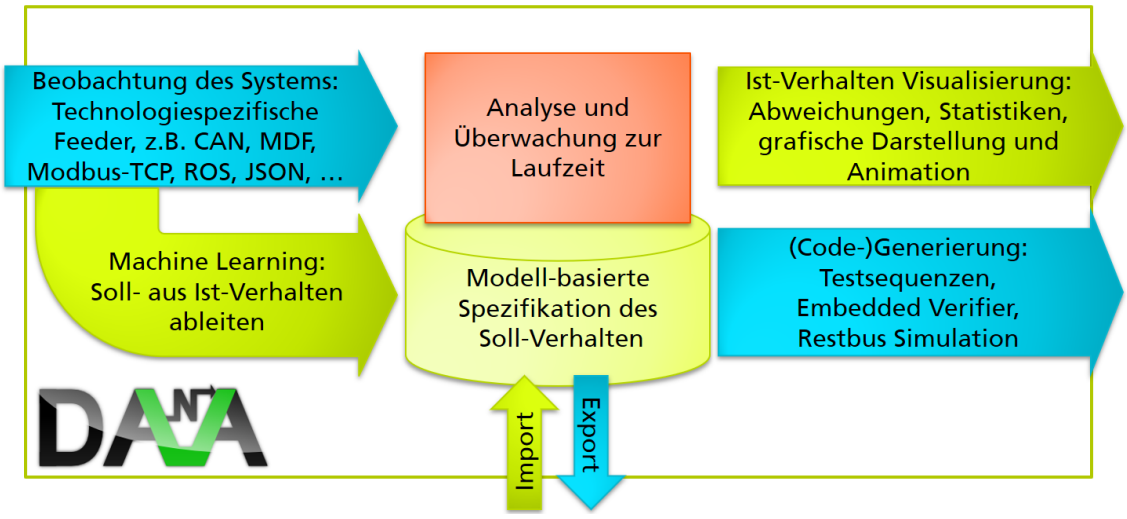


Abb. 6.1: Übersicht der Funktionalität der Werkzeugplattform DANA.

Die modell-basierte Spezifikation, also die Beschreibung des erwarteten Verhaltens bildet die Grundlage der Überwachung. Deshalb wird zunächst die verwendete Realisierung der Modellierung genauer beschrieben. Die Herausforderungen für die Umsetzung des Verifikationsmechanismus wurden bereits bei der Beschreibung von Ziel 3 aufgezählt. Eine Grundvoraussetzung für die Überprüfung ist, dass die notwendigen Daten eingelesen und interpretiert werden können. Um Daten aus möglichst vielen Quellen zu verarbeiten und auch die Integration von proprietären Format zu ermöglichen, ist der Aufwand zur Anbindung klein zuhalten. Dies ist beispielsweise der Fall, wenn aus dem proprietären Format nur die notwendigen Daten extrahiert werden müssen und die weitere Verarbeitung über einheitliche Schnittstelle erfolgt. Häufig benötigte Schritte der Analyse, beispielsweise das Auswerten der Daten, um ein Ereignis zuzuordnen, können nicht immer neu implementiert werden. Nach der Modellierung wird eine neue flexible Architektur vorgestellt, mit der die benötigte Modularität und Erweiterbarkeit der Datenverarbeitung im Prototypen erreicht wird. Aufbauend auf dieser Architektur wird anschließend die Ausführung der Überprüfung mit Resumption beschrieben. Der nächste Abschnitt befasst sich mit den Werkzeugen zur Auswertung, also wie die Ergebnisse der Überprüfung grafisch aufbereitet werden, um eine weitergehende Analyse und Interpretation durch den Benutzer des Werkzeugs zu ermöglichen. Während die vorangegangenen Schritte alle in einer Entwicklungsumgebung ablaufen, muss ebenfalls eine kompakte Variante der Überprüfung angeboten werden, um sie möglichst nahe, oder sogar in dem beobachteten System, ausführen zu können. Nur dann sind Reaktionen zur Laufzeit auf die Analyseergebnisse möglich. Dazu wird abschließend vorgestellt, wie eine kompakte Version des Zustandsautomaten in Form von C++-Code erzeugt werden kann.

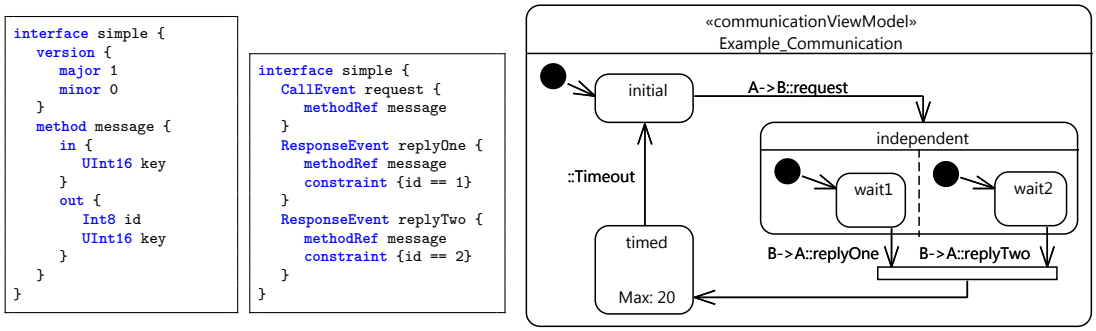
Für die maschinellen Lernverfahren sei auf die Master-Arbeit von Salvi [Sal17] verwiesen, die zwar vom Autor dieser Arbeit fachlich betreut wurde, aber über den Rahmen dieser Arbeit hinausgeht. Salvi hat eine Erweiterung der Plattform entwickelt, die Referenz-Modelle durch maschinelle Lernverfahren aus beobachtetem Ist-Verhalten erzeugt. Basis dafür waren die Ebenen der Topologie und Schnittstellenbeschreibung des in Kapitel 4 vorgestellten Modells, sowie Traces der Kommunikation. Er hat an Verfahren geforscht, mit denen die Schichten des Modells für Ereignisse und Verhaltensbeschreibung aus den Beobachtungen automatisch synthetisiert werden können. Die Hierarchie der Ereignisse ermöglicht auch eine schrittweise Verfeinerung, wenn weitere Trainingsdaten vorliegen. Sind (noch) nicht genügend Eingabe-Daten vorhanden, um ein vollständiges Verhaltensmodell zu lernen, kann es dank Resumption dennoch für eine Überprüfung herangezogen werden. Unerwartetes Verhalten, das sich nach genauerer (manueller) Prüfung als keine Abweichung herausstellt, kann dem Lernverfahren als zusätzliche Eingabe dienen und wird im angepassten Modell dann ebenfalls erwartet.

6.1 Realisierung der Modellierung

Dieser Abschnitt beschreibt die konkrete und abstrakte Syntax zu der in Kapitel 4 beschriebenen Modellierung, sowie die zur Bearbeitung bereitgestellten Werkzeuge. Wie bereits bei der Beschreibung der Semantik vorgestellt wurde, ist die Modellierung in mehrere Schichten eingeteilt. Die Schichten wurden mit verschiedenen Methoden zur Modellierung umgesetzt, sind aber alle in die Entwicklungsumgebung Eclipse [ECLIPSE] eingebunden. Ihre abstrakte Syntax wurde daher mit den Werkzeugen aus dem Eclipse Modelling Framework (EMF) beschrieben und geeignete Projekte für die Bereitstellung von Editoren wurden ausgewählt. Sie wurden bereits in anderen Arbeiten [Dra+13; DW17] vorgestellt.

Für die Schnittstellenbeschreibung konnte auf die existierende Beschreibungssprache *Franca IDL* [Franca] zurückgegriffen werden. Diese basiert ebenfalls auf EMF und bietet eine mit *XText* [XTEXT] erzeugte domänenspezifische Sprache (DSL, engl.: *Domain Specific Language*) samt Editor an. Während für die Überprüfung semantisch nur zwischen Client und Server einer Schnittstelle unterschieden wird, ist für eine generelle Schnittstellenbeschreibung auch die Art der Nachrichtenübermittlung, etwa eine Differenzierung nach Methodenaufrufen und passenden Antworten sowie nach Broadcasts oder Abfragen oder Manipulationen von Attributswerten nützlich, damit der korrekte Code für die verwendete Middleware erzeugt werden kann. Zusätzlich können spezifische Eigenschaften für eine bestimmte Middleware über sogenannte *Deployment*-Modelle erfasst werden. Abbildung 6.2a zeigt eine mit Franca IDL definierte Schnittstelle. Die Schnittstellenbeschreibungssprache bietet auch eine Möglichkeit zur Beschreibung von Automaten. Diese beziehen sich auf die jeweilige Schnittstelle und sind in ihren Möglichkeiten zur Beschreibung begrenzt. So sind beispielsweise die in Abschnitt 4.6 vorgestellten erweiterten Konzepte zur Verhaltensbeschreibung nicht vorgesehen.

Für die Ereignisbeschreibung wurde, ebenfalls mit XText, eine zweite DSL erschaffen, genannt *Event DSL*. Diese verfeinert die Elemente der in Franca IDL beschriebenen Schnittstellen, um eine Abbildung auf Ereignisse entsprechend Abschnitt 4.5.2 zu ermöglichen. Die Ereignisse sind dabei in Hierarchien organisiert und jedes Ereignis besteht aus einem Bezeichner und einer Bedingung. Die Hierarchie entspricht dabei der in Abschnitt 4.5.1 vorgestellten Relation $\triangleright$ zur Beschreibung von Ereignisspezialisierung. Daher gelten die Bedingungen eines allgemeineren Ereignisses immer auch für das speziellere. Die Wurzelereignisse haben normalerweise keine Bedingung oder nur solche Bedingungen, die immer von gültigen Nachrichten erfüllt werden müssen. Eine Bedingung kann aus einfachen arithmetischen und logischen Ausdrücken aufgebaut werden und auf die Parameter des Schnittstellenelements zurückgreifen. Ein Beispiel der Ereignisbeschreibung ist in Abbildung 6.2b dargestellt.



(a) Franca IDL. *(b)* Event DSL. *(c)* UML Profil für UML Zustandsdiagramme.

Abb. 6.2: Beispiele der zur Modellierung verwendeten Sprachen.

Für die Schichten der Modellierung, die eine grafische Darstellung und Bearbeitung vorsehen, wurde ein UML Profil erstellt. Damit kann Papyrus [Lan+09], ein grafischer Editor für UML in Eclipse [ECLIPSE], zur Bearbeitung verwendet werden. Das erstellte UML Profil schränkt die Vielfalt von UML ein, um eine Modellierung entsprechend Kapitel 4 zu erreichen. Die Modellierungskonzepte wurden gegenüber UML eingeschränkt und auf eingebetteten Code im Zustandsdiagramm wurde verzichtet, um die Verwendung potenziell mehrdeutiger Konzepte zu verhindern und eine nachvollziehbare Beschreibung des Verhaltens zu fördern. Dies erleichtert auch die prototypische Umsetzung von Resumption, da nur für die erlaubten Konzepte eine Übersetzung implementiert werden muss. Zudem ermöglichen die Stereotypen des Profils weitere Informationen in einem Modell zu hinterlegen. Das Verhalten, also die möglichen Sequenzen von Ereignissen einer Kommunikationsbeziehung, werden mit *UML Zustandsdiagrammen* [OMG-UML] spezifiziert. In UML wird an den Transitionen neben dem Auslöser auch eine Bedingung angegeben. In dem vorgeschlagenen Ansatz werden diese Bedingungen allerdings in der Ereignisbeschreibung und den Kontexten beschrieben. Durch das Profil kann die Transition den Bezug zu einem Ereignis abbilden. Ebenso können Sender und Empfänger des Ereignisses benannt werden. Abbildung 6.2c zeigt ein aus Papyrus exportiertes Verhaltensmodell mit den verfügbaren Elementen zur Modellierung. In der Beschreibung von hierarchischen Automaten in Abschnitt 2.2.3 werden Transitionen auf höherer Ebene bevorzugt, da dies eine einfachere Darstellung der Semantik erlaubt. Um dennoch Transitionen in verfeinernden Zuständen zu erlauben, müssen dafür explizite Ausnahmen definiert werden. In UML haben aber die Transitionen der Verfeinerungen Vorrang. Die Semantik des Profils ist genauso definiert. Diese Umkehr der Priorität ist durch Erzeugen der entsprechenden Ausnahmen trivial auf die Semantik abbildbar.

Kontexte wurden im Prototypen mit in die Event DSL integriert. Es können Variablen im Kontext deklariert und für jedes Ereignis Bedingungen und Aktionen definiert werden. Abbildung 6.3 zeigt ein Beispiel für einen Kontext mit der Variable *value*. Solange

```
Context CompareWithStoredValue {
   var UInt16 value
   constraint request {isset value == false} {value = key}
   constraint replyOne {value == key} {}
   constraint replyTwo {value == key} {}
}
```

Abb. 6.3: Beispiel für die Beschreibung eines Kontexts.

value nicht gesetzt wurde, akzeptiert der Kontext jede Anfrage, die zum Ereignis `request` passt. Der Kontext setzt dann *value* auf den *key* aus der zu `request` passenden Nachricht. Die Ereignisse `replyOne` und `replyTwo` werden nur von dem Kontext akzeptiert, wenn der Parameter *key* in der zugehörigen Nachricht mit dem gespeicherten Wert *value* übereinstimmt.

Zeit-Bedingungen für Verzögerungen werden, wie in Abschnitt 4.6.4 beschrieben, über Zeitschranken an Zuständen annotiert. Dies wird auch direkt im Diagramm angezeigt. Im Beispiel in Abbildung 6.2c ist für den Zustand *timed* eine maximale Aktivierungsdauer von 20 ms angegeben. Das bei Überschreiten dieser Zeitschranke ausgelöste Ereignis, bezeichnet mit *Timeout*, wird in dem Beispiel an der von diesem Zustand ausgehenden Transition verwendet. Über den Ausdruck `Periodic(name,lower,upper,jitter)` kann mit Kontexten auch auf die Einhaltung von Zeitfenstern für periodische Nachrichten geprüft werden.

6.2 Flexible Architektur zur Datenverarbeitung

Bei der Strukturierung der in Kapitel 4 vorgestellten Modellierung, wurde viel Wert auf eine Technologie-unabhängige Beschreibung des Verhaltens geachtet. Herausforderung 1 (siehe Abschnitt 1.2) motiviert dies durch eine Vielzahl an Middleware-Technologien und Hardware-Schnittstellen, die verwendet und regelmäßig ersetzt werden. Um die Interaktionen zu überprüfen, müssen die Daten aus den verschiedenen Technologien aber auch eingelesen werden können. Für die Realisierung wurde daher auch ein Ansatz für eine flexible Architektur zur Datenverarbeitung erarbeitet. Dies ist bei der Überprüfung notwendig, da je nach Anwendungsszenario andere Schritte zur Aufbereitung der Daten vorgenommen werden müssen. Um auch Daten in höchst individuellen Datenformaten einlesen zu können, ist eine einfache Möglichkeit zur Anbindung bestehender Bibliotheken vorzusehen. Es wurde deshalb ein einfaches, aber flexibel anpassbares Konzept für eine modulare Datenverarbeitungskette umgesetzt. Hierfür gibt es auch existierende Ansätze, wie z. B. die Pipelines und Prozessoren von BeepBeep [Hal16], die viele zusätzliche Funktionen bieten. Die folgende Beschreibung dokumentiert den verwendeten, vom Autor dieser Arbeit erstellten Ansatz. Er zielt auf eine minimale, leicht zu implementierende Schnittstelle und eine Integration der Beschreibung mit Franca IDL [Franca] ab.

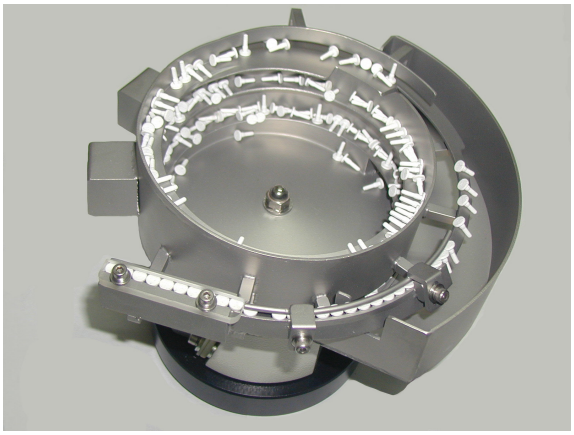


Abb. 6.4: Abbildung eines Vibrationswendelförderers (engl.: Bowl Feeder) [Feeder].

Als Inspiration für die Funktionsweise der Datenverarbeitung dienten Bausteine aus der Automatisierung, wie der Vibrationswendelförderer (engl.: *Bowl Feeder*, siehe Abbildung 6.4). Diese verwenden zur Sortierung und Orientierung der Teile verschiedene Standardmechanismen, die darauf aber individuell angepasst werden müssen. Nachfolgende Schritte werden durch die bekannte Orientierung einfacher und können diese bei Bedarf aus einem Speicher am Ende des Förderers entnehmen. Die Grundeinheit im digitalen ist daher ein Förderer (Feeder), der Nachrichten bzw. Ereignisse in aufbereiteter Form an einen Speicher (Storage) übergibt. Förderer können auch gleichzeitig Speicher für andere Förderer sein, wodurch sich Ketten bilden lassen. Die Schnittstelle *Feeder* dient dabei der Kontrolle des Moduls, also beispielsweise zum Starten und Stoppen, aber auch zum Zuweisen des Speichers. Dessen Schnittstelle Storage bietet nur eine Methode, die Objekte mit einem bestimmten Datentyp entgegennimmt. Durch die Verwendung von *Generics* [Bra04] wird bereits zum Zeitpunkt der Kompilierung sichergestellt, dass der produzierte mit dem entgegengenommenen Datentyp kompatibel ist. Die Schnittstellen sowie die Basis-Klassen haben, neben den Standard Java-Klassen, nur Abhängigkeiten zu den Klassen von Franca [Franca]. Diese sind notwendig, da mit Franca die in den Nachrichten enthaltenen Daten beschrieben werden.

Das folgende Beispiel veranschaulicht den typischen Aufbau einer Verarbeitungskette. Dabei werden auch ein paar der umgesetzten Realisierungen der Schnittstellen vorgestellt, die Klassennamen entsprechen den englischen Bezeichnungen.

Im ersten Schritt werden die Nachrichten von einem Förderer für Eingaben von dem gewünschten Medium gelesen und in ein einheitliches Format, die sog. SignalEventInstance, gebracht. Diese ermöglicht den Zugriff auf die Meta-Daten und Inhalte der Nachricht. Beispielsweise liest ein URIFeeder einen Datenstrom ein, der über eine URI [URI] erreichbar ist. Sind die darin enthaltenen Nachrichten in der *JavaScript Object Notation* [JSON], kurz JSON, beschrieben, können sie von einem JSONFeeder in

SignalEventInstances übersetzt werden. Es werden auch verschiedene Implementierungen angeboten, um Datenströme aus mehreren Quellen zusammenzubringen. Durch bereitgestellte abstrakte Klassen können Feeder für neue Eingabeformate einfach erstellt werden. Beispielsweise müssen für eine Spezialisierung von ThreadedFeeder nur Methoden zum vorbereiten (engl.: *prepare*), fördern (engl.: *feed*) und aufräumen (engl.: *cleanup*) implementiert werden. Dabei soll die Methode zum Fördern die nächste Nachricht einlesen bzw. erzeugen und zurückgeben. Durch die abstrakte Klasse wird sie in einem eigenen Thread wiederholt aufgerufen und kann somit auch blockieren, wenn sie beispielsweise auf externe Daten oder einen API-Aufruf warten muss, was sich dort als übliches Muster herausgestellt hat.

Im zweiten Schritt identifiziert der Ereignisabbildungsförderer (EventMappingFeeder) basierend auf der SignalEventInstance das zur Nachricht passende Element einer Franca Schnittstelle und ein Ereignis (vgl. Abschnitt 4.5.2), sofern dies nicht bereits beim Einlesen der Nachricht geschehen ist. Hier werden ebenfalls die kompatiblen Kontexte identifiziert (vgl. Abschnitt 4.6.2). Anschließend werden Ereignisse, die nicht für das beobachtete Szenario relevant sind, durch einen SignalEventFilter herausgefiltert. Als nächstes injiziert der SystemEventFeeder System-Ereignisse in den Strom aus Nachrichten. Dies sind spezielle Ereignisse für die Initialisierung und das Ende der Eingaben, aber auch Ereignisse für das Überschreiten von Zeitschranken im Automaten. Er bietet dazu eine Schnittstelle über die andere Komponenten System-Ereignisse registrieren und verwerfen können. Hierbei handelt es sich letztendlich um eine spezielle Umsetzung von Kontexten für die Überprüfung der Zeitschranken, entsprechend des in Abschnitt 4.6.4 beschrieben Prinzips.

Der EventExecutionFeeder gibt die Ereignisse sequenziell an einen EventExecutor und erhält jeweils eine Beobachtung (Observation). Dieser Schritt der Ausführung der Überprüfung wird in Abschnitt 6.4 genauer betrachtet. Die Beobachtung wird an die SignalEventInstance angehängt. Ebenso werden in diesem Schritt durch den ActionRunnerFeeder die Aktionen zur Aktualisierung von Kontexten ausgeführt. Die Ergebnisse werden in einem Puffer-Speicher (BufferStorage) abgelegt, aus dem sie angezeigt oder weiter verarbeitet werden können. Er bietet dazu einen blockierenden Iterator (BlockingIterator) an, der auf neue Ereignisse wartet und damit als Brücke zwischen den Schnittstellen Feeder und Iterable dient. Der Ablauf ist in Abbildung 6.5 zusammengefasst. Die Mechanismen zur Konfiguration einer solchen Verarbeitungskette werden im nächsten Abschnitt vorgestellt.

Abb. 6.5: Typische Sequenz der Verarbeitung von Nachrichten.

6.3 Konfiguration der Überprüfung

Eine flexible Architektur ist nur dann hilfreich, wenn auch ihre Konfiguration leicht an die vorliegenden Bedarfe angepasst werden kann. Dieser Abschnitt stellt deshalb die Konfiguration der Überprüfung vor. Die Konfiguration ist selbst modular aufgebaut. Prinzipiell ist ein Feeder eine gewöhnliche Java-Klasse und kann als solche programmatisch instanziiert werden. Damit kann die komplette Kette der Verarbeitung, wie sie im vorangegangenen Abschnitt beschrieben wurde, ganz individuell zusammengebaut werden.

Im Allgemeinen werden zur Konfiguration häufig ähnliche Kombinationen von bestimmten *Feeder*-Instanzen zusammengebaut. Ein Block solcher Feeder wird in einer Ausführungs-Erweiterung (LaunchExtension) zusammengefasst. Diese Erweiterungen führen dann einen koordinierten Aufbau der Verarbeitungskette durch, indem in mehreren Phasen Informationen in einem zentralen Start-Kontext (LaunchContext) hinterlegt und neue Feeder erstellt und in die Kette integriert werden. Eine solche Erweiterung (UMLModel-LaunchExtension) dient beispielsweise dem Laden des Modells, während eine andere (SCXMLLaunchExtension) alle notwendigen *Feeder* registriert, um eine Ausführung des Modells zur Überprüfung zu ermöglichen, wie im nächsten Abschnitt beschrieben wird. Eine dritte Erweiterung (DebugTargetLaunchExtension) bereitet die Anzeige der Ergebnisse in der Eclipse IDE vor.

Der Förderer für Eingaben ist hingegen häufig sehr individuell zusammengesetzt. Um hier einen gewissen Grad an Automatisierung zu ermöglichen, wurde eine Fabrik implementiert (FeederFactory), die basierend auf Abbildungen von Schlüsselworten auf Konfigurationsparameter einen Feeder oder eine Kette erzeugt. Um diese Abbildungen ohne Code zu beschreiben, wurden zwei verschiedene Erweiterungen umgesetzt: eine liest sie aus einer speziellen DSL aus, die andere aus der in Eclipse hinterlegten Konfiguration. Die meisten der Erweiterungsmodule haben ebenfalls ein solches Konfigurationsmodul, eine ConfigExtension. Diese Konfigurationsmodule sind in den Ausführungs-Mechanismus von Eclipse [EclDbg] integriert und erlauben dem Benutzer über eine individuell implementierte grafische Benutzerschnittstelle (GUI, engl.: *Graphical User Interface*) die Parameter der Erweiterung anzugeben. Abbildung 6.6 zeigt die GUI mit dem integrierten Editor für die DSL zur Konfiguration des Förderers für Eingaben. Durch Schaltflächen können einfach die verschiedenen Erweiterungen aktiviert und deaktiviert werden.

Nur die Elemente zur Konfiguration aktiver Erweiterungen werden angezeigt. Somit ist auch die GUI selbst einfach und schnell an die aktuellen Bedarfe anpassbar und man kann die flexible Architektur übersichtlich bearbeiten. Wenn auch nicht im Rahmen dieser Arbeit genutzt, wurde unter anderem bereits eine Erweiterung implementiert, mit der die Plattform

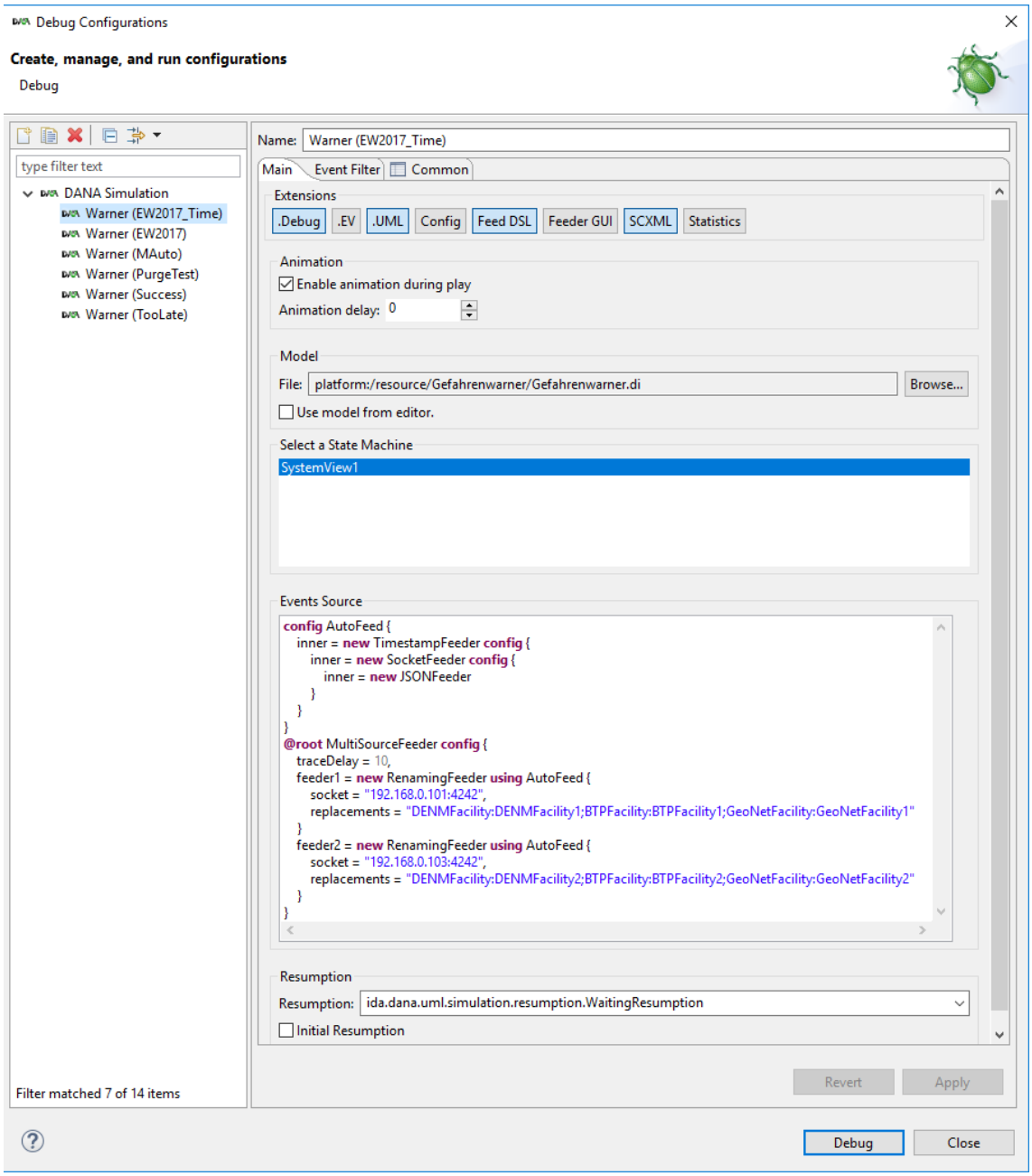


Abb. 6.6: Dialog zur Konfiguration der Ausführung integriert in den Debug-Mechanismus von Eclipse.

auch Zustandsautomaten erlernen kann [Sal17]. Durch das hier vorgestellte modulare Konzept kann leicht zwischen Lernen und Überprüfen gewechselt werden. Identische Teile, beispielsweise die Konfiguration des Förderers für Eingaben, bleiben unverändert, während das Umschalten von Lernen auf Überprüfen durch Wechseln der aktiven Erweiterungen erfolgt. Um solch ein schnelles Umschalten zu fördern, bleibt auch die Konfiguration von nicht aktiven Erweiterungen in der Run-Konfiguration erhalten. In dem Beispiel sind neben der Erweiterung zur Konfiguration des Förderers für Eingaben (*Feed DSL*) auch die Erweiterungen zur Integration in das Debug-Framework von Eclipse (*.Debug*), zum Laden eines UML Modells (*.UML*) und zur Ausführung mit *SCXML* aktiv. Bei letzterer kann auch angegeben werden, welche Strategie für Resumption verwendet werden soll. Die Ausführung wird im nächsten Abschnitt genauer betrachtet und Abschnitt 6.5 erläutert die Integration in das Debug-Framework von Eclipse.

6.4 Ausführung der Überprüfung

Dieser Abschnitt gibt einen Überblick darüber, wie die Ausführung der Überprüfung in der Werkzeugplattform umgesetzt wurde. Dazu wird die Ausführung mit der Erweiterung *SCXML* vorgestellt. Dieser Name wurde gewählt, weil *Commons SCXML* [ApaSCX] (Version 0.9) zur Ausführung des Zustandsautomaten verwendet wird. Die Wahl fiel auf SCXML [SCXML15], da die Sprache eine gewisse Ähnlichkeit zu den verwendeten UML Zustandsautomaten hat und bereits durch verschiedene Bibliotheken auf unterschiedlichen Plattformen realisiert wurde, was mögliche Portierungen vereinfachen kann. Zudem war zum Zeitpunkt der Entscheidung für SCXML noch nicht die genaue Ausführungssemantik für UML Zustandsautomaten [OMG-PSSM] festgelegt. Es gab daher auch keine Bibliotheken mit standardisiertem Verhalten. Im Folgenden wird zunächst beschrieben, wie die Verhaltensbeschreibung nach SCXML übersetzt wird, bevor auf die Integration von Instanzen, Zeitschranken und Resumption eingegangen wird. Abschließend wird die Integration der Ausführung in die flexible Architektur erläutert.

Übersetzung nach SCXML Viele Modellierungselemente und Konzepte aus dem Verhaltensmodell können direkt auf SCXML abgebildet werden: Initiale Knoten, Transitionen und Zustände mit Regionen für Verschachtlung sowie paralleles Verhalten. Durch die Verwendung von voll-qualifizierten Namen für die Ereignisse können die Transitionen mit regulären Ausdrücken auch spezialisierte Ereignisse erkennen. Benötigte Kontexte, dazu gehören auch die für Sender und Empfänger, werden in der Bedingung jeder Transition geprüft.

Die Übersetzung von *Join*-Elementen erfordert einen Zwischenschritt. Das *Join*-Element für den koordinierten Austritt aus parallelen Regionen wurde in der Semantik für hierarchische Automaten (vgl. Abschnitt 2.2.3) nicht definiert. Dies liegt daran, dass es sich hierbei lediglich um eine kompaktere Syntax handelt. Nach der für das Profil vereinbarten Semantik [DAN12] darf die ausgehende Transition vom Join erst verwendet werden, wenn auch alle eingehenden Transitionen aktiviert wurden. Dies kann durch zusätzliche Zustände emuliert werden, sowie durch eine Transition, die nur aktiv ist, wenn auch alle der zusätzlichen Zustände aktiv sind. SCXML enthält dieses Element ebenfalls nicht, es kann dort aber auch auf diese Weise nachgebildet werden.

Integration von Instanzen, Zeitschranken und Resumption in SCXML Instanzen wurden in SCXML integriert, indem bei Betreten eines entsprechend markierten Zustands dessen Inhalt in einen neuen parallel Zustand unter dem Wurzelzustand kopiert und aktiviert wird. Dies wurde so gewählt, da Commons SCXML Veränderungen am Modell zur Laufzeit toleriert, solange die Konfiguration gültig bleibt. Eine Änderung der Konfiguration, damit ein Zustand auch mehrfach aktiv sein kann, hätte tiefer greifende Veränderungen vorausgesetzt. Die Transformation (ScxmlTransformation) erzeugt dazu zwei zusätzliche Zustände in dem markierten Zustand. Der eine enthält die Vorlage für die zu kopierenden Zustände und der andere ist leer. Der leere Zustand wird als initialer Unterzustand markiert und verhindert so, dass die Vorlage aktiv wird. Die Aktion zur Erzeugung der Instanz (InstanceCreationAction) wird im markierten Zustand hinterlegt und bei Betreten ausgeführt. Neben dem Kopieren der Zustände übersetzt sie auch Referenzen zu Kontexten, die mit dem markierten Zustand assoziiert sind, mit Varianten spezifisch für die Instanz. Kontexte werden über weitere Aktionen (Register- und UnregisterContextAction) bei Betreten und Verlassen entsprechender Zustände registriert und abgemeldet. So müssen nur aktuell relevante Kontexte evaluiert werden. Zeitschranken melden sich bei einem TimeoutHandler auf die gleiche Weise an und ab (Register- und UnregisterTimeoutAction), damit dieser sie bei Bedarf in die Sequenz der Ereignisse injizieren kann. In der Kette vorgestellt in Abschnitt 6.2 handelt es sich dabei um den SystemEventFeeder.

Resumption und die implizite Fehlersemantik wurde durch eine Spezialisierung der Klasse SCXMLSemantics integriert. Diese definiert Teilfunktionen jedes Ausführungsschrittes des SCXML-Automaten. Sie listet mögliche Transitionen auf, wählt daraus die Aktiven, folgt diesen und führt Aktionen aus. Die Spezialisierung überprüft vor dem Folgen von Transitionen, ob mindestens eine Transition gefunden wurde. Sofern Resumption nicht bereits aktiv ist, meldet sie den Fehler und stößt Resumption an, um eine neue Konfiguration zu finden. Nach den Aktionen wird sichergestellt, dass im letzten Schritt hinzugefügte Instanzen in der Konfiguration aktiviert und entfernte Instanzen abgemeldet werden.

Integration in die flexible Architektur Die Übersetzung zwischen dem ursprünglichen Automaten wird in einer SCXMLMap festgehalten. Diese Abbildung verlinkt die in Beziehung stehenden Elemente der beiden Modelle sowie deren Bezeichner miteinander. Damit ist es möglich die Beobachtungen am SCXML-Automaten zurück auf den ursprünglichen Automaten abzubilden. Die Übersetzung wird nur einmal in der Vorbereitung der Ausführung durchgeführt. Da das Schema für die Umbenennung von Kopien für Instanzen bekannt ist, können diese auch den ursprünglichen Elementen zugeordnet werden. Diese Abbildung ermöglicht anderen Komponenten, beispielsweise der Visualisierung, eine Interpretation der Ergebnisse auf Basis des UML Modells.

Durch einen EventExecutionFeeder, der einen SCXMLEventExecutor erhält, wird die Ausführung in eine Verarbeitungskette der flexiblen Architektur eingebunden. Der SCXMLEventExecutor gibt jede eingehende Nachricht an das, wie oben beschrieben, angepasste Common SCXML weiter und übersetzt sie dabei in eine SCXMLObservation. Jede Beobachtung enthält ein Verdikt und die Veränderungen an der Konfiguration des SCXML-Automaten. Ein SystemEventFeeder fügt bei Bedarf Ereignisse für Zeitschranken ein und wird dazu als _TimeoutHandler im SCXMLExecutor hinterlegt. Wie alle anderen Auswertungsergebnisse, wird die Beobachtung an das SignalEventInstance der Nachricht angehängt, damit die im nächsten Abschnitt beschriebene Darstellung Zugriff auf alle Auswertungen hat. Die Zusammenstellung und Konfiguration der verschiedenen Feeder und anderen Klassen, für die Ausführung der Überprüfung mit SCXML, kann über die Klasse SCXMLLaunchExtension automatisiert werden.

6.5 Darstellung der Auswertung

Dieser Abschnitt beschreibt einige der Mechanismen, die umgesetzt wurden, um die Ergebnisse der Auswertung in der IDE aufzubereiten und darzustellen. Hierbei wird der Fokus auf die interaktiven Möglichkeiten zur Auswertung gelegt. Sie sind in den Debug-Mechanismus von Eclipse [EclDbg] integriert. In Abbildung 6.7 ist die Eclipse IDE dargestellt, wie sie typischerweise zur interaktiven Auswertung konfiguriert ist. Diese Konfiguration des Fensters wird von Dana als *Verification*-Perspektive angeboten. Perspektiven sind ein Mechanismus zur Verwaltung und zum Wechsel zwischen verschiedenen Konfigurationen der Ansichten des Eclipse-Fensters.

In dem Fenster ist oben links ① der UML Editor zu sehen, in dem die aktiven Zustände während der Ausführung farblich markiert werden. Diese Animation läuft asynchron zur Auswertung des Traces und ist verlangsamt, damit der Ablauf beobachtet werden kann. In dem Automaten sind ebenfalls Markierungen in Form von roten Kreisen mit einem weißen 'X' zu sehen. Diese werden an den aktiven Zuständen annotiert, wenn eine Abweichung

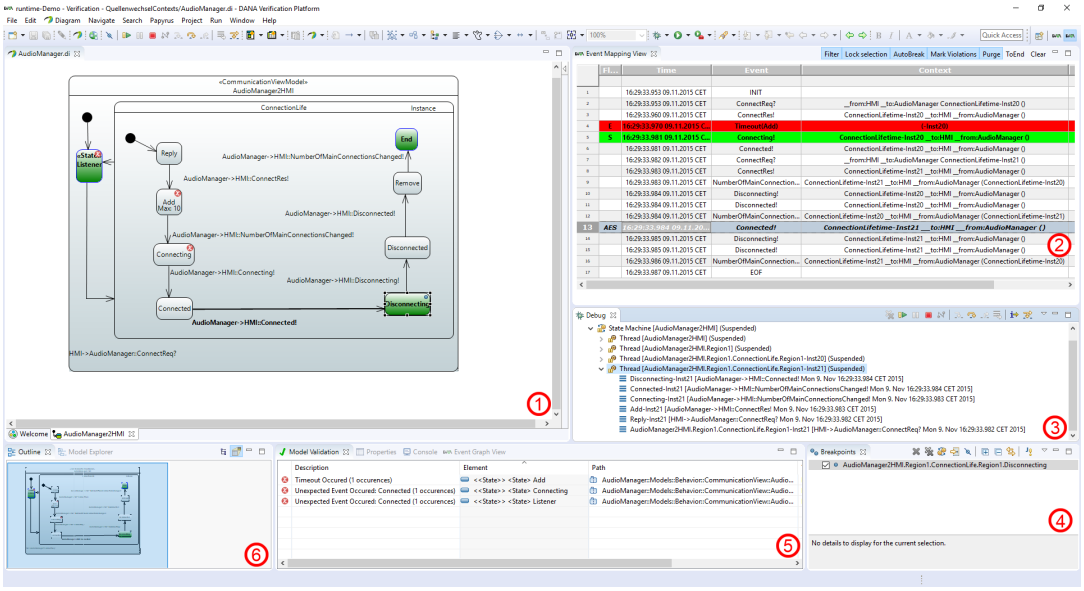


Abb. 6.7: Eclipse IDE während der Ausführung der Überprüfung. Ansichten, von links oben im Uhrzeigersinn: Editor mit Animation der aktiven Zustände im Automaten ①, Liste der verarbeiteten Nachrichten ②, Sortierung der Ereignisse nach betroffener Region und Instanz ③, Übersicht der Breakpoints ④, Liste erkannter Abweichungen ⑤ und Übersicht des Modells ⑥.

erkannt wurde. Wenn die Maus über eine solche Markierung bewegt wird, zeigt ein Tooltip die Beschreibung der Abweichung an. Um den Editor ordnen sich weitere Ansichten an, die andere Details zur Auswertung liefern.

Oben rechts ② ist die Ansicht der Ereignis-Abbildungen (*Event Mapping View*). Diese Tabelle gibt zu jedem Element des Traces in sequenzieller Reihenfolge Auskunft. In der ersten Spalte stehen Kennzeichen für Abweichung ('E'), Synchronisierung ('S') und die aktuelle Position der Animation ('A'). Abweichung und Synchonisierung werden zusätzlich durch Einfärben der gesamten Zeile hervorgehoben. Danach folgen ein Zeitstempel und der Name des Ereignisses, auf den die Nachricht abgebildet wurde. In der Spalte *Context* sind die getroffenen Kontexte aufgezählt. Nicht getroffene, aber kompatible Kontexte sind in Klammer genannt. Die Tabelle erlaubt ein Filtern nach Begriffen. So können beispielsweise alle Ereignisse, die mit einem bestimmten Kontext kompatibel sind, durch Angabe des Namens gefunden werden. Die Schaltflächen in der Kopfzeile der Ansicht steuern die Interaktion mit der Animation. Beispielsweise kann die Auswahl automatisch auf das animierte Ereignis gesetzt, die Animation bei einer Abweichung unterbrochen, und das automatische Markieren von Abweichungen im Automaten ein- und ausgeschaltet werden. Die Ansicht erzwingt, dass der analysierte Trace im Speicher verbleibt. Damit auch der Umgang mit längeren Traces möglich ist, verfügt sie dafür über eine Funktion zur automatischen Säuberung (*Purge*), die ältere Ereignisse automatisch verwirft. Über das Kontext-Menü kann die Animation zu einer beliebigen Zeile springen, das Ende ist auch direkt über die Schaltfläche *toEnd* erreichbar.

Die *Debug*-Ansicht ③ ist Teil des Debug-Mechanismus von Eclipse und zeigt normalerweise die Struktur eines untersuchten Prozesses und seine Threads an. Bei der Überprüfung mit DANA wird jede aktive Region des Automaten als eigener Thread angegeben. Ist eine Region einem Zustand mit Instanzen untergeordnet, wird zur Unterscheidung der Name der Instanz an den Namen der Region angehängt. Wenn die Animation pausiert ist, wird unter jeder Region die Historie ihrer aktiven Zustände angezeigt, zusammen mit dem Ereignis, durch den der Zustand betreten wurde. Dies erlaubt eine schnelle Übersicht über die Abläufe in parallelen Regionen und Instanzen. Ebenfalls vom Eclipse Debug-Mechanismus stammen die Schaltflächen zum Weiterführen, Pausieren oder Beenden der Debug-Sitzung. In DANA wirkt sich dies, bis auf das Beenden, nur auf die Animation aus. Die Analyse läuft im Hintergrund weiter, da das System unter Beobachtung (SuO, engl.: *System under Observation*) ebenfalls nicht angehalten wird.

Die untere Zeile der Ansichten enthält eine Übersicht (engl.: *outline*) ⑥ des Modells zur leichteren Navigation in größeren Zustandsdiagrammen, eine Liste der gefundenen Abweichungen integriert in die Ansicht zur Validierung von Modellen aus Papyrus (engl.: *model validation*) ⑤ und eine Aufzählung der Haltepunkte (engl.: *breakpoints*) ④. Die Haltepunkte können an Zuständen gesetzt werden, um die Animation bei Betreten anzuhalten.

Details zu einer Nachricht sind in der Ansicht für Eigenschaften (engl.: *properties*, siehe Abbildung 6.8) sichtbar, wenn die entsprechende Zeile in der Tabelle ausgewählt wurde. Dort wird der komplette Inhalt der zugehörigen SIGNALEVENTINSTANCE angezeigt. Neben den Parametern und Meta-Daten der Nachrichten können dort auch die während der Auswertung gesammelten Anhänge betrachtet werden, beispielsweise auch die SCXMLOBSERVATION mit den genauen Veränderungen der Konfiguration des SCXML-Automaten, sowie die getroffenen Kontexte und die in den Kontexten hinterlegten Informationen.

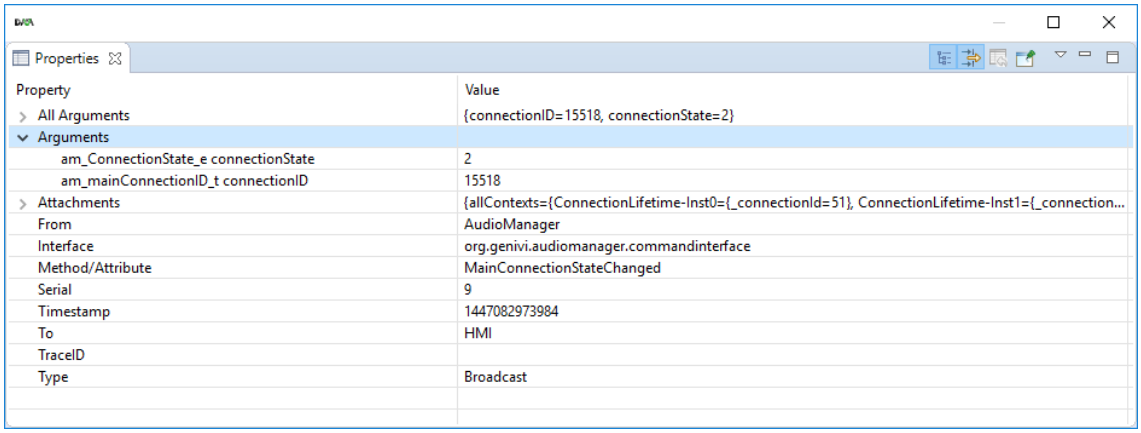


Abb. 6.8: Ansicht der Eigenschaften einer SIGNALEVENTINSTANCE mit ihren Meta-Daten und Parametern einer Nachricht sowie die angehängten Auswertungen.

Diese Ansichten wurden in verschiedenen Modulen für die Verarbeitungskette und durch Anbindung an den Debug-Mechanismus von Eclipse in der Erweiterung *.Debug* zusammengefasst. Da beide Konzepte modular aufgebaut sind, können weitere Werkzeuge zur Analyse leicht integriert werden. Beispielsweise wurde eine Ansicht umgesetzt, mit der Signalverläufe aus Nachrichten extrahiert und grafisch aufbereitet werden können. Diese ist in Abbildung 6.9 dargestellt. Eine weitere Erweiterung wird im Rahmen der Evaluation in Abschnitt 7.1.5 demonstriert, indem Statistiken über die Verweildauer in einem Zustand gewonnen und im Editor angezeigt werden.

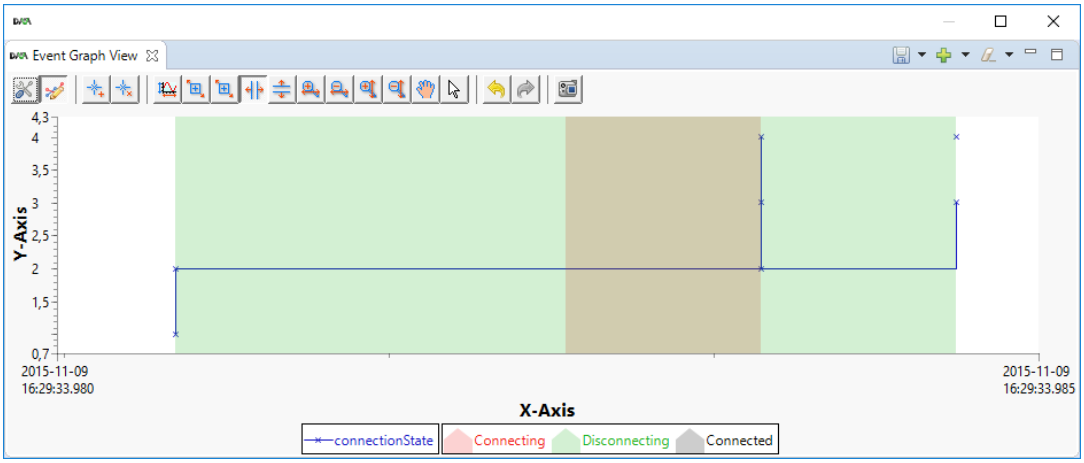


Abb. 6.9: Ansicht zur grafischen Darstellung von Signalverläufen (Linien) und aktiven Zuständen (Flächen). Die Darstellung kann konfiguriert werden.

6.6 Embedded Verifier

Während die in den vorangegangenen Abschnitten vorgestellten Methoden hilfreich sind, um interaktiv einen Trace zu erkunden, ergeben sich andere Anforderungen für eine automatisierte Analyse in oder nahe des eingebetteten Zielsystems. Hier steht ein effizienter Umgang mit den verfügbaren Ressourcen im Vordergrund. Deshalb wurde eine Übersetzung des UML Zustandsautomaten in kompakten C++-Code realisiert, ein Eingebetteter Überprüfer (EV, engl.: *Embedded Verifier*).

Der EV überprüft nur die zeitliche Reihenfolge der Ereignisse, also ob ein Ereignis bei Auftreten erwartet wurde oder ein erwartetes Ereignis fehlt. Die Identifikation der Ereignisse (und kompatibler Kontexte) wird separat gehandhabt. Während der Code-Generator auch dedizierte Routinen aus den Bedingungen der Ereignisse erzeugen kann, wurde im Projekt AUTOTRACE [Fra17] ein anderer Ansatz gewählt.

Zur Aufzeichnung und Instrumentierung werden dort sog. *Trace Manager* verwendet. Diese erhalten eine Konfigurationsdatei mit Regeln, die bestimmen, welche Daten zu

welchen Aktionen führen sollen. Entsprechend erzeugte Konfigurationsdateien führen dazu, dass die Trace Manager den EV über die Ereignisse informieren. Dies hat den Vorteil, dass weniger Daten von der aufzeichnenden Komponente zum EV übertragen werden müssen.

Die Implementierung ist zweigeteilt. Sie besteht zum einen aus einem Satz an statisch implementierten Klassen, mit denen die Semantik des Zustandsautomaten umgesetzt ist. Zur Erhöhung der Kompatibilität wurde dabei auf die Verwendung externer Abhängigkeiten und dynamische Allokation von Speicher verzichtet. Die Konfiguration eines hierarchischen Automaten mit Instanzen wächst dynamisch mit jeder neuen Instanz, da dann ein weiterer aktiver Zustand hinzukommt. Um auch den Speicher für sie statisch zu allozieren, muss die maximale Anzahl an Instanzen und damit die der gleichzeitig aktiven Zustände begrenzt werden.

Die eigentliche Struktur des Zustandsautomaten wird bei der Initialisierung durch den Wurzelzustand übergeben, der mit allen weiteren Zuständen verbunden ist. Dieser zweite Teil der Implementierung wird aus dem UML Zustandsdiagramm erzeugt und dabei um Resumption erweitert. Dazu wird der in Abschnitt 5.3 vorgestellte Algorithmus verwendet, um einen möglichst kompakten hierarchischen Automaten zu generieren. Der generierte Code deklariert und initialisiert die Zustände sowie Transitionen des Automaten und legt den Wurzelzustand unter dem externen Symbol *entry* (zu deutsch Eintritt) ab.

Auch wenn es aktuell nicht im Prototypen umgesetzt ist, erlaubt diese Zweiteilung die Struktur des Automaten dynamisch zu laden und sogar diese zur Laufzeit zu verändern. Dabei muss allerdings eine konsistente Konfiguration des EV erhalten bleiben bzw. durch geeignete Maßnahmen wiederhergestellt werden. Da Resumption aktuell für den EV statisch vorberechnet wird, sollten die Veränderungen auch die Erweiterung mit Resumption aktualisieren.

Die Generierung eines EV kann in der Eclipse IDE über das Kontext-Menü des UML Zustandsautomaten angestoßen werden. Daraufhin wird ein C++-Projekt angelegt und das statische Template dorthin kopiert. Ebenfalls wird dort der Code geschrieben, der die Struktur des Automaten beschreibt, sowie die Deklaration der Ereignisse. In dem Projekt werden ebenfalls Regeln für die Trace Manager abgelegt, sofern die dafür notwendigen Informationen zur Adressierung mit einem Franca Deployment angegeben sind.

6.7 Fazit

In diesem Kapitel wurde ein Überblick über die in Ziel 3 geforderte Realisierbarkeit des in dieser Arbeit vorgestellten Ansatzes und ihre Integration in die Werkzeugplattform Dana gegeben. Dazu wurde gezeigt, wie die konkrete und abstrakte Syntax der in Kapitel 4 beschriebenen Modellierung durch Kombination verschiedener Modellierungssprachen, umgesetzt wurde. Durch die Verwendung von Franca IDL [Franca] zur Beschreibung der statischen Schnittstellen wird die in Herausforderung 1 geforderte Technologieunabhängigkeit umgesetzt. Die dadurch notwendige Anpassbarkeit wird durch die flexible Architektur zur Datenverarbeitung und die Konfiguration der Überprüfung ermöglicht. Für die Ausführung der Überprüfung stehen zwei Alternativen zur Verfügung: Durch Interpretation des Modells mit SCXML oder durch Generierung von ausführbarem C++Code. Letzteres ermöglicht die in Herausforderung 7 geforderte Reaktion auf Abweichungen im Betrieb. Um die Auswertung von Beobachtungen zu erleichtern, wurde der Debug-Mechanismus von Eclipse [EclDbg] erweitert und dabei Möglichkeiten zur Animation der Zustandsautomaten und zur grafischen Darstellungen von Signalwerten geschaffen. Die Realisierung ermöglicht auch die praktische Evaluierung des Ansatzes im nächsten Kapitel.

Kapitel 7

7 Evaluation

Dieses Kapitel präsentiert die Evaluation des vorgestellten Ansatzes zur Laufzeitüberprüfung vernetzter eingebetteter Systeme. Die Evaluation ist selbst das vierte Forschungs-Ziel und soll Antwort geben, inwieweit der vorgestellte Ansatz die weiteren in Abschnitt 1.3 gestellten Ziele und Fragestellungen erfüllt. Dazu nutzt sie die in Kapitel 6 vorgestellte Implementierung.

Zunächst werden verschiedene Anwendungsfälle betrachtet, auf die der vorgestellte Ansatz angewendet wurde. Damit wird gezeigt, dass mit der Modellierung auch komplexe Interaktionen repräsentiert werden können.

Im zweiten Teil wird die Fähigkeit zur Erkennung aller Abweichungen von einem spezifizierten Soll-Verhalten ausgewertet. Dabei wird die Fragestellung, welche Abweichungen erkennbar sind, empirisch untersucht. Die theoretische Betrachtung in Kapitel 5 hat bereits gezeigt, das ein mit Resumption erweiterter Monitor alles unerwartete Verhalten in einem Trace erkennen kann. Die Evaluation wird dies bestätigen.

Wie in Theorem 3 gezeigt wurde, kann ein Monitor nur Abweichungen erkennen, die für ihn offensichtlich sind. Die Möglichkeiten, nicht offensichtlich abzuweichen, werden reduziert, wenn der Monitor seine Unsicherheit über den Zustand des System unter Beobachtung (SuO, engl.: *System under Observation*) reduziert. Daher wird angenommen, dass ein erweiterter Monitor genau dann Abweichungen gut erkennen kann, wenn Resumption gut funktioniert.

In Abschnitt 5.2 wurde die automatische Erweiterung eines Monitors mit Resumption durch Resumption-Strategien vorgestellt. In der Evaluation wird untersucht, wie verschiedene solcher Strategien sich auf die Erkennung von unterschiedlichen Arten von Abweichungen auswirken.

7.1 Verwendung in verschiedenen Anwendungsfällen

Dieser Abschnitt zeigt Beispiele, in denen die Modellierung unter Verwendung verschiedener Entwicklungsstufen des realisierten Prototyps in unterschiedlichen Anwendungsfällen verwendet und evaluiert wurde. Daran wird jeweils erläutert, welche Erkenntnisse aus dem Anwendungsfall für die iterative Realisierung der Konzepte gewonnen wurden. Ebenso werden Aspekte betrachtet, die sich auf die Ausarbeitung der vorgestellten Ansätze zur Modellierung und Resumption ausgewirkt haben. Die Anwendungsfälle wurden teilweise bereits in früheren Veröffentlichungen des Autors beschrieben [Dra+13; DPW15; DW17; DWB18].

7.1.1 Park Assistenz Dienst (ParkA)

Problemstellung Der erste Anwendungsfall betrachtet einen einfachen Proxy einer Park-Assistenz Software, genannt *ParkA* [Dra+13]. Hierbei handelt es sich um einen Dienst, wie er beispielsweise zur Realisierung eines Assistenz-Systems verwendet wird, das den Abstand zu Hindernissen in der Umgebung anzeigt. Der Proxy bietet einen zuverlässigen und konfigurierbaren Zugriff auf aktuelle Sensorwerte. Nachdem er gestartet wurde, sendet er diese mit der gewünschten Auflösung periodisch an einen registrierten Client. Der Client kann durch eine Anfrage beim Proxy die Auflösung im laufenden Betrieb verändern. Anhand dieses Anwendungsfalls wurde der Einsatz einer ersten Version der in Kapitel 6 vorgestellten Werkzeugplattform erprobt. Der Fokus lag dabei auf einer Beschreibung des Verhaltens an der Schnittstelle mit anschließender Überprüfung beobachteter Interaktionen zur Laufzeit.

Lösungsansatz Für die Beschreibung der Kommunikation zwischen dem Dienst ParkA und einem Client wurde das in Abbildung 7.1 abgebildete Referenz-Modell erstellt. Abbildung 7.1a zeigt einen Auszug der Schnittstelle des Dienstes. Sie bietet Methoden zum Hochfahren (engl.: *startup*) und Herunterfahren (engl.: *shutdown*) des Dienstes, sowie zum Einstellen der Auflösung (engl.: *resolution*) zurückgegebener Sensorwerte. Der ParkA Dienst sendet Sensorwerte mit dem Broadcast `sensorValues`. In Abbildung 7.1b ist ein Teil der definierten Ereignisse für die Schnittstelle zu sehen. Beispielsweise werden für das Hochfahren unterschiedliche Ereignisse für die Antwort des Methodenaufrufs definiert, abhängig davon, ob ein erfolgreicher Start oder ein Fehler zurückgemeldet wird.

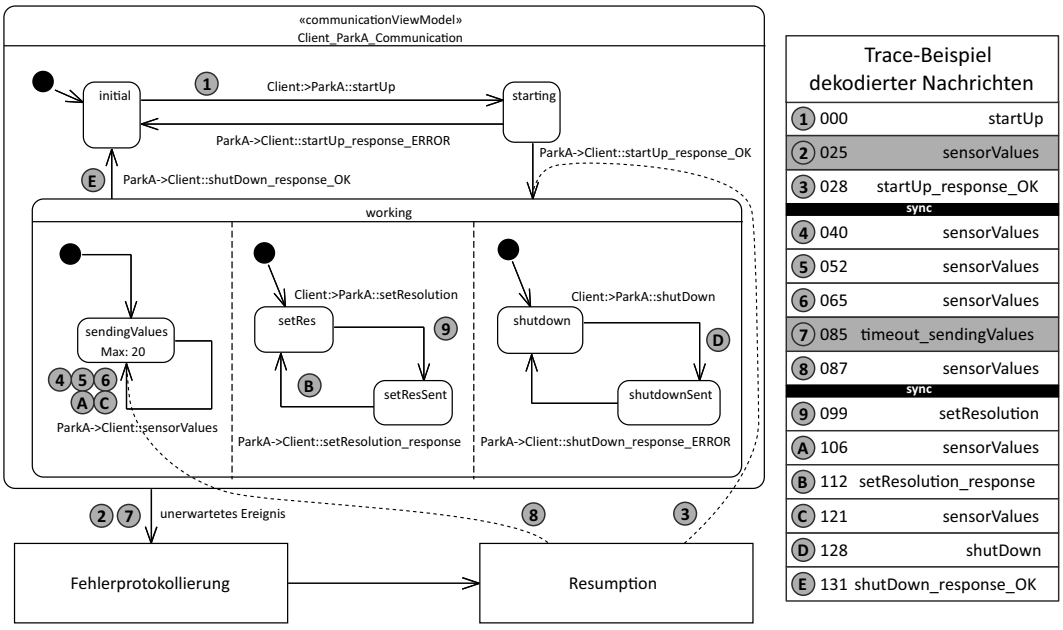
Neben der Struktur und den Ereignissen der Schnittstelle wird auch das Protokoll für eine konforme Interaktion mit dem ParkA Dienst definiert. Das Verhaltensmodell ist in Abbildung 7.1c dargestellt. Um die Kommunikation zu initiieren, muss der Client die `startUp`-Methode aufrufen. Bei Erfolg (`startUp_response_OK`) ist der Client verbunden

```
interface ParkA{
  method startUp{
    in{Resolution res}
    out{AM_Error returnValue} }
  broadcast sensorValues {
    out{
      Int32[] front_lmr
      Int32[] rear_lmr
      Resolution res } }
  enumeration Resolution{ cm mm }
  enumeration AM_Error{ OK ERROR }
  ... }
```

(a) Auszug der ParkA Schnittstellendefinition.

```
Interface ParkA {
  CallEvent startUp {
    methodRef startUp }
  ResponseEvent startUp_response_OK {
    methodRef startUp
    constraint {returnValue == OK} }
  ResponseEvent startUp_response_ERROR {
    methodRef startUp
    constraint {returnValue != OK} }
  BroadcastEvent sensorValues {
    methodRef sensorValues }
  ... }
```

(b) Auszug der ParkA Ereignisdefinition.



(c) ParkA Verhaltensmodell und Trace-Beispiel.

Abb. 7.1: Referenz-Modell der Kommunikation zwischen dem ParkA Dienst und einem Client. Anhand eines Trace-Beispiels wird die Überprüfung mit Resumption veranschaulicht. [Dra+13]

und der Dienst beginnt zyklisch Sensordaten mit dem Broadcast `sensorValues` zu senden. Die Zeit zwischen diesen Broadcasts darf dabei 20ms nicht übersteigen (`Max: 20`). Der Client kann im `working`-Zustand jederzeit die Auflösung der gemeldeten Sensordaten mit der `setResolution`-Methode verändern oder den Dienst über die `shutDown`-Method herunterfahren. Da diese Dinge unabhängig voneinander stattfinden, werden sie mit parallelen Regionen beschrieben.

Anhand der Abbildung wird zudem die Überprüfung eines Traces unter Verwendung von Resumption (siehe Kapitel 5) erläutert. Dadurch wird eine (Wieder-)Synchronisierung der Zustände von Modell und SuO nach einer Abweichung möglich. Der Trace ist rechts in Abbildung 7.1c abgebildet. Jede Zeile enthält eine Nummer zur Identifikation, einen Zeitstempel und den Namen des Ereignisses der bereits dekodierten Nachricht. In diesem Beispiel wurden die Nummern der Ereignisse an den entsprechenden Transitionen anno-

tiert. Ebenso sind dort zur Erläuterung zusätzlich die Schritte der Fehlerprotokollierung und Resumption eingezeichnet. In der implementierten Entwicklungsumgebung kann dies u. a. anhand der Animation des Automaten nachvollzogen werden. Dies ist genauer in Abschnitt 6.5 beschrieben.

In dem Trace beginnt der Dienst ParkA mit dem Senden der Sensorwerte direkt nach dem Aufruf der `startUp`-Methode. Dies ist eine Abweichung vom spezifizierten Verhalten und löst damit ein `unerwartetes Ereignis` aus. Der Fehler wird protokolliert und das nächste Ereignis (`startUp_response_OK`) wird zur Synchronisierung verwendet. Die nächste Abweichung wird bei Nachricht ⑦ erkannt. Die Zeitschranke des Zustands `sendingValues` wird überschritten und erzeugt ein `timeout_sendingValues`-Ereignis. Da das Ereignis keine Transition an einem aktiven Zustand auslöst, handelt es sich um ein unerwartetes Ereignis. Resumption sollte mit dem nächsten `sensorValues`-Broadcast den aktuellen Zustand wiederherstellen. Im Trace treten keine weiteren Abweichungen auf. Zwecks Übersichtlichkeit wurden nur die betroffenen Transitionen vom Resumption-Zustand aus eingezeichnet.

Ergebnisse Am Beispiel dieses Anwendungsfalls wurde realitätsnah erprobt, wie das Soll-Verhalten eines Systems mit einem Referenz-Modell beschrieben und automatisiert überprüft wird. Hierfür wurden bereits erste Prototypen der beschriebenen Implementierung verwendet. Zudem konnte anhand des Beispiels die Grundidee von Resumption aufgezeigt und einem ersten Test unterzogen werden.

7.1.2 Quellenwechsel

Problemstellung Der Anwendungsfall *Quellenwechsel* befasst sich damit, wie parallel die Lebenszyklen verschiedener Audioverbindungen in einem Infotainment System auf korrekte Interaktion mit einem zentralen AudioManager überprüft werden können. Dazu wurden die Interaktionen zwischen der Mensch-Maschine-Schnittstelle (HMI, engl.: *Human Machine Interface*) und dem AudioManager modelliert und die Überprüfung an einem Prototyp der Automotive Infotainment-Plattform GENIVI [GENIVI] evaluiert. Der Demonstrationsaufbau ist in Abbildung 7.2 abgebildet.

Eine besondere Herausforderung in diesem Anwendungsfall ist, dass mehrere unabhängige Audioverbindungen gleichzeitig existieren können, deren Lebenszyklen teilweise unabhängig ablaufen. Die Überprüfung des konformen Verhaltens an dieser Schnittstelle ist besonders relevant, da Audioverbindungen von Komponenten verschiedener Hersteller aufgebaut werden. Unerkanntes Fehlverhalten kann sich auf andere Komponenten auswirken oder wie ein Fehler einer anderen Komponente aussehen.

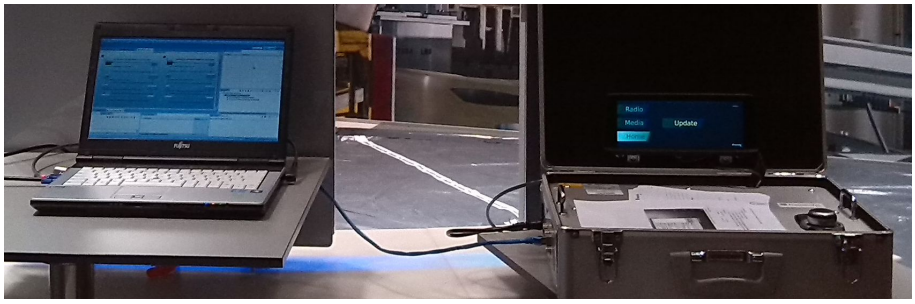


Abb. 7.2: Demonstration der Live-Überprüfung des Quellenwechsels an dem GENIVI Prototypen.

Lösungsansatz für bekannte Anzahl gleichzeitiger Verbindungen Zunächst wurde der Quellenwechsel auf zwei Verbindungen gleichzeitig vereinfacht betrachtet, um deren Verhalten mit einer festen Anzahl an parallelen Regionen beschreiben zu können (vgl. Abbildung 7.3). Die beiden parallelen Regionen besitzen eine identische Struktur, nur ihre Ereignisse unterscheiden sich durch die Referenz verschiedener Variablen mit der Verbindungskennung, aus der hervorgeht, welche Verbindung sie beobachten.

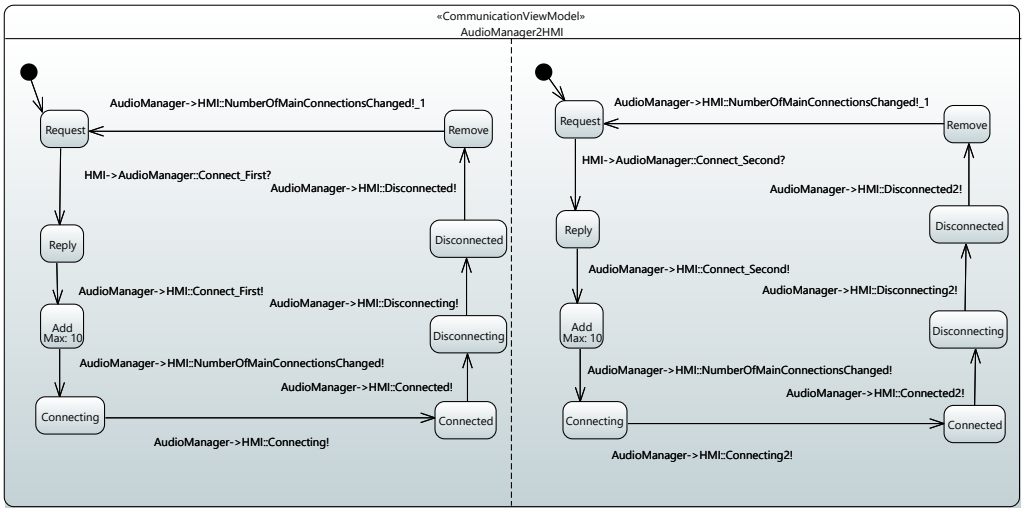
Als dieser Anwendungsfall untersucht wurde, verfügte die Implementierung der Überprüfung lediglich über einen globalen Kontext, in dem die Werte für den Vergleich hinterlegt werden konnten. Der Einsatz der Überprüfung konnte so aber bereits für Fälle, in denen Verbindungen zu maximal zwei Audioquellen gleichzeitig aktiv sind, live an dem in Abbildung 7.2 abgebildeten Prototypen demonstriert werden. Dabei wurden beispielsweise unerwartet mehrfach gesendete Nachrichten in der GENIVI Plattform entdeckt.

Allgemeiner Lösungsansatz Der Anwendungsfall wurde erneut betrachtet, nachdem Instanzen und Kontexte in die Plattform integriert wurden. Das Zustandsdiagramm ist in Abbildung 7.4b dargestellt und enthält zwei Zustände auf oberster Ebene.

Der Zustand *Listener* ist dabei vor und nach jedem Überprüfungsschritt aktiv. Im Fall einer Anfrage für eine neue Verbindung (*ConnectReq*) wird eine neue Instanz des Zustands *ConnectionLife* gestartet und zurück in den Zustand *Listener* gewechselt. In dem Zustand *ConnectionLife* ist das Verhalten analog zu einer Region des in Abbildung 7.3b dargestellten Zustandsdiagramms beschrieben. Auf das Schließen des Zyklus konnte aber verzichtet werden, da die Überprüfung zukünftiger Verbindungen in einer eigenen Instanz stattfindet. Die Zuordnung von Nachrichten bzw. Ereignissen zu den verschiedenen Instanzen erfolgt auch hier über die Verbindungskennung, die nun in einem an die Instanz geknüpften Kontext (*ConnectionLifetime*) hinterlegt wird. Somit können beliebig viele gleichzeitig aktive Verbindungen überprüft werden.

```
import "../franca/CommandInterface.fidl"
var Double firstConnectionID
var Double secondConnectionID
Interface CommandInterface {
   CallEvent Connect { methodRef Connect
      children {
         CallEvent Connect_First {
            constraint { (isset firstConnectionID == false) && (sourceID > 0) && (sinkID > 0) }}
         CallEvent Connect_Second {
            constraint { (isset firstConnectionID) && (sourceID > 0) && (sinkID > 0) }}}}
   ResponseEvent Connect { methodRef Connect
      children {
         ResponseEvent Connect_First {
            constraint { (isset firstConnectionID == false) && (mainConnectionID > 0) && (result == E_OK) }
            action { firstConnectionID = mainConnectionID }}
         ResponseEvent Connect_Second {
            constraint { (isset firstConnectionID) && (firstConnectionID != mainConnectionID) && (result == E_OK) }
            action { secondConnectionID = mainConnectionID }}}}
... }
```

(a) Auszug der Ereignisdefinition.

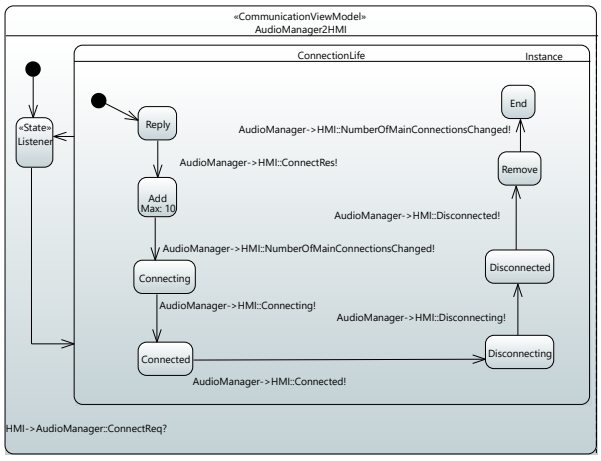


(b) Verhaltensmodell.

Abb. 7.3: Modellierung zweier Lebenszyklen von Audioverbindungen durch parallelen Regionen.

```
import "../franca/CommandInterface.fidl"
Interface CommandInterface {
   CallEvent ConnectReq { methodRef Connect
      constraint { (sourceID > 0) && (sinkID > 0) }}
   ResponseEvent ConnectRes { methodRef Connect
      constraint { (mainConnectionID > 0)
                   && (result == E_OK) }}
... }
Context ConnectionLifetime {
   var UInt16 _connectionId
   constraint ConnectReq {isset _connectionId == false}
         {_connectionId = 0}
   constraint ConnectRes {_connectionId == 0
            || _connectionId == mainConnectionID}
         {_connectionId = mainConnectionID}
... }
```

(a) Auszug der Ereignisdefinition.



(b) Verhaltensmodell.

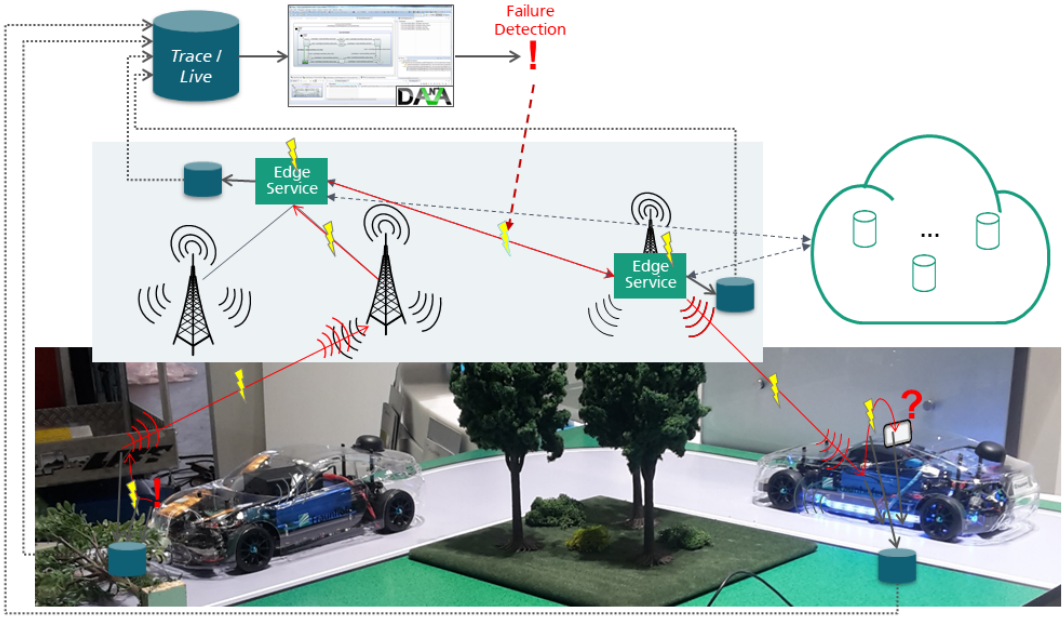
Abb. 7.4: Modellierung der Lebenszyklen von Audioverbindungen durch Instanz und Kontext.

Ergebnisse Durch die Erweiterung der Modellierung um Instanzen und Kontexte kann der Anwendungsfall deutlich kompakter beschrieben werden, wie ein Vergleich von Abbildung 7.3 und Abbildung 7.4 veranschaulicht. Sowohl in der Ereignisdefinition als auch in dem Verhaltensmodell entfallen die redundanten Elemente. Zudem sind alle Ereignisse, die eine Verbindung betreffen, nun in einer Instanz zusammengefasst, wodurch eine mögliche weitere Auswertung vereinfacht wird.

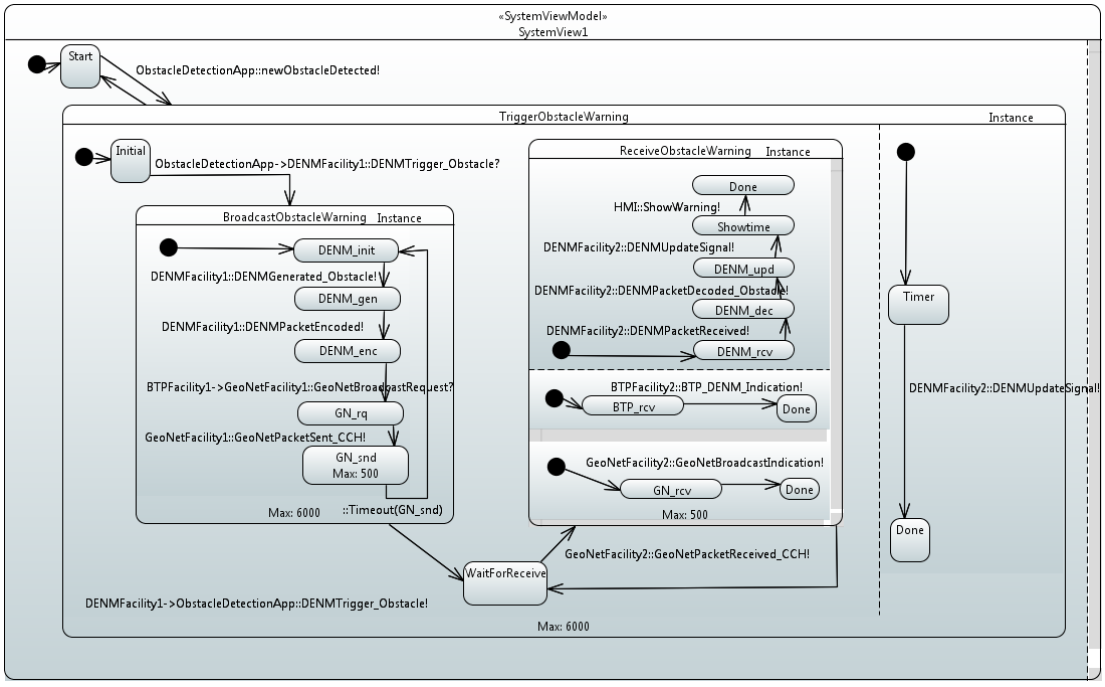
7.1.3 Gefahrenwarner Applikation

Problemstellung Das nächste Beispiel betrachtet eine *Gefahrenwarner Applikation* [DWB18]. Ein plötzlich auftauchendes Hindernis im Verkehr ist gefährlich, insbesondere wenn Fahrer das Hindernis erst spät oder nur indirekt durch die Reaktionen anderer wahrnehmen können. Eine Gefahrenwarnung hilft, die Fahrer frühzeitig zu informieren. Beispielsweise bremst der Fahrer des linken Autos in Abbildung 7.5a scharf, da er ein Hindernis bemerkt. Da die Bäume in der Kurve die Sicht der Fahrerin im rechten Auto verdecken, kann sie erst spät auf die Gefahr reagieren. Sie könnte früher mit dem Bremsen beginnen, wenn sie eine Meldung mit einer Gefahrenwarnung von dem vorderen Auto erhält. Wenn eine Nachricht nicht angezeigt wird, kann dies viele mögliche Ursachen haben. Der in dieser Arbeit vorgestellte Ansatz ermöglicht, auch vernetzte Systeme wie verbundene Autos zu analysieren. Durch Abbilden des erwarteten Verhaltens in einem Modell, kann das beobachtete Verhalten damit überprüft und die Fehlersuche vereinfacht werden.

Lösungsansatz Wenn Systeme und Komponenten während der Entwicklung verändert oder ausgetauscht werden, ist beispielsweise eine weitere Fehlerquelle die Konfiguration des Kommunikationsstacks. Deshalb wurde für den Anwendungsfall der Weg einer Nachricht in Sender und Empfänger modelliert (Abbildung 7.5b), um Abweichungen frühzeitig festzustellen und die Fehlerursache schnell eingrenzen zu können. Um auch mehrere überlappende Meldungen prüfen zu können, wird für jedes erkannte Hindernis der Zustand `TriggerObstacleWarning` instanziiert. Darin wird auf der linken Seite in Zustand `BroadcastObstacleWarning` der Ablauf im Stack des Senders geprüft und rechts in Zustand `ReceiveObstacleWarning` für einen Empfänger. Da der Sender die Nachricht mehrmals wiederholt, während mehrere Empfänger erfolgreich übertragene Nachrichten bereits verarbeiten können, werden beide Zustände instanziiert. Die rechte Region des Zustands `TriggerObstacleWarning` überwacht, ob die Gefahrenmeldung innerhalb einer vorgegebenen Zeit bei dem Ziel ankommt. Die Beobachtungen können den verschiedenen Instanzen über Kontexte zugeordnet werden. Abweichungen im Verhalten einer Implementierung der Gefahrenwarnung können dadurch identifiziert werden.



(a) Illustration des Anwendungsfalls.



(b) Verhaltensmodell des Anwendungsfalls.

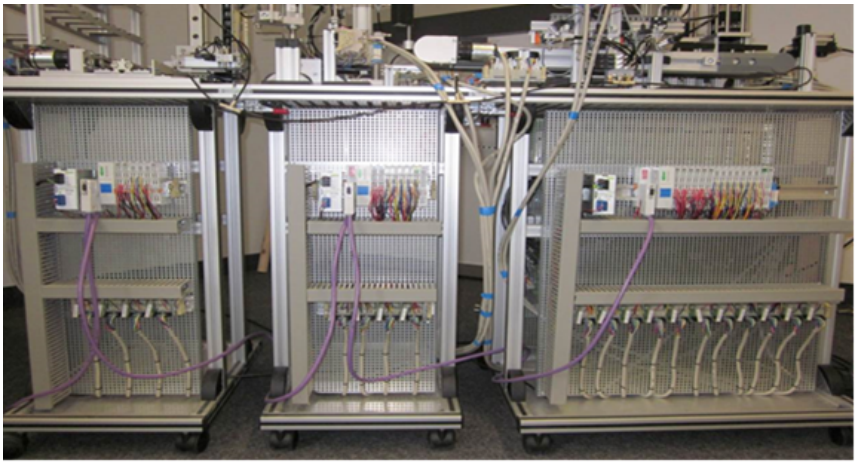
Abb. 7.5: Anwendungsfall einer Gefahrenwarner Applikation.

Ergebnisse Beispielsweise kann durch Beobachten des Ablaufs in der Animation des Zustandsautomaten leicht die Ursache lokalisiert werden, warum eine Gefahrenwarung nicht beim Empfänger angezeigt wurde. Dabei hilft auch, dass durch die Verwendung von Instanzen und Kontexten der Ablauf kompakt und übersichtlich dargestellt wird. Resumption erleichtert in diesem Beispiel auch die Einrichtung der Konfiguration zur Überwachung. Zunächst wurden die Unterzustände von `ReceiveObstacleWarning` auch als Sequenz modelliert. Obwohl die Gefahrenmeldung empfangen wurde, sind häufig Abweichungen im Verlauf gemeldet worden. Dank Resumption wurden die Nachrichten aber weiterhin der richtigen Instanz zugeordnet und überprüft, wenn auch mit einigen Fehlermeldungen. Eine Analyse der Meldungen hat ergeben, dass der Gefahrenwarner richtig funktioniert und die Ereignisse bei der Beobachtung der Dienste `GeoNetFacility2` und `BTPFacility2` in einer unerwarteten Reihenfolge auftreten. Bedingt durch die Art der verwendeten Instrumentierung, können die Nachrichten der beiden Dienste nicht immer in der kausalen Reihenfolge beobachtet werden. Die Zustände wurden daher teilweise in parallelen Regionen modelliert.

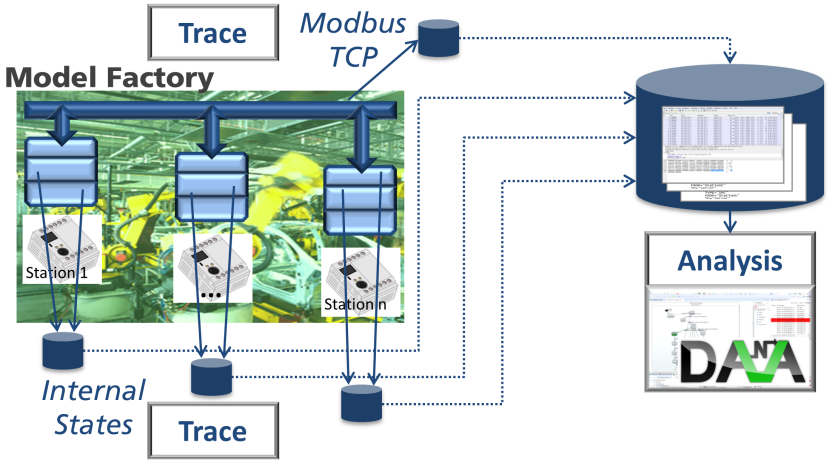
7.1.4 Gelerntes Modell einer Industrieanlage

Problemstellung Die Modellierung und Resumption wurde auch für Anwendungsfälle außerhalb der Automobilbranche eingesetzt. Abbildung 7.6a zeigt eine kleine Industrieanlage, die aus drei Stationen besteht und Würfel aus zwei Hälften zusammensetzt. Die erste Station entnimmt Teile aus zwei Magazinen und überprüft deren Orientierung und Material. Die zweite Station setzt die Teile mit einer hydraulischen Presse zusammen. Die zusammengesetzten Würfel werden dann in der dritten Station aufbewahrt. Im Rahmen dieses Anwendungsfalls wurde auch die Modularität der Plattform erprobt.

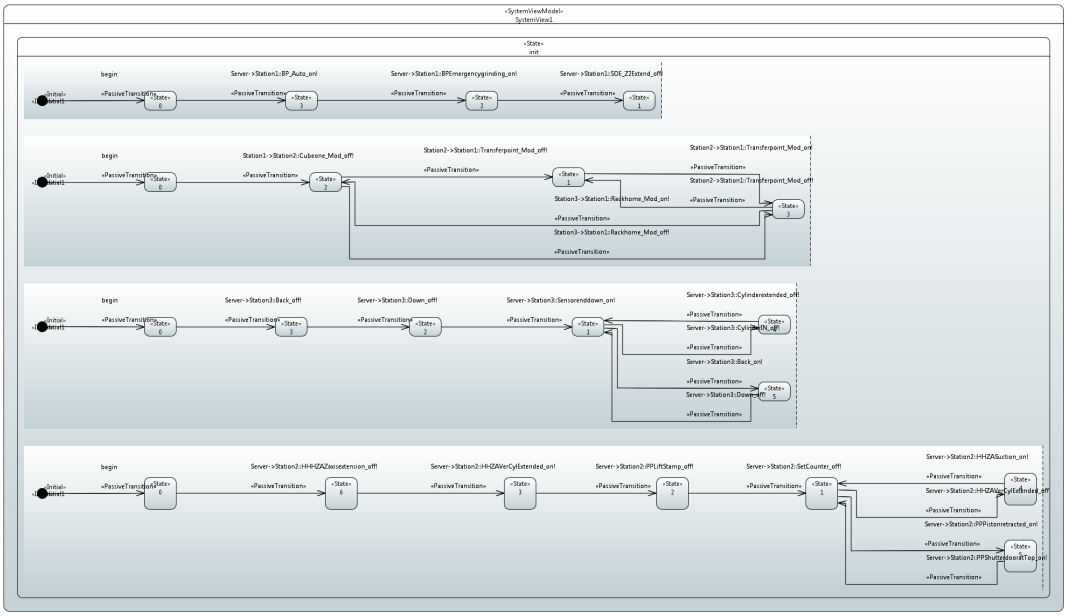
Lösungsansatz In der Masterarbeit von Salvi [Sal17] wurde die Dana Plattform um eine halbautomatische Erstellung von Referenz-Modellen durch selbst-lernende Methoden erweitert. Das Verfahren erhält als Eingabe die Schnittstellenbeschreibung und lernt dann anhand von Traces die Ereignisse und das Verhaltensmodell. Durch eine entsprechende Vorrichtung in der Laufzeitumgebung der Stationen, können Änderungen an Sensoren und Aktuatoren beobachtet werden, ohne das ursprüngliche Kontrollprogramm zu verändern. Die Kommunikation zwischen den Stationen wird an einem Switch der Ethernet-basierten Modbus TCP Verbindungen abgegriffen. Dies ist schematisch in Abbildung 7.6b abgebildet. Abbildung 7.6c zeigt ein gelerntes Verhaltensmodell für den beschriebenen Anwendungsfall. Eine der parallelen Regionen beschreibt die Kommunikation zwischen den Stationen und die anderen jeweils das Verhalten einer Station.



(a) Foto der Modellanlage.



(b) Illustration des Anwendungsfalls.



(c) Verhaltensmodell des Anwendungsfalls.

Abb. 7.6: Anwendungsfall einer kleinen Industrieanlage.

Ergebnisse Die Erweiterbarkeit der in Abschnitt 6.2 vorgestellten flexiblen Architektur zur Datenverarbeitung hat ermöglicht, für Lernen und Überprüfung die gleichen Mechanismen zum Einlesen der Traces zu verwenden. Durch die modulare Konfiguration kann einfach zwischen den beiden umgeschaltet werden.

Resumption hilft bei dem Umgang mit Unvollkommenheiten, die ein gelerntes Modell, gerade in frühen Phasen, haben kann. Zum Beispiel, wenn noch nicht alles erwartete Verhalten beobachtet wurde und daher nicht in das Modell einfließen konnte. Wenn der Monitor unerwartetes Verhalten meldet, kann Resumption Model- und Systemzustand wieder ausrichten, um die Überprüfung fortzusetzen. Wenn der Entwickler die Meldung untersucht und feststellt, dass es sich um erwartetes Verhalten handelt, kann damit das Verhaltensmodell überarbeitet werden.

7.1.5 Auswertung von Zeitverhalten

Problemstellung In diesem Anwendungsfall werden Daten in einer längeren Übertragungskette über verschiedene Stationen hinweg weitergereicht. Um die Leistung und Zuverlässigkeit der Kette zu messen und das Potenzial für Verbesserungen zu bestimmen, sollte bestimmt werden, wie lange die Daten für die Übertragung insgesamt sowie zwischen den verschiedenen Stationen brauchen und wie diese Zeiten schwanken. Die Reihenfolge der Nachrichten kann sich an jeder Station ändern und sie enthalten keine spezielle Kennung, um sie über verschiedene Stationen hinweg zu verfolgen. Dennoch sollten nicht nur die Zeiten zwischen zwei Stationen, sondern auch die Ende-zu-Ende Zeiten ermittelt werden, also von Anfang bis Ende der Kette.

Lösungsansatz Die Nachrichten lagen in Traces je Station vor, wobei jede eine eigene Nachrichtenstruktur verwendete. Für die verschiedenen Strukturen wurden entsprechende Schnittstellenbeschreibungen erstellt und die notwendigen Feeder realisiert. Als Modell diente das gleiche Grundmuster wie im Quellenwechsel mit Kontexten.

Das Zustandsdiagramm ist in Abbildung 7.7 dargestellt und besteht aus zwei Zuständen auf oberster Ebene, von denen einer auf den Beginn neuer Ketten wartet. Wenn das erste Ereignis der Kette auftritt, wird der zweite Zustand instanziiert, der die Kette mit Unterzuständen nachvollzieht. Auch die Zusammenfassung von Nachrichten einer Übertragungskette konnte dank Verwendung eines Kontexts schnell implementiert werden. Jede Nachricht der Kette enthält Merkmale, die dort hinterlegt und überprüft werden können.

Für die eigentliche Erhebung der statistischen Werte wurde zu diesem Zweck die Erweiterung *Statistik* der Verarbeitungskette realisiert. Hiermit kann erfasst werden, wie oft und wie

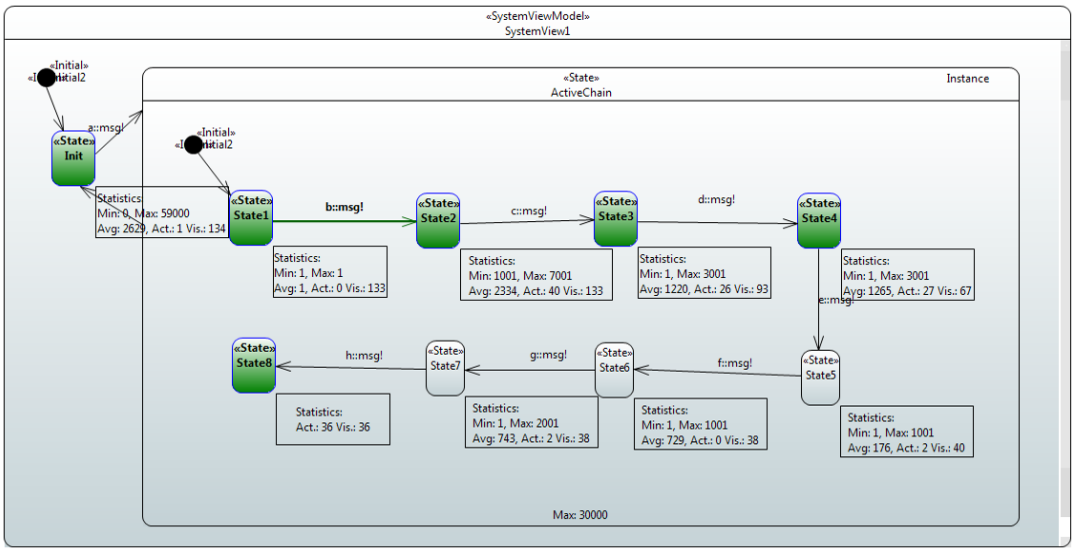


Abb. 7.7: Beispiel mit Auswertung und Anzeige von Statistiken zur Analyse der Verweildauer in den Zuständen bzw. dem Abstand oder Jitter von Nachrichten.

lange ein Zustand aktiv war und dies optional auch im Zustandsdiagramm zusammen mit der Animation live angezeigt werden. Die Zeit, die ein Zustand aktiv ist, entspricht dabei der Zeit zwischen zwei Ereignissen der Kette, beispielsweise dem Eintreffen einer Nachricht an einer Station und dem Abschicken an die Nächste. Die Ende-zu-Ende Zeiten können durch den Zustand *ActiveChain* erfasst werden.

Ergebnisse Mit diesem Anwendungsfall wurden insbesondere folgende Aspekte des Ansatzes und der Realisierung herausgefordert:

- Zuordnung von Nachrichten in unterschiedlichen Formaten
- Umgang mit großen Nachrichtenmengen
- Erweiterung der Auswertung um Statistiken über aktive Zustände

Die großen Datenmengen haben dazu beigetragen, Speicherlecks und Flaschenhälse in der Datenverarbeitung der Realisierung aufzudecken und zu beheben. Insbesondere die Anzeige der Nachrichten in der Tabelle der Ereignisanzeige war hier bei Verwendung nativer Java-Elemente problematisch und es musste auf eine Alternative umgestiegen werden. Zudem wurde die Notwendigkeit deutlich, das Suchfenster, also wie lange ein Kontext aktiv ist, zu limitieren, damit dieser schließlich freigegeben werden kann. Hierfür wurde die Zeitschranke des instanziierten Zustands verwendet, die ebenfalls instanziiert wird und damit die maximal verfolgte Dauer der Übertragungskette festlegt. Durch den Anwendungsfall konnte gezeigt werden, wie der vorgestellte Ansatz zur statistischen Analyse von Zeitverhalten eines Systems verwendet werden kann. Das gleiche Modell ermöglicht auch eine Auswertung in Echtzeit, wenn die Traces zentral zusammengeführte werden.

7.2 Vergleich verschiedener Resumption-Strategien

Der erste Teil der Evaluation hat anhand von verschiedenen Anwendungsfällen die Anwendbarkeit der Modellierungsmethodik gezeigt und die daraus resultierenden Erfahrungen durch den Einsatz der Methode zur Laufzeitüberprüfung betrachtet. Im zweiten Teil der Evaluation wird experimentell untersucht, wie sich verschiedene Resumption-Strategien auf die Identifizierung von Abweichungen auswirkt. Dazu wird im Folgenden das Evaluierungssetup erläutert, bevor die betrachteten Resumption-Strategien vorgestellt werden. Daran schließt sich die Präsentation und Diskussion der Ergebnisse an.

7.2.1 Evaluierungssetup

Der grundlegende Ansatz der Evaluation besteht darin, verschiedene Resumption-Strategien für eine Vielzahl verschiedener Modelle und Abweichungen zu beobachten. In Abbildung 7.8 ist ein Überblick über den Ablauf zur Gewinnung dieser Daten abgebildet.

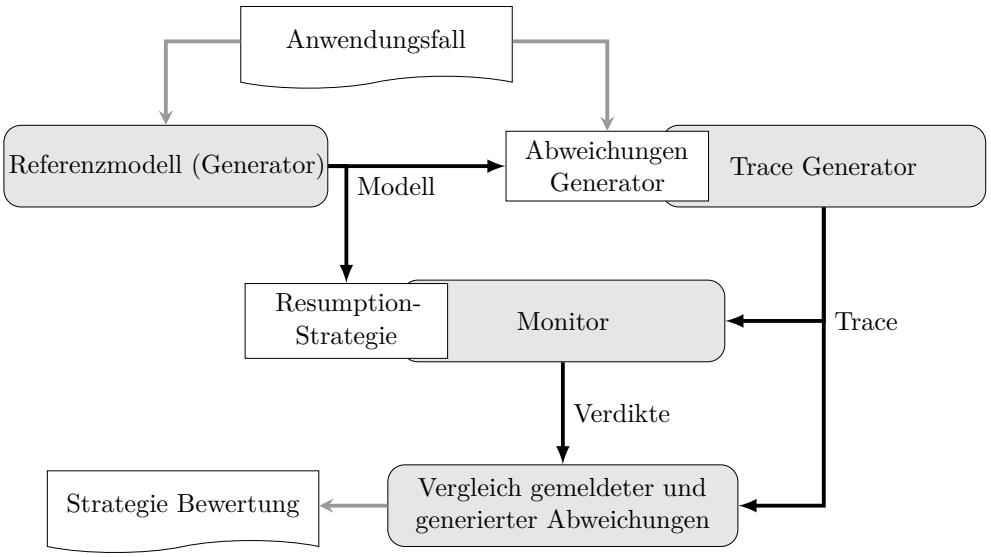


Abb. 7.8: Übersicht der Evaluierungsmethodik für Resumption-Strategien.

Ein konkreter *Anwendungsfall* stellt normalerweise eine Spezifikation und damit ein *Referenz-Modell*. Um im Rahmen dieser Evaluation eine allgemeine Aussage über die Leistung der Strategien zu ermöglichen und nicht nur einzelne Anwendungsfälle zu bewerten, wird ein Generator zur Erzeugung unterschiedlicher Modelle verwendet. Der Generator startet mit einem einzelnen Zustand und verfeinert das Modell über mehrere Iterationen. In jedem Iterationsschritt wird ein Zustand zufällig gewählt und durch eine zufällige Anzahl an Zuständen ersetzt. Die ein- und ausgehenden Transitionen des ersetzten Zustands

werden zufälligen Zuständen der Untermenge zugeordnet. Die neuen Zustände werden dann zufällig untereinander mit Transitionen verbunden. Dabei wird garantiert, dass jeder Zustand von mindestens einer eingehenden Transition des ursprünglichen Zustands erreicht werden kann und eine der ausgehenden Transitionen erreicht. Die Menge der Ereignisse, aus der die Auslöser der Transition zufällig gewählt werden, verändert sich mit jedem Iterationsschritt. Eine zufällige Auswahl an Ereignissen wird entfernt und dafür werden neue hinzugefügt. Daraus ergeben sich globale Ereignisse, die im gesamten Automaten verwendet werden, sowie lokale Gruppen, die teilweise spezielle Ereignisse verwenden. Die Anzahl der Iterationsschritte variiert, um Zustandsautomaten unterschiedlicher Größe und Struktur zu erzeugen. Die Generierung wurde unter der Annahme entworfen, dass eine iterative Verfeinerung bei der Erstellung einer Spezifikation verwendet wird. Die iterative Generierung macht das Auftreten globaler Ereignisse und lokaler Gruppen wahrscheinlicher. Dies schränkt aber nicht die möglichen erzeugten Automaten ein. Erfolgt beispielsweise nur ein Iterationsschritt, erzeugt dies einen Automaten mit einer beliebigen Anzahl an Zuständen und Transitionen.

Um die Automaten zu charakterisieren, werden verschiedene Metriken über ihre Struktur berechnet, beispielsweise ihre Anzahl an Zuständen und die *Eindeutigkeit* eines Automaten. *Eindeutigkeit* bezeichnet die Wahrscheinlichkeit, dass ein Ereignis eindeutig ist, also einen bestimmten Folgezustand festlegt. Als Annäherung wird dazu der Anteil aller Transitionen eines Automaten mit einem eindeutigen Ereignis herangezogen. Diese Metrik ist für die Evaluation relevant, da es nach einem eindeutigen Ereignis nur genau einen möglichen Zustand im Verhaltensmodell gibt, der zu gültigem Verhalten passt.

Aus einer Verhaltensbeschreibung werden mehrere Traces erzeugt. Dafür wird sie zunächst durch einen *Abweichungen-Generator* modifiziert, bevor zufällig Pfade gewählt werden. Die Abweichungen entsprechen den vier in Abschnitt 2.2.4 vorgestellten Gruppen von Abweichungen. Eine Abweichung kann per Definition nur für Ereignisse auftreten, die für den aktuellen Zustand (q_s) nicht definiert sind, also für ein Ereignis χ mit $\chi \notin \mathbb{E}^{q_s}$. Die Gruppen der Abweichungen unterscheiden sich danach, welcher Zielzustand (q_t) gewählt wird. Es wird unterschieden zwischen *überflüssigen* ($q_t = q_s$), *veränderten* ($\exists e : \delta_{\mathcal{L}}(q_s, e) \mapsto q_t$), *ausgelassenen* ($\exists e : \delta_{\mathcal{L}}(q_s, e) \mapsto q_i \wedge \delta_{\mathcal{L}}(q_i, \chi) \mapsto q_t$) und *zufälligen* ($q_t \in \mathbb{S}_{\mathcal{L}}$) Ereignissen, die zu Abweichungen führen. Diese Gruppierung dient nicht dazu, erkannte Abweichungen darin eindeutig einzuordnen, deshalb können die Gruppen auch überlappen. Sie dient vielmehr der Einordnung der Ergebnisse dieser Evaluation, da in Abhängigkeit von dem konkreten Anwendungsfall bestimmte Abweichungen häufiger vorkommen können als andere. Die Ergebnisse werden deshalb nach diesen Gruppen von Abweichungen aufgeschlüsselt. Die so generierten Transitionen für Abweichungen entsprechen den Fehlern in einer realen Implementierung. Komplexere Abweichungen können durch Kombination simuliert werden. Um den Einfluss jeder Gruppe von Abweichungen getrennt betrachten zu können, wird für

jeden Trace nur eine Gruppe angewendet. Die injizierten Abweichungen werden für die spätere Analyse in den Meta-Daten des Traces markiert, diese sind für den Monitor nicht sichtbar.

Die Traces werden schließlich von einem Monitor überprüft, der durch Erweiterung der Verhaltensbeschreibung mit einer Resumption-Strategie aufgebaut wird. Für die Evaluation wurde das Eclipse-basierte Werkzeug DANA erweitert und verwendet. Das Werkzeug und die wichtigsten im Rahmen dieser Arbeit durchgeführten Erweiterungen wurden in Kapitel 6 vorgestellt. Durch Schnittstellen in der Ausführung der Überprüfung kann Resumption bei Bedarf zugeschaltet werden. Dies ermöglicht einen leichten Wechsel zwischen den verschiedenen Resumption-Strategien. Das flexible Ausführungskonzept ermöglicht auch, die erweiterten Monitore parallel zu betreiben und auszuwerten. Dadurch erhalten sie in jedem Durchlauf alle denselben Trace als Eingabe. Zusätzlich wird für jeden Durchlauf zum Vergleich auch ein alternatives Out-of-Service Verfahren angewendet, das durch Analyse des gesamten Traces bestimmt, wie mit einer minimalen Anzahl an Änderungen (engl.: *least changes*) ein gültiger Trace erstellt werden kann. Dieses wird im Folgenden als $\mathcal{R}_{\text{l-c}}$ bezeichnet und wurde in Abschnitt 3.2.1 vorgestellt.

Das Evaluierungssetups soll messen, wie *gut* ein bestimmter Monitor in einem gegebenen Anwendungsfall mehrere Abweichungen in einem Lauf entdecken kann. Deshalb richtet sich die Bewertung der Leistung nach einem *Vergleich gemeldeter und generierter Abweichungen*. Dazu werden bekannte Metriken aus dem Bereich *Information Retrieval* verwendet: *Precision* und *Recall* [Pow11; Yin+16].

Precision (7.1), beschreibt die **Genauigkeit** und ist der Anteil der gemeldeten Abweichungen (engl.: *reported deviation*, rd) die richtig erkannt (engl.: *true deviation*, td) wurden.

$$p = |td \cap rd|/|rd| \tag{7.1}$$

Recall (7.2) ist die **Vollständigkeit** der Treffermenge und berechnet sich als Anteil aller Abweichungen, die auch gemeldet wurden.

$$r = |td \cap rd|/|td| \tag{7.2}$$

Zur Kombination beider Werte wird üblicherweise ihr harmonisches Mittel gebildet, auch bekannt als F_1 Score (7.3).

$$F_1 = 2 \cdot \frac{p \cdot r}{p + r} \tag{7.3}$$

Ein Monitor, der alle und nur wahre Abweichungen meldet, hat perfekte Genauigkeit ($p = 1$) und Vollständigkeit ($r = 1$). Bei bekanntem Startzustand und bis zur ersten Abweichung haben alle erweiterten Monitore diese Genauigkeit (vgl. Theorem 1). Gewöhnliche Monitore

behalten eine perfekte Genauigkeit nur, weil sie alles Folgende ignorieren. Erweiterte Monitore können an Genauigkeit verlieren, da sie versuchen, weitere Abweichungen zu finden. Deshalb misst die Vollständigkeit, wie zuverlässig alle wahren Abweichungen gemeldet werden. Ein gewöhnlicher Monitor meldet nur die erste Abweichung, seine Vollständigkeit entspricht also $|td|^{-1}$.

7.2.2 Betrachtete Resumption-Strategien

Dieser Abschnitt stellt die verglichenen Strategien für Resumption vor. Ähnliche Strategien werden häufig verwendet oder nachgeahmt, um beim manuellen Erstellen eines robusten Monitors die Spezifikation zu erweitern. Basierend auf einem beobachteten Ereignis und einer Menge von Kandidaten für den aktiven Zustand, bestimmt die Resumption-Strategie ($\mathcal{R}$) die Kandidaten für den neuen Zustand des SuO in Bezug auf die Spezifikation.

Die präsentierten Strategien gliedern sich in zwei Kategorien. *Lokale* Strategien berücksichtigen den Zustand, der vor einer Abweichung aktiv war, während *globale* Strategien alle Zustände gleich betrachten. Da mit jeder Strategie eine Variante der erweiterten Transitionsrelation δ^+, wie in Abschnitt 5.1.2 vorgestellt, definiert wird, bestimmt jede Strategie ein eigenes *Fehlermodell*. Sie finden nur dann garantiert alles unerwartete Verhalten, wenn die auftretenden Abweichungen zu diesem Fehlermodell passen. Einige Annahmen für ein Fehlermodell führen allerdings zu einem erheblich einfacherem erweitertem Monitor $\mathrm{M}_{\mathcal{E}}$. Als Beispiel wird jeweils das Ergebnis der Anwendung der Strategie auf den in Abbildung 2.1 gezeigten Automaten dargestellt. Dabei kennzeichnen **Fett-** und normal-gedruckte Beschriftungen jeweils **akzeptierende** und ablehnende Verdikte. Die Kennzeichnung mit $*$ ist stellvertretend für alle Ereignisse, die von keiner anderen Transition an dem ausgehenden Zustand aktivieren werden.

Waiting Die lokale Strategie *Waiting* (dt.: *warten*, $\mathcal{R}_{\text{wait}}$ (7.4)) wird häufig auch ohne Absicht verwendet, wenn der Trace von einem unveränderten Automaten interpretiert wird. Dies liegt daran, dass Implementierungen normalerweise zusätzliche Nachrichten ignorieren und im gleichen Zustand bleiben, um dort auf das nächste gültige Ereignis zu warten. Durch die Strategie entsteht entsprechend kein Overhead zur Laufzeit. Sie nimmt die Überprüfung mit dem nächsten Ereignis, das von dem bereits aktiven Zustand q akzeptiert wird, wieder

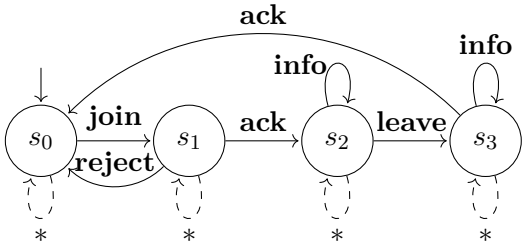


Abb. 7.9: Beispiel eines Automaten erweitert mit $\mathcal{R}_{wait}$.

auf. Solange bleibt der aktive Zustand q und alle Ereignisse, die nicht in $\mathbb{E}^q$ enthalten sind, werden abgelehnt. Damit entstehen Schleifen an jedem Zustand, wie sie für das Beispiel in Abbildung 7.9 gezeigt werden – die mit * markierten Transitionen. Die Annahme von $\mathcal{R}_{\text{wait}}$ ist, dass alle Abweichungen überflüssige Nachrichten sind, die einfach ignoriert werden können. Nur dann kann alles unerwartete Verhalten gefunden werden. Die Erwartung ist deshalb, dass die Strategie für andere Abweichungen schlecht funktionieren wird. Tauchen die Ereignisse der ausgehenden Transitionen in keinem anderen Zustand auf, kann der Monitor den Zustand nicht verlassen und wird alles ablehnen, auch wenn das weitere Verhalten des SuO der Spezifikation entspricht, bis das SuO sich wieder in dem Zustand befindet.

$$\mathcal{R}_{\text{wait}}(\mathbb{S}_{in}, e) = \mathbb{S}_{in} \tag{7.4}$$

Nearest Eine offensichtliche Gefahr besteht, wenn das SuO nie ein Ereignis erzeugt, das vom aktiven Zustand akzeptiert wird. Deshalb berücksichtigt die nächste Strategie auch Zustände in der Nähe des aktiven Zustands. Als Abstandsmaß $\|q_s, q_t\|$ wir die Anzahl der Transitionen $\in \delta_{\mathcal{L}}$ auf dem kürzesten Pfad zwischen einem Startzustand q_s und einem Zielzustand q_t verwendet. Das Maß wird für Mengen zu $\|\mathbb{S}_s, \mathbb{S}_t\|$ erweitert, und gibt dann die minimale Anzahl der Transitionen auf dem kürzesten Pfad, der mit einem Zustand aus $\mathbb{S}_s$ beginnt und einem aus $\mathbb{S}_t$ endet. Die Strategie *Nearest* (dt.: *nächster*, $\mathcal{R}_{\text{near}}$ (7.5)) nimmt die Überprüfung mit dem nächsten Ereignis wieder auf, das von einem Zustand akzeptiert wird, der vom aktiven Zustand erreicht werden kann. Abbildung 7.10 zeigt die Erweiterung des Beispiels mit dieser Strategie. Sollten mehrere Transitionen von erreichbaren Zuständen das Ereignis akzeptieren, wird die am wenigsten weit entfernte Transition gewählt. Zusätzliche Zustände für die Überprüfung gegenüber den Zuständen der Spezifikation ergeben sich dabei nur, wenn für ein Ereignis mehrere Transitionen den gleichen, kürzesten Abstand haben. In allen anderen Fällen wird die Strategie mit dem nächsten akzeptierten Ereignis die Überprüfung wiederaufnehmen. Das Fehlermodell besteht aus Abweichungen durch ausgelassene Nachrichten. Da die Strategie nur auf folgende Transitionen achtet, können überflüssige oder veränderte Nachrichten dazu führen, dass zu viele Transitionen ausgelassen werden.

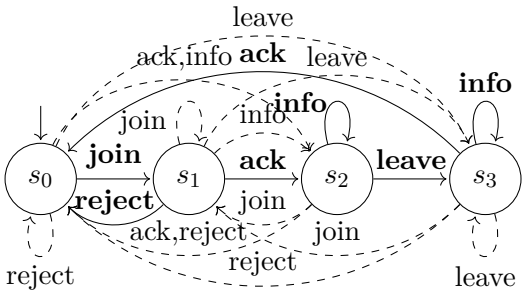


Abb. 7.10: Beispiel eines Automaten erweitert mit $\mathcal{R}_{near}$.

$$\mathcal{R}_{\text{near}}(\mathbb{S}_{in}, e) = \underset{q_t \in \delta_{\mathcal{L}}(\mathbb{S}_C, e)}{\operatorname{argmin}} \min_{q_s \in \mathbb{S}_{in}} \|q_s, q_t\| \tag{7.5}$$

Nearest-or-Waiting Die Strategie *Nearest-or-Waiting* (dt.: *nächster or warten*, $\mathcal{R}_{\text{n-o-w}}$ (7.6)) ist eine Kombination von *Nearest* und *Waiting*. Sie misst die Länge des kürzesten Pfads vom aktiven Zustand zu dem Zustand, der von *Nearest* gewählt wird und von einem beliebigen Zustand mit einer akzeptierenden Transition zum aktiven Zustand. Ist der zweite Pfad kürzer, wird stattdessen *Waiting* verwendet. Die Erweiterung des Beispiels mit $\mathcal{R}_{\text{n-o-w}}$ ist in Abbildung 7.11 abgebildet. Die Grundidee ist dabei, überflüssige Nachrichten zu ignorieren und diese dadurch zu identifizieren, indem soweit zurück geschaut wird, wie auch entlang der Transitionen nach einem Treffer gesucht werden musste.

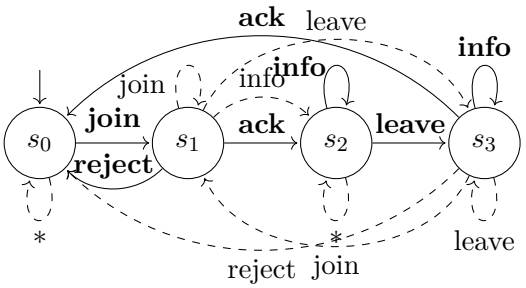


Abb. 7.11: Beispiel eines Automaten erweitert mit $\mathcal{R}_{n\text{-}o\text{-}w}$.

$$\mathcal{R}_{\text{n-o-w}}(\mathbb{S}_{in}, e) = \begin{cases} \mathcal{R}_{\text{wait}}, & \text{if } \|\mathbb{S}^e_C, \mathbb{S}_{in}\| < \|\mathbb{S}_{in}, \mathcal{R}_{\text{near}}\| \\ \mathcal{R}_{\text{near}}, & \text{otherwise} \end{cases} \tag{7.6}$$

Globale Strategien analysieren zur Bestimmung des aktuellen Zustands der Kommunikation die gesamte Spezifikation. Deshalb betrachten sie alle Zustände gleichermaßen, um alle Optionen für Resumption zu berücksichtigen.

Unique-Event Die erste betrachtete globale Strategie, *Unique-Event* (dt.: *eindeutiges Ereignis*, $\mathcal{R}_{\text{u-e}}$ (7.7)), nimmt die Verifikation wieder auf, wenn ein eindeutiges Ereignis auftritt. Diese Erweiterung ist für das Beispiel in Abbildung 7.12 illustriert. Ein Ereignis ist eindeutig, wenn alle Transitionen, an denen es akzeptiert wird, zu dem gleichen Zustand führen. Da es somit nur genau einen Zielzustand im Automaten gibt, betrachtet die Strategie dies als eine Art *Synchronisationspunkt*. Für nicht-eindeutige Ereignisse wechselt sie in einen Wiederaufnahmezustand ($s_{\mathcal{R}}$), der diese Ereignisse ignoriert. $\mathcal{R}_{\text{u-e}}$ ist der einzige untersuchte $\mathcal{R}$, der unabhängig von den Zuständen, die als Parameter übergeben

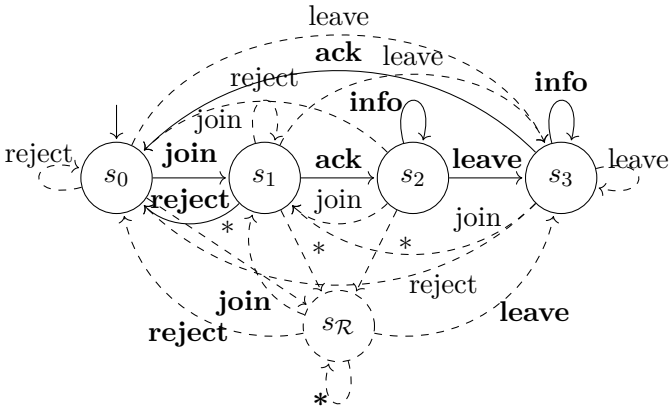


Abb. 7.12: Beispiel eines Automaten erweitert mit $\mathcal{R}_{u\text{-}e}$.

werden, arbeitet. Eine statische Berechnung benötigt daher nur einen einzelnen zusätzlichen Zustand ($s_{\mathcal{R}}$).

$$\mathcal{R}_{\text{u-e}}(\mathbb{S}_{in}, e) = \begin{cases} \delta_{\mathcal{L}}(\mathbb{S}_C, e), & \text{if } |\delta_{\mathcal{L}}(\mathbb{S}_C, e)| = 1 \\ \{s_{\mathcal{R}}\}, & \text{otherwise} \end{cases} \tag{7.7}$$

Unique-Sequence Da eindeutige Ereignisse nicht in jeder Spezifikation auftauchen oder nicht unbedingt regelmäßig beobachtet werden, erweitert *Unique-Sequence* (dt.: *eindeutige Sequenz*, $\mathcal{R}_{\text{u-s}}$ (7.8)) die vorherige Strategie um eindeutige Sequenzen von Ereignissen. $\mathcal{R}_{\text{u-s}}$ folgt dazu allen gültigen Pfaden simultan und nimmt die Verifikation wieder auf, wenn exakt ein Zielzustand mit der beobachteten Sequenz erreichbar ist. Ähnlich wie die im modellbasierten Testen verwendeten Leitsequenzen [San05], zielt $\mathcal{R}_{\text{u-s}}$ darauf ab, die Ungewissheit über den aktuellen Zustand mit jedem Schritt zu reduzieren. Dazu überprüft die Strategie, welche der als Parameter übergebenen Zustände das Ereignis akzeptieren und gibt die Zielzustände der jeweiligen Transitionen aus $\delta_{\mathcal{L}}$ aus.

$$\mathcal{R}_{\text{u-s}}(\mathbb{S}_{in}, e) = \begin{cases} \delta_{\mathcal{L}}(\mathbb{S}_{in}, e), & \text{if } \delta_{\mathcal{L}}(\mathbb{S}_{in}, e) \neq \emptyset \\ \delta_{\mathcal{L}}(\mathbb{S}_C, e), & \text{else if } \delta_{\mathcal{L}}(\mathbb{S}_C, e) \neq \emptyset \\ \{s_{\mathcal{R}}\}, & \text{sonst} \end{cases} \tag{7.8}$$

Ist das beobachtete Ereignis Teil einer Trennsequenz, dann werden die nicht passenden Zustände aus der Menge der Kandidaten entfernt. Wenn eine Zusammenführungssequenz gefunden wurde, fallen zwei oder mehr der Zielzustände zusammen und reduzieren die Anzahl der Kandidaten weiter. Somit erkennt $\mathcal{R}_{\text{u-s}}$ jede Leitsequenz für $\mathcal{L}$, die das SuO ausgibt. Unerwartetes Verhalten ergibt sich per Definition daraus, dass $\delta_{\mathcal{L}}(\mathbb{S}_{in}, e)$ leer ist. In diesem Fall wird die Resumption zurückgesetzt, indem alle Zustände wieder in die Menge der möglichen Kandidaten aufgenommen werden. Die Ungewissheit über den aktuellen Zustand wird durch neue Zustände erfasst. Beispielsweise steht der Zustand 's_2, s_3' in Abbildung 7.13 für die

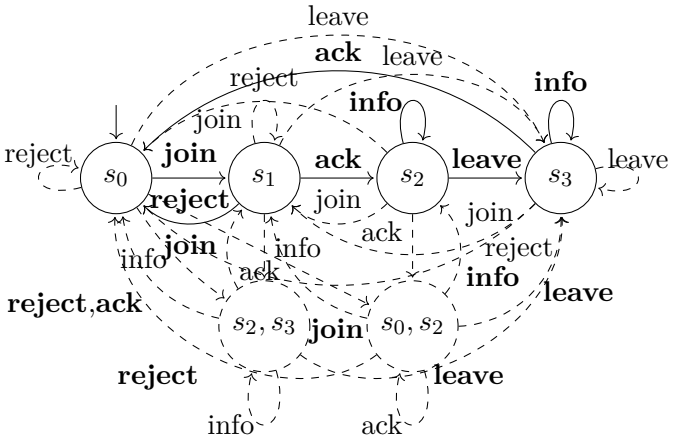


Abb. 7.13: Beispiel eines Automaten erweitert mit $\mathcal{R}_{u\text{-}s}$.

Annahme von $\mathrm{M}_{\mathcal{E}}$, dass das SuO sich in Zustand s_2 oder s_3 befindet. Daher ist die Größe des Zustandsraums durch $|A^{\mathcal{P}}|$ begrenzt.

Durch einen Vergleich mit dem Ergebnis der formalen Analyse in Abschnitt 5.1.2 zeigt sich, dass $\mathcal{R}_{\text{u-s}}$ noch nicht der allgemeinste Fall ist. Wie die anderen Strategien kann sie nur zuverlässig verwendet werden, solange die Abweichungen zu ihrem Fehlermodell passen. Die Wiederaufnahme mit $\mathcal{R}_{\text{u-s}}$ nutzt aber immer bereits das letzte Ereignis des vorherigen unerwarteten Verhaltens als erstes Ereignis in der neuen eindeutigen Sequenz. Dadurch kann die Überprüfung getäuscht werden. Denn dies entspricht einer Abweichung zu dem Zielzustand einer akzeptierenden Transition statt zu einem beliebigen Zustand.

Expected-Behavior Die Strategie *Expected-Behavior* (dt.: *erwartetes Verhalten*, $\mathcal{R}_{\text{e-b}}$ (7.9)) berücksichtigt genau das erwartete Verhalten. Sie ist die Übersetzung von der in Gleichung 5.4 definierten Funktion δ^{+}. Wie für das Beispiel in Abbildung 7.14 dargestellt, wird im Fall eines Verstoßes immer der Zustand $s_{\mathbb{S}}$ betreten. Dies spiegelt die Annahme wieder, dass das SuO sich nach einer Abweichung in einem beliebigen Zustand befinden kann.

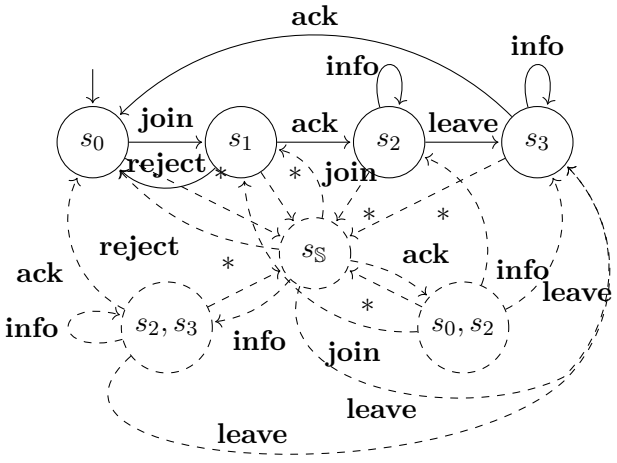


Abb. 7.14: Beispiel eines Automaten erweitert mit $\mathcal{R}_{e\text{-}b}$.

$$\mathcal{R}_{\text{e-b}}(\mathbb{S}_{in}, e) = \begin{cases} \delta_{\mathcal{L}}(\mathbb{S}_{in}, e), & \text{if } \delta_{\mathcal{L}}(\mathbb{S}_{in}, e) \neq \emptyset \\ \mathbb{S}_C, & \text{otherwise} \end{cases} \tag{7.9}$$

Theorem 7 weist darauf hin, dass mehrere eindeutige Sequenzen an erwartetem Verhalten die Zuverlässigkeit eines Monitors bei der Identifikation von Abweichungen erhöhen kann. Als Beispiel wird in der Evaluation die Strategie $\mathcal{R}_{\text{2-e-b}}$ betrachtet, die zwei solcher Sequenzen verwendet.

In diesem Abschnitt wurde eine Vielfalt an Resumption-Strategien vorgestellt. Diese Aufzählung ist nicht abschließend. Weitere Strategien können erstellt werden, um bestimmte Anforderungen zu erfüllen. Das vorgestellte Evaluierungssetup kann auf diese ebenfalls angewendet werden, um ihre Stärken und Schwächen zu identifizieren und sie mit den anderen Strategien zu vergleichen. Damit kann für einen bestimmten Anwendungsfall die beste Strategie identifiziert werden.

7.2.3 Ergebnisse

Für das Beispiel aus Abschnitt 7.2.2 ergab die Evaluation von 8000 zufällig erzeugten Traces mit je 20 injizierten Abweichungen folgende F_1 Score: $\mathcal{R}_{\text{wait}} \mapsto 0{,}6$; $\mathcal{R}_{\text{near}} \mapsto 0{,}7$; $\mathcal{R}_{\text{n-o-w}} \mapsto 0{,}8$; $\mathcal{R}_{\text{u-e}} \mapsto 0{,}8$; $\mathcal{R}_{\text{u-s}} \mapsto 0{,}8$; $\mathcal{R}_{\text{e-b}} \mapsto 0{,}99$; $\mathcal{R}_{\text{2-e-b}} \mapsto 0{,}99$; $\mathcal{R}_{\text{l-c}} \mapsto 0{,}93$. Um der höheren Genauigkeit der Ergebnisse der letzten drei Strategien zu entsprechen, wurden 12000 weitere zufällige Traces für diese ausgewertet.

Um einen möglichst allgemeingültigen Vergleich als Grundlage zur Einordnung der vorgestellten Resumption-Strategien zu bieten, werden im Folgenden die Ergebnisse der Anwendung des Evaluierungssetups für diese Strategien und mit Vielzahl an (synthetischen) Anwendungsfällen präsentiert. Dabei wurden Traces mit insgesamt 80 Millionen Abweichungen in 220 verschiedenen Automaten, die bis zu 360 Zustände haben, erzeugt und durch Monitore analysiert, die mit den Resumption-Strategien erweitert wurden. Jeder Trace enthielt 20 injizierte Abweichungen. Damit liegt die *Vollständigkeit* eines Monitors, der exakt nur die erste Abweichung meldet bei 0,05 und seine F_1 Score bei 0,095. Abbildung 7.15 zeigt *Genauigkeit* und *Vollständigkeit* für jede Strategie pro Art der Abweichung.

Die Strategie $\mathcal{R}_{\text{wait}}$ hat die niedrigste Genauigkeit für viele Arten von Abweichungen, jedoch durchweg hohe Werte für die Vollständigkeit. Für Abweichungen durch überflüssige Ereignisse erzielte sie ein perfektes Ergebnis. Von dieser Ausnahme abgesehen, sind die Ergebnisse für Abweichungen durch überflüssige und veränderte Ereignisse sehr ähnlich. Im direkten Vergleich von $\mathcal{R}_{\text{near}}$ und $\mathcal{R}_{\text{n-o-w}}$ ergibt sich für den Erstgenannten eine etwas geringere Genauigkeit, dafür aber eine höhere Vollständigkeit. $\mathcal{R}_{\text{u-e}}$ erzielt, unabhängig von der Art der Abweichung, eine geringe Vollständigkeit aber auch eine gute Genauigkeit bei ausgelassenen Ereignissen. Generell eine hohe Vollständigkeit liefert $\mathcal{R}_{\text{u-s}}$.

Auch wenn dies Nahe an die Grenzen der Signifikanz der durchgeführten Evaluation kommt, ist eine leichte Verbesserung der Genauigkeit zur Erkennung von Abweichungen von $\mathcal{R}_{\text{e-b}}$ (0,9878) mit $\mathcal{R}_{\text{2-e-b}}$ durch Erwarten einer zweiten eindeutigen Sequenz zu 0,9995 sichtbar. Der Out-of-Service-Algorithmus $\mathcal{R}_{\text{l-c}}$ erzielt ein perfektes Ergebnis für Abweichungen durch überflüssige oder veränderte Ereignisse. Auch ausgelassene Ereignisse liefert er mit hoher Zuverlässigkeit.

Abbildung 7.16 vergleicht den F_1 Score der Strategien für verschiedene Stufen von Eindeutigkeit und Größen der generierten Automaten. Zur besseren Übersicht wurden die Automaten in Gruppen entsprechend der Metriken zusammengefasst und für jede Gruppe der Mittelwert gebildet. Dies ermöglicht einen schnellen Vergleich der Strategien, aber es versteckt die eigentliche Verteilung über die ausgewerteten Automaten. Diese Details können in den Streudiagrammen Abbildung 7.17 und Abbildung 7.18 betrachtet werden.

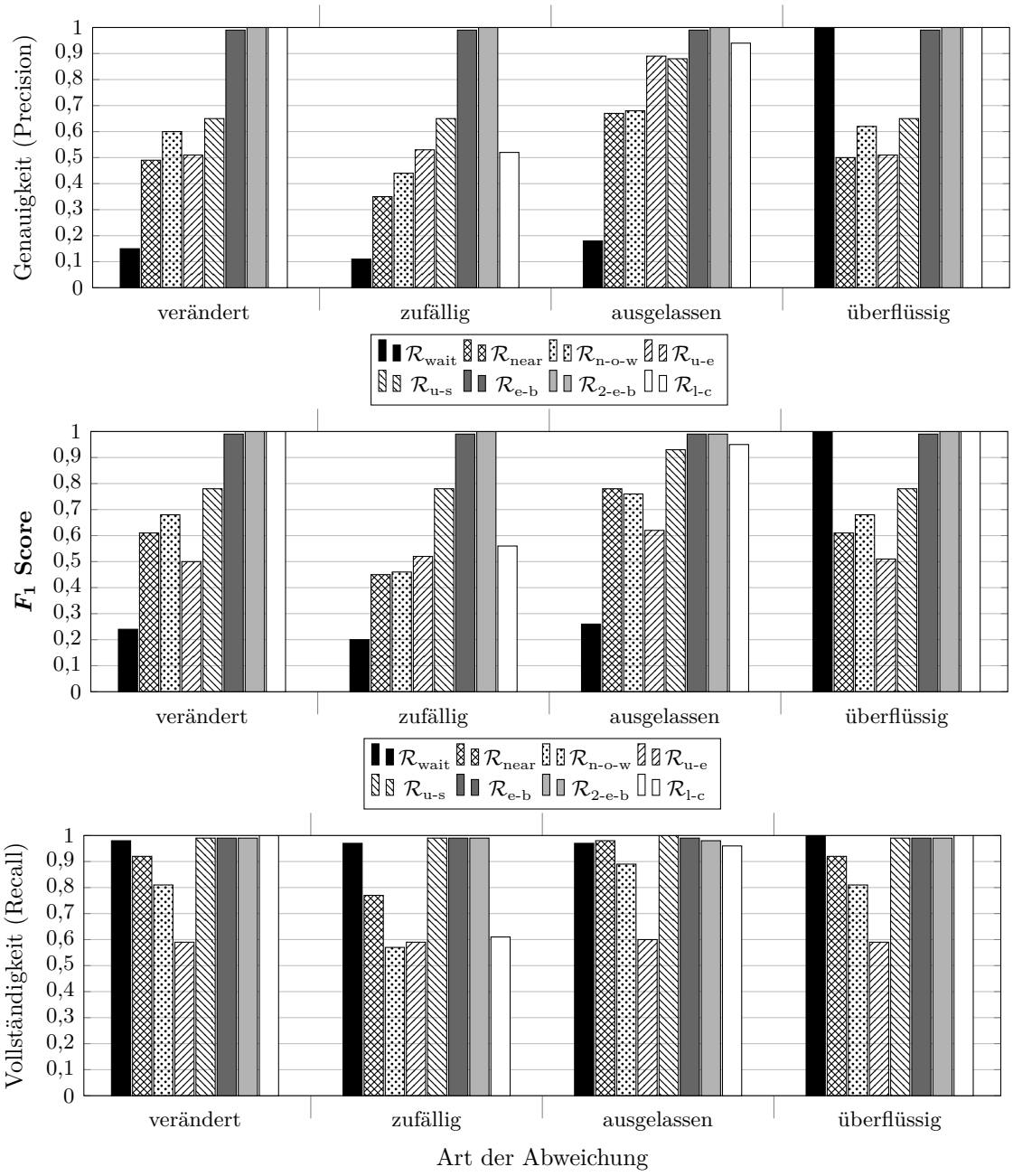


Abb. 7.15: Vergleich der erzielten Ergebnisse der mit den Resumption-Strategien erzeugten Monitoren für unterschiedliche Arten von Abweichungen. (Höher ist besser.)

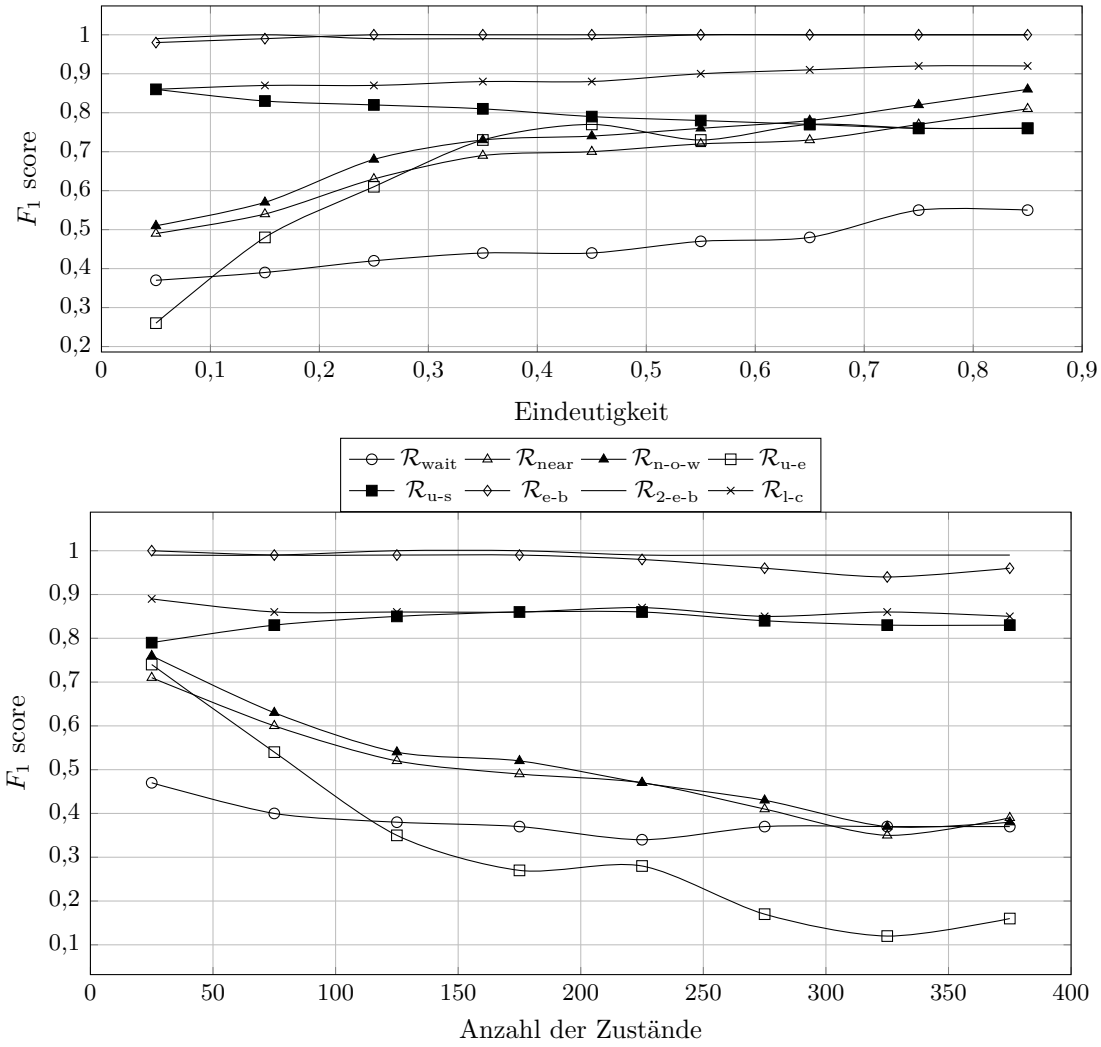
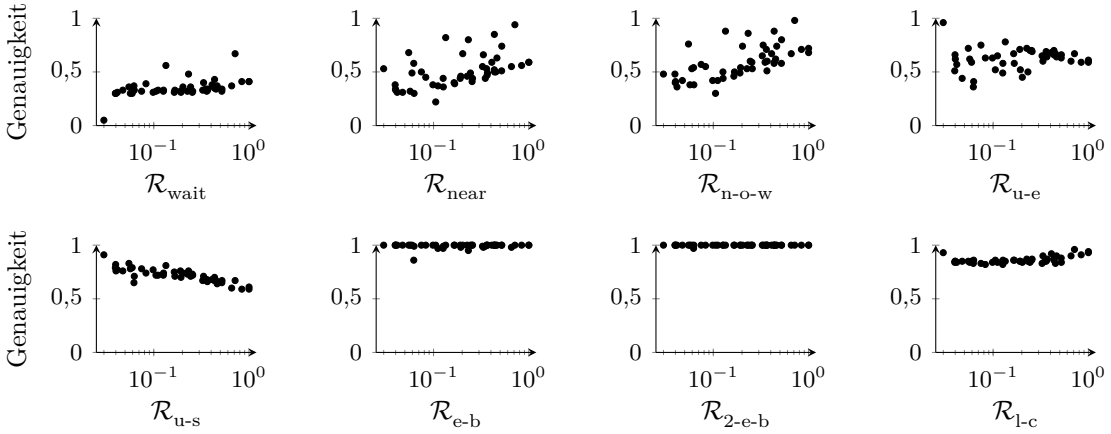
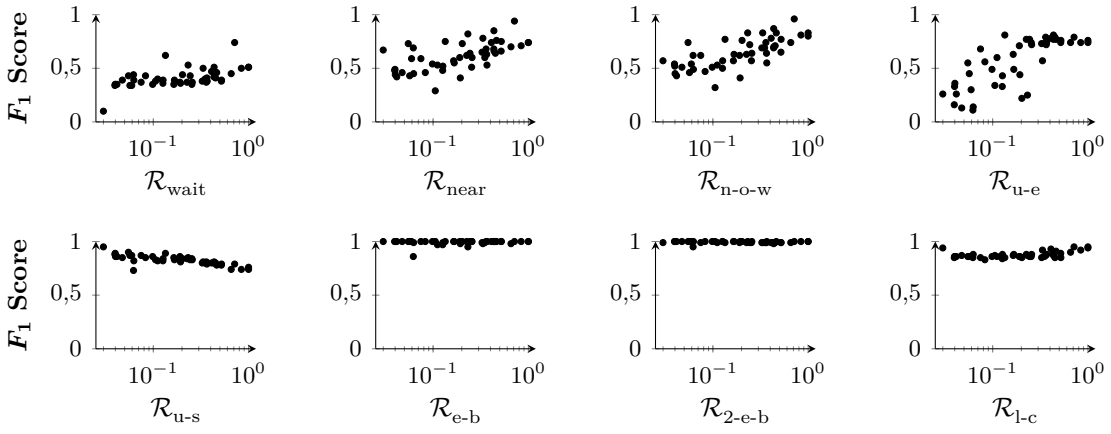


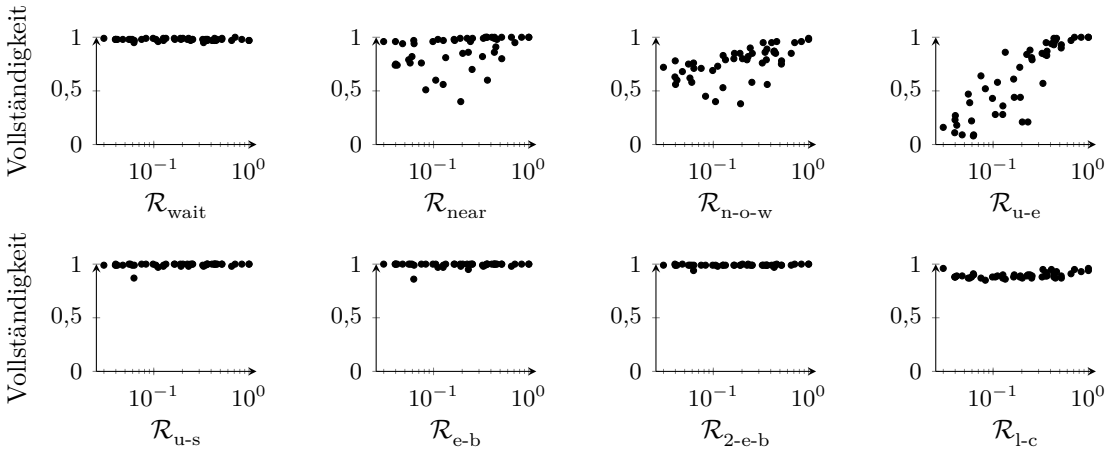
Abb. 7.16: F_1 Score der Resumption-Strategien im Vergleich nach Aufbau der Automaten entsprechend der Metriken Eindeutigkeit und Anzahl der Zustände. (Höher ist besser.)



(a) X-Achse: Eindeutigkeit, Y-Achse: Genauigkeit

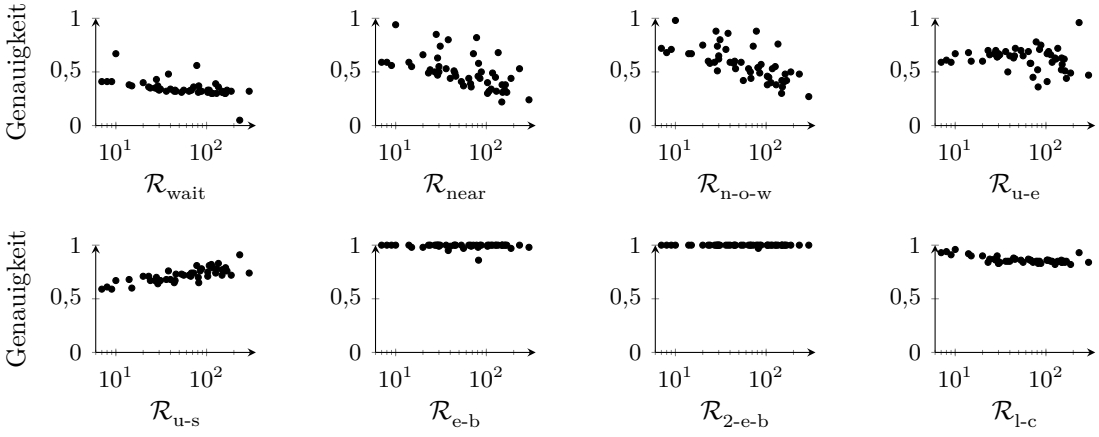


(b) X-Achse: Eindeutigkeit, Y-Achse: $\boldsymbol{F_1}$ **Score**

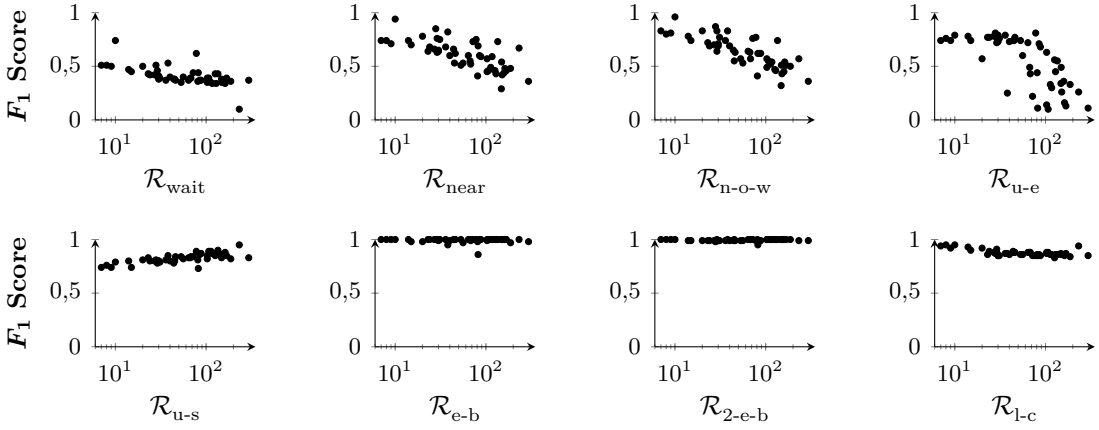


(c) X-Achse: Eindeutigkeit, Y-Achse: Vollständigkeit

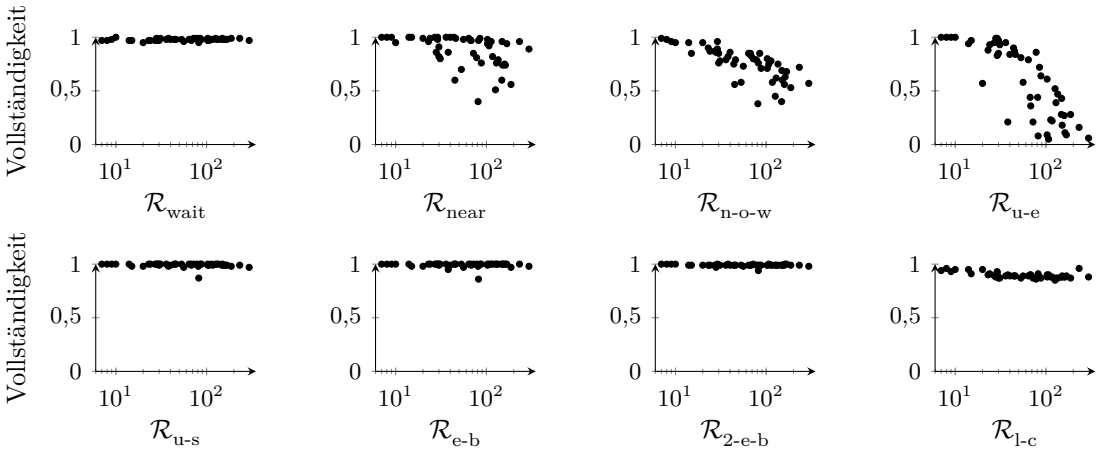
Abb. 7.17: Streudiagramme der Ergebnisse der Resumption-Strategien mit der Eindeutigkeit der betrachteten Automaten auf der X-Achse. Die Y-Achse zeigt jeweils die Genauigkeit, die Vollständigkeit und den F_1 Score. Jeder Punkt repräsentiert das Ergebnis aller Auswertungen eines mit der angegebenen Strategie erstellten Monitors. (Höher ist besser.)



(a) X-Achse: Anzahl der Zustände, Y-Achse: Genauigkeit



(b) X-Achse: Anzahl der Zustände, Y-Achse: $\boldsymbol{F_1}$ **Score**



(c) X-Achse: Anzahl der Zustände, Y-Achse: Vollständigkeit

Abb. 7.18: Streudiagramme der Ergebnisse der Resumption-Strategien mit der Anzahl der Zustände der betrachteten Automaten auf der X-Achse. Die Y-Achse zeigt jeweils die Genauigkeit, die Vollständigkeit und den F_1 Score. Jeder Punkt repräsentiert das Ergebnis aller Auswertungen eines mit der angegebenen Strategie erstellten Monitors. (Höher ist besser.)

Zum Beispiel liefert $\mathcal{R}_{\text{2-e-b}}$ für alle getesteten Fälle ein fast perfektes Ergebnis, wohingegen die Ergebnisse für $\mathcal{R}_{\text{near}}$ deutlich schwanken. Für beide Metriken sind die überall niedrigen Werte für $\mathcal{R}_{\text{wait}}$ klar sichtbar. $\mathcal{R}_{\text{u-s}}$ liefert in Zustandsautomaten mit niedriger Eindeutigkeit ein besseres Ergebnis als viele andere Strategien. Der F_1 Score nimmt sogar leicht mit zunehmender Eindeutigkeit ab. Viele der anderen Strategien profitieren von einer Zunahme der Eindeutigkeit, insbesondere $\mathcal{R}_{\text{u-e}}$. Bei sehr hoher Eindeutigkeit sind $\mathcal{R}_{\text{u-s}}$ und $\mathcal{R}_{\text{u-e}}$ identisch. Dennoch sind sowohl $\mathcal{R}_{\text{n-o-w}}$ als auch $\mathcal{R}_{\text{near}}$ in diesem Fall besser. Eine Zunahme in der Anzahl der Zustände führt zu einem Leistungsrückgang der Strategien $\mathcal{R}_{\text{u-e}}$, $\mathcal{R}_{\text{n-o-w}}$ und $\mathcal{R}_{\text{near}}$. $\mathcal{R}_{\text{u-e}}$ fällt sogar unter die Werte von $\mathcal{R}_{\text{wait}}$.

$\mathcal{R}_{\text{e-b}}$ und $\mathcal{R}_{\text{2-e-b}}$ liefern ein nahezu perfektes Ergebnis für alle getesteten Automaten und sind kaum durch deren Anzahl an Zuständen und Eindeutigkeit beeinflusst. Der leichte Vorteil von $\mathcal{R}_{\text{2-e-b}}$ wird etwas deutlicher erkennbar bei der leichten Verschlechterung von $\mathcal{R}_{\text{e-b}}$ für niedrige Eindeutigkeit und für eine hohe Anzahl an Zuständen. Der Out-of-Service-Algorithmus $\mathcal{R}_{\text{l-c}}$ kann diese Leistung nicht ganz erreichen, aber ist besser als die übrigen Strategien. Ab einer Anzahl von ca. 100 Zuständen ist der F_1 Score von $\mathcal{R}_{\text{l-c}}$ mit dem von $\mathcal{R}_{\text{u-s}}$ vergleichbar.

Die Streudiagramme in Abbildung 7.17 und Abbildung 7.18 zeigen das durchschnittliche Ergebnis der vorgestellten Resumption-Strategien für jeden Automaten. Ein Datenpunkt in einem dieser Streudiagramme entspricht dem Ergebnis eines erweiterten Monitors, gemittelt über alle analysierten Traces eines Automaten. In den Streudiagrammen ist gut zu erkennen, dass die lokalen Strategien ($\mathcal{R}_{\text{wait}}$, $\mathcal{R}_{\text{near}}$ und $\mathcal{R}_{\text{n-o-w}}$) sowie $\mathcal{R}_{\text{u-e}}$ eine hohe Streuung der Ergebnisse je Automat haben. Ebenfalls ist die durch Abbildung 7.16 nahegelegte Ähnlichkeit des F_1 Scores von $\mathcal{R}_{\text{u-e}}$ und $\mathcal{R}_{\text{l-c}}$ für Automaten mit vielen Zuständen in Abbildung 7.18 (b) erkennbar. Durch Vergleich der Streudiagramme zu den Strategien in (a) und (c) der Abbildung werden jedoch Unterschiede in der Genauigkeit und Vollständigkeit sichtbar.

7.2.4 Diskussion

Auf den ersten Blick sieht es so aus, dass die Resumption-Strategien $\mathcal{R}_{\text{e-b}}$ und $\mathcal{R}_{\text{2-e-b}}$ besser Abweichungen erkennen, als das Out-of-Service Verfahren $\mathcal{R}_{\text{l-c}}$. Die Strategien $\mathcal{R}_{\text{e-b}}$ und $\mathcal{R}_{\text{2-e-b}}$ erzielten für alle getesteten Anwendungsfälle nahezu perfekte oder perfekte Ergebnisse. Das Verfahren $\mathcal{R}_{\text{l-c}}$ erlangt dies ebenfalls für Abweichungen, die zu seinen vorgesehenen Änderungsoperationen passen. Dies sind zusätzliche, fehlende und ersetzte Ereignisse. Insbesondere Abweichungen durch zufällige Ereignisse werden von $\mathcal{R}_{\text{l-c}}$ nicht immer richtig erkannt. Bei diesen Abweichungen kann beispielsweise das SuO in einen ganz anderen Abschnitt des Automaten springen oder Zustände aktivieren, die nicht (mehr) über Transitionen vom aktuellen Zustand erreichbar sind. Dies passt nicht zu dem Fehlermodell,

für das $\mathcal{R}_{\text{l-c}}$ entwickelt wurde. Das Verfahren kann solche Abweichungen nur behandeln, indem es mehrere Schritte zurückgeht und weitere Änderungen einfügt. Hierdurch wird an anderer Stelle die Änderung, also die Abweichung, gemeldet. Im Gegensatz dazu können $\mathcal{R}_{\text{e-b}}$ und $\mathcal{R}_{\text{2-e-b}}$ solche Sprünge durch zufällige Ereignisse direkt erfassen. $\mathcal{R}_{\text{l-c}}$ gibt dafür einen möglichst ähnlichen, gültigen Trace zu den Beobachtungen zurück.

Generell ist zu beachten, dass immer nur unerwartetes Verhalten erkannt werden kann. Diese Evaluation untersucht, in wieweit das gemeldete unerwartete Verhalten mit den Abweichungen übereinstimmt. Wie in Abschnitt 5.1.3 erläutert, gibt es Situationen, in denen Abweichungen nicht erkannt werden können. Dies ist genau dann der Fall, wenn nicht genügend Informationen vorliegen, damit die Abweichung für einen Beobachter offensichtlich wird. Beispielsweise bei zwei kurz hintereinander auftretenden Abweichungen kann es für die zweite Abweichung noch gültige Pfade geben und es wird kein unerwartetes Verhalten erkannt. Dieses tritt dann entweder erst bei einem späteren Ereignis oder gar nicht auf. Deshalb können die Strategien auch nicht immer ein perfektes Ergebnis erzielen. Die leichte Verbesserung mit $\mathcal{R}_{\text{2-e-b}}$ bekräftigt, dass eine höhere Genauigkeit bei der Erkennung des abweichenden Ereignisses erreicht werden kann, wenn mehr als eine eindeutige Sequenz für die Wiederaufnahme der Verifikation erfordert wird. Dies kann allerdings zulasten der Vollständigkeit gehen. Da für die Erkennung mehrerer eindeutiger Sequenzen auch mehr Ereignisse zwischen den Abweichungen notwendig sind, kann der Monitor mehr Abweichungen übersehen.

Während $\mathcal{R}_{\text{e-b}}$ beständige Ergebnisse liefert, kann es effizientere Strategien geben. Insbesondere dann, wenn genauere Informationen über die zu erwarteten Abweichungen vorliegen. So sind die perfekte Genauigkeit und Vollständigkeit von $\mathcal{R}_{\text{wait}}$ für Abweichungen durch überflüssige Ereignisse wie erwartet, denn sie passen exakt zu dem Verhalten der Strategie bei der Wiederaufnahme. Dies zeigt, dass Wissen über die Arten der Abweichungen, die in einem Anwendungsfall erwartete werden, bei der Formulierung von hocheffizienten Strategien helfen kann. Ebenso macht es deren Spezialisierung deutlich, denn $\mathcal{R}_{\text{wait}}$ erzielt für alle anderen Abweichungen das schlechteste Ergebnis. Während das SuO intern in einen neuen Zustand wechselt, bleibt der Monitor mit $\mathcal{R}_{\text{wait}}$ im Vorherigen. Das SuO müsste zu diesem Zustand zurückkehren, damit der Monitor wieder den richtigen Zustand kennt. Die Strategie profitiert dabei (wie die meisten anderen) von einer hohen Eindeutigkeit des Automaten, da dies die Wahrscheinlichkeit erhöht, dass er auch von einem falschen Zustand aus in den Richtigen gelangt. $\mathcal{R}_{\text{wait}}$ wird oft (unbeabsichtigt) verwendet, da er der Fortführung der Überprüfung ohne Wiederaufnahme entspricht. Abweichende Ereignisse werden einfach ignoriert und im gleichen Zustand gewartet. Für den allgemeinen Fall ist dies nicht immer korrekt und in der Regel sehr unzuverlässig. Wenn jedoch Abweichungen nur überflüssig sind – das SuO bleibt beim Abweichen im gleichen Zustand – bietet $\mathcal{R}_{\text{wait}}$ eine perfekte Resumption und Erkennung von Abweichungen. Der Beweis ist einfach, da

die aktuelle Zustandsunsicherheit immer genau einen Zustand enthält. Dadurch kann Theorem 1 angewendet werden.

Die Metrik *Eindeutigkeit* kann als Indikator verwendet werden, wie komplex die Strategie für den Anwendungsfall sein sollte. Bei niedrigen Werten muss die Strategie in der Lage sein, mehrere Ereignisse zu kombinieren, um Monitor und SuO zuverlässig zu synchronisieren, beispielsweise $\mathcal{R}_{\text{u-s}}$. $\mathcal{R}_{\text{u-s}}$ verliert jedoch mit zunehmender Eindeutigkeit an Genauigkeit. Dies erklärt sich durch die zunehmende Wahrscheinlichkeit, dass eine Abweichung ein eindeutiges Ereignis verwendet. Wenn im Extremfall alle Ereignisse eindeutig sind, dann wären auch alle beobachteten Ereignisse von Abweichungen eindeutig und die Strategie würde diese zur Wiederaufnahme verwenden. Im Fehlermodell ist aber nicht festgelegt, dass das SuO nach einer Abweichung mit einem eindeutigen Ereignis auch in den bei normaler Operation erreichten Zustand eintritt. Da das nächste Ereignis ebenfalls eindeutig ist, entdeckt der Monitor diesen Unterschied und meldet einen weiteren Verstoß. Damit werden aber zwei Abweichungen statt nur einer vom Monitor gemeldet, wodurch sich die Genauigkeit reduziert. Das gleiche gilt für $\mathcal{R}_{\text{u-e}}$. $\mathcal{R}_{\text{e-b}}$ ist davon nicht betroffen, da nach unerwartetem Verhalten immer alle Zustände als Möglichkeit betrachtet werden. Entspricht das unerwartete Verhalten der Abweichung, wird damit das zugehörige (nicht spezifizierte) Ereignis ignoriert und erst mit den folgenden Ereignissen der Zustand des SuO ermittelt. Ein Bias zu einer niedrigeren Eindeutigkeit bei zunehmender Anzahl an Zuständen in den generierten Automaten, hat entsprechend einen negativen Einfluss auf das Ergebnis von $\mathcal{R}_{\text{u-e}}$. Dieser Bias wird durch die diagonale Anordnung der Punkte in Abbildung 7.19 deutlich. Sie zeigt ein Streudiagramm der Verteilung von Eindeutigkeit und Anzahl der Zustände der evaluierten Automaten.

Insbesondere bei sehr hoher Eindeutigkeit kann eine weitere Einschränkung der Möglichkeiten, an welchen Stellen die Wiederaufnahme ansetzen darf, wünschenswert sein, um eine Wiederaufnahme an einer falschen Stelle zu vermeiden. Die lokalen Strategien verwenden dazu den vorher aktiven Zustand. Die Auswahl zwischen $\mathcal{R}_{\text{near}}$ and $\mathcal{R}_{\text{n-o-w}}$ ist eine Abwägung zwischen höherer Vollständigkeit oder Genauigkeit. Der F_1 Score der beiden Strategien ist in Abbildung 7.16 nahezu identisch, der von $\mathcal{R}_{\text{n-o-w}}$ liegt oberhalb des Ergebnisses für $\mathcal{R}_{\text{near}}$. Diese Strategien können sich aber, da sie lokal um den aktiven Zustand arbeiten, in falsche Bereiche des Automaten oder in eine Sackgasse manövrieren und sind deshalb weniger geeignet für Automaten mit einer größeren Anzahl an Zuständen. Der F_1 Score der erweiterten Monitore, die mehr als eine Abweichung melden, ist in allen evaluierten Fällen besser als der eines einfachen Monitors, der nur die erste meldet.

Mit dem vorgestellten Evaluierungssetup kann die am besten geeignete Resumption-Strategie für einen individuellen Anwendungsfall ermittelt werden. Die Ergebnisse für das Dienst-Beispiel (Eindeutigkeit 0,43, 4 Zustände) und die entsprechenden Ergebnisse

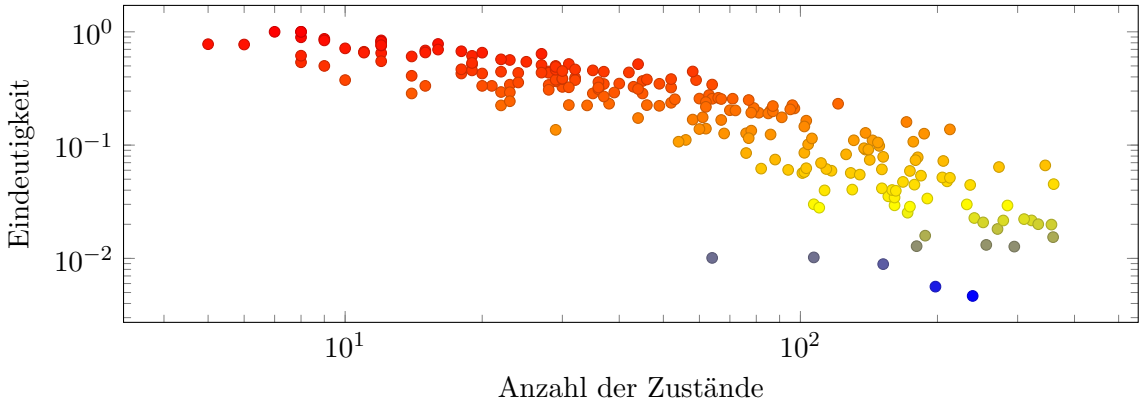


Abb. 7.19: Streudiagramm der Verteilung von Eindeutigkeit und Anzahl der Zustände in den für die Evaluation verwendeten Automaten. Jeder Punkt entspricht einem Automaten.

aus Abbildung 7.16 stimmen gut überein und geben damit ein weiteres Beispiel dafür, dass die Metriken Eindeutigkeit und Anzahl der Zustände für eine grobe Vorabschätzung herangezogen werden können. Normalerweise sollte die Wahl auf eine der Strategie fallen, die Sequenzen betrachtet, auch wenn dies etwas mehr Platz und damit Ressourcen benötigt. Nur bei hoher Eindeutigkeit des Automaten erreichen andere der evaluierten Strategien ähnliche Bewertungen.

Die größte Schwierigkeit bei der Resumption, die in dieser Arbeit aufgezeigt wird, besteht darin, dass die Unsicherheit über den aktuellen Zustand nur durch eine offensichtlich konforme Zusammenführungssequenz nachweislich reduziert werden kann und es keine Garantie dafür gibt, dass eine solche Sequenz existiert. Die Auswertung hat zwar gezeigt, dass es in vielen Fällen immer noch möglich ist, Abweichungen mit hoher Wahrscheinlichkeit durch Resumption zu identifizieren, es kann aber Fälle geben, in denen es unmöglich ist, Abweichungen zu erkennen. Da ohne weitere Informationen diese Abweichungen nicht erkannt werden können, gibt es hierzu bereits verschiedene Ansätze. Einige betrachten bestimmte Ereignisse als vertrauenswürdig, z. B, modellbasiertes Testen [PL05] kann sich auf die Ereignisse verlassen, die es dem SuO als Input liefert. Wenn solche zuverlässigen Ereignisse in die Spezifikation integriert werden, können sie als eine Art Kontrollpunkt dienen, um die Verifikation definiert wieder aufzunehmen. Ein weiteres Beispiel ist ein festgelegter Anfangszustand. Andere Ansätze erwarten, dass die Spezifikation in Merkmale aufgeteilt wird, die einzeln beschrieben werden, beispielsweise mit LTL [BLS11]. Sie sind darauf angewiesen, als Auslöser eine bestimmte Sequenz zu erkennen, bevor sie eine Beschränkung verifizieren. Ihr Verdikt entscheidet damit immer über Auslöser und Randbedingung. Daher wird häufig nicht zwischen beiden unterschieden. Außerdem können diese Auslöser recht komplex werden und z. B. auf notwendige oder hinreichende Bedingungen des Merkmals prüfen. Gibt es viele zu prüfende Merkmale, können die Auslöser schnell redundant werden.

Daher kann es effizienter sein, einen einzigen Zustandsautomaten zu verwenden. Wie gezeigt werden konnte, kann dieser alles unerwartete Verhalten erkennen. Unterschiedliche Verdikte können mit bestimmten (fehlenden) Zustandsübergängen in Verbindung gebracht und so dazu verwendet werden, die Kategorisierung von unerwartetem Verhalten, das durch eine Reihe von Merkmalen bereitgestellt wird, beizubehalten.

Resumption strebt die Wiederaufnahme der Verifizierung an, nachdem eine Abweichung beobachtet wurde. Im Allgemeinen hängt die Qualität der Resumption von einer sorgfältigen Auswahl der Kandidaten für den tatsächlichen Zustand des SuO ab. Daraus ergibt sich eine starke Beziehung zu den Abweichungen, die möglich oder zu erwarten sind. Unterschiedliche Annahmen über das System, z. B. die Art der Abweichungen, führen zu einer anderen Auswahl der optimalen Kandidaten. Die Auswahl der Kandidaten erfolgt durch eine *Resumption-Strategie*. Dieser ersetzt δ^+ aus Gleichung 5.4 durch eine andere Funktion. δ^+ bestimmt die aktuelle Zustandsunsicherheit für den nächsten Schritt. Wenn eine andere Strategie gewählt wird, bleiben die in dieser Arbeit aufgestellten Theoreme gültig, solange alle Abweichungen diesem Fehlermodell entsprechen. Jede Abweichung, die nicht mit den von δ^+ gelieferten Kandidaten übereinstimmt, bricht jedoch beispielsweise die Garantie, alle unerwarteten Verhaltensweisen zu erkennen. Dennoch erlaubt dies die Verwendung spezialisierter Strategien für spezifische Anwendungsszenarien.

7.2.5 Bedrohungen der Validität

Zum Abschluss des zweiten Teils der Evaluation wird noch auf mögliche Bedrohungen der Validität der präsentierten Ergebnisse eingegangen, begonnen mit ihrer Allgemeingültigkeit. Für den Vergleich der Resumption-Strategien wurden zufällig erzeugte Modelle und Abweichungen herangezogen. Die Erzeugung der Modelle wurde so gestaltet, dass die resultierenden Automaten eine ähnliche Struktur aufweisen, wie sie von manuell erstellten Verhaltensbeschreibungen, die schrittweise verfeinert werden, erwartet wird. Durch die über 200 verschiedenen Automaten ist auch eine Vielfältigkeit gesichert. Eine gezielte Auswahl der Automaten erfolgte nicht, wie beispielsweise auch an dem Bias zu niedrigerer Eindeutigkeit bei zunehmender Anzahl von Zuständen sichtbar wird. Viele der Ergebnisse sind nur als Mittelwerte angegeben, um den generellen Trend aufzuzeigen. Die Verteilungen der Ergebnisse können aber detailliert in Abbildung 7.17 und Abbildung 7.18 nachvollzogen werden. Um Implementierungsfehler auszuschließen, wurden die Module einzeln getestet und Plausibilitätsprüfungen in die Implementierung mit aufgenommen. Beispielsweise, dass die erste Abweichung stets exakt gefunden wird. Zudem wurden die Resultate von Vorläufen manuell in Stichproben und bei auffälligen Abweichungen von den restlichen oder erwarteten Ergebnissen überprüft. Die hier vorgestellten Ergebnisse wurden erst erfasst, nachdem alle Auffälligkeiten der Vorläufe plausibilisiert werden konnten.

7.3 Fazit

In diesem Kapitel wurden Nutzen und Verwendbarkeit des Ansatzes in verschiedenen Anwendungsbeispielen demonstriert. Dabei wurde auch gezeigt, wie der modulare Ansatz eine schrittweise Erweiterung der Methodik ermöglicht hat, um auf die verschiedenen Anforderungsprofile der Szenarien angepasst zu werden. Die Methodik zur Wiederaufnahme der Überprüfung erlaubt den Einsatz von unterschiedlichen Strategien, um zu entscheiden, wann und wie die Überprüfung fortgesetzt wird. Durch den Vergleich verschiedener dieser Resumption-Strategien wurde untersucht, ob auf Basis solcher Soll-Beschreibungen alle beobachtbaren Abweichungen erkannt und gemeldet werden. Das Evaluierungssetup ermöglicht zudem für einen bestimmten Anwendungsfall, die am besten geeignete Resumption-Strategie zu ermitteln. Dies erfüllt somit Ziel 4.

Die allgemeine Untersuchung anhand von zufällig erzeugten Automaten führte zu folgenden Ergebnissen: $\mathcal{R}_{\text{wait}}$ ist ein Beispiel für eine an bestimmte Situationen optimal angepasste Strategie. Bei einem unerwarteten Ereignis bleibt der Monitor dabei im gleichen Zustand und wartet dort, dass ein erwartetes Ereignis eintritt. Die Strategie verändert die Größe des Automaten nicht, funktioniert aber ausschließlich dann, wenn nur überflüssige Ereignisse als Abweichungen auftreten. Hingegen ist die Strategie $\mathcal{R}_{\text{e-b}}$, die beliebiges erwartetes Verhalten sucht, die stabilste und zuverlässigste der verglichenen Strategien. Für die Resumption-Strategie diente die erweiterte Transitionsfunktion δ^+ als Vorlage, die bereits in Abschnitt 5.1.2 bei der theoretischen Betrachtung als allgemeiner Fall verwendet wurde. Die Laufzeitprüfung mit $\mathcal{R}_{\text{e-b}}$ entspricht somit der Segmentierung des Traces nach unerwartetem Verhalten für eine beliebige Abweichung. In der Evaluation wurde auf die exakte Identifikation der injizierten Abweichungen getestet. $\mathcal{R}_{\text{e-b}}$ konnte diese in nahezu allen Fällen exakt erkennen. Dies ist nur möglich, da die Strategie mit den Ereignissen zwischen zwei Abweichungen den aktuellen Zustand ausreichend genau bestimmen konnte.

8 Kapitel 8
Schlussbetrachtung

Vernetzte eingebettete Systeme sind heute allgegenwärtig. Um auch komplexere Dienste anbieten zu können, interagieren immer mehr Komponenten in einem System miteinander. Für die Qualität der entstehenden Applikation ist, neben der Korrektheit der einzelnen Komponenten, die geordnete Kommunikation der Komponenten untereinander entscheidend. Ein robustes System funktioniert weiter, nachdem ein unkritischer Fehler oder eine Abweichung von seiner Spezifikation aufgetreten ist. Daher kann es während einer einzigen Ausführung mehrfach von der Spezifikation abweichen. Ein Monitor, der das System beobachtet, sollte möglichst jede Abweichung einzeln identifizieren können. Daher hat diese Arbeit *Resumption* erforscht, ein neuer Ansatz, der alle erkennbaren Abweichungen zwischen der Ausführung eines System unter Beobachtung (SuO, engl.: *System under Observation*) und dessen Spezifikation mit einem einzigen Monitor finden kann, anstatt nur allgemeine nicht-Konformität zu melden.

In diesem Kapitel werden die wesentlichen Inhalte und Beiträge dieser Arbeit zusammengefasst und diskutiert. Abschließend wird ein Ausblick auf mögliche fortführende Forschungsarbeiten gegeben.

8.1 Zusammenfassung

Die Aufgabenstellung dieser Arbeit wurde in Kapitel 1 mit Herausforderungen motiviert. Es wurden Ziele und wissenschaftliche Fragestellungen definiert, um diesen zu begegnen. Auf die Erfüllung und Beantwortung dieser wird zum Ende dieser Zusammenfassung eingegangen. Die für diese Arbeit relevanten Grundlagen von Automatentheorie, modellbasierter Entwicklung und Laufzeitüberprüfung wurden in Kapitel 2 zusammengefasst.

Kapitel 3 hat den Ansatz in den aktuellen Stand der Technik zur Absicherung interaktiver Software-intensiver Systeme eingeordnet. Dabei wurde herausgestellt, dass er komplementär zu bestehenden Ansätzen ist, da er eine Möglichkeit zur Wiederaufnahme der Überprüfung

bietet, die auf beliebige mit Automaten beschreibbare Merkmale der Laufzeitüberprüfung angewendet werden kann. Dadurch entfällt die Notwendigkeit, die Spezifikation in unabhängige Merkmale aufzuteilen, die dann getrennt und wiederholt überprüft werden können.

Kapitel 4 hat einen neuen Ansatz zur Modellierung des Soll-Verhalten eines verteilten eingebetteten Systems vorgestellt, das als Grundlage für die Überprüfung dieses Systems dienen kann. Das Modell stellt dabei die Schnittstellen des Systems in den Vordergrund, da ein Kommunikationskanal häufig einfacher überwacht werden kann, als die Verarbeitung innerhalb einer Komponente.

Durch die eingeführten Erweiterungen gegenüber einfachen Automaten können neben sequentiellen Abhängigkeiten des Kontrollflusses auch die Zusammenhänge von Daten der Nachrichten kompakt beschrieben werden. Bedingungen an das Zeit-Verhalten des Systems können ebenfalls mit Kontexten und über die maximale Aktivierungsdauer jedes Zustands beschrieben werden. Eine Beschreibung mit positivem Soll-Verhalten vereinfacht zudem die Generierung von weiteren Artefakten, beispielsweise für eine Restbus Simulation oder um die Abdeckung einer Test-Sammlung zu prüfen.

In Kapitel 5 wurde untersucht, wie unerwartetes Verhalten und Abweichungen mit einem Monitor erkannt werden können. Dazu wurde der neue Ansatz zur Wiederaufnahme der Überprüfung, genannt *Resumption*, vorgestellt. Resumption ermöglicht einem Monitor, mehr als nur den ersten Verstoß gegen die Spezifikation zu erkennen. Dies wurde zuerst für einfache Automaten gezeigt und ein Verfahren zur automatisierten Erweiterung von Automaten mit Resumption wurde vorgestellt. Die Verwendung von Resumption meldet genau dann unerwartetes Verhalten, wenn die Beobachtungen nicht auf den Automaten abgebildet werden können. Für jede erkennbare Abweichung kann der Bereich des Traces abgegrenzt werden, in dem sie sich befindet.

Da das Verhalten von realen Systemen häufig komplexer ist, als mit einfachen Automaten nachvollziehbar beschrieben werden kann, wurden in Kapitel 4 Erweiterungen der Modellierung vorgestellt. Diese können auf einfache Automaten zurückgeführt werden und daher kann auch Resumption auf sie angewendet werden. Da dies zu einer (mehrfach) exponentiellen Zunahme an Zuständen führen kann, wurden in der zweiten Hälfte von Kapitel 5 Rahmenbedingungen betrachtet, unter denen der Zustandsraum für Resumption vereinfacht werden kann.

In Kapitel 6 wurde ein Überblick über die Konzepte zur Anwendung des in dieser Arbeit vorgestellten Ansatzes und die Integration in die Werkzeugplattform Dana gegeben. Dazu wurde gezeigt, wie durch den Einsatz verschiedener Modellierungssprachen, die flexible Architektur zur Datenverarbeitung und die Konfiguration der Überprüfung eine

Technologieunabhängigkeit umgesetzt wird. Für die Ausführung der Überprüfung kann durch Interpretation des Modells mit SCXML oder durch Generierung von ausführbarem C++Code erfolgen. Letzteres ermöglicht eine Reaktion auf Abweichungen im Betrieb. Um die Auswertung der Beobachtungen zu erleichtern, wurden der Debug-Mechanismus von Eclipse [EclDbg] erweitert und Möglichkeiten zur Animation der Zustandsautomaten sowie zur grafischen Darstellungen von Signalwerten geschaffen. Durch die Realisierung wird auch eine Überprüfung der Praxistauglichkeit des Ansatzes und eine Evaluierung von Resumption möglich.

Die Evaluierung erfolgte in Kapitel 7. Im ersten Teil wurden Nutzen und Verwendbarkeit des Ansatzes in verschiedenen Anwendungsbeispielen demonstriert. Zudem hat der modulare Ansatz eine schrittweise Erweiterung der Methodik ermöglicht, um auf die verschiedenen Anforderungsprofile der Szenarien zu reagieren. Durch die Hinzunahme von Instanzen und Kontexte konnten beliebig viele parallele Quellenwechsel verfolgt werden. Das Modell einer Industrieanlage konnte durch Integration von selbst-lernenden Methoden automatisch ermittelt und anschließend für die Überprüfung verwendet werden, unter Wiederverwendung der gleichen Architektur zur Datenverarbeitung. Die Erweiterung um Statistiken ermöglichte die Auswertung und Anzeige der Zeiten in einer Übertragungskette.

Im zweiten Teil der Evaluierung wurde experimentell untersucht, ob auf Basis solcher Soll-Beschreibungen alle beobachtbaren Abweichungen erkannt und gemeldet werden. Dazu wurde ein Evaluierungssetup vorgestellt, mit dem die Genauigkeit und Vollständigkeit von Monitoren bei der Erkennung von Abweichungen ermittelt werden kann. Es dient auch als Hilfestellung, um die beste Strategie für einen individuellen Anwendungsfall zu ermitteln. Es wurden unterschiedliche Strategien zur Wiederaufnahme der Überprüfung vorgestellt und ihre Ergebnisse bei der Auswertung von Traces generierter Automaten mit vier verschiedenen Fehlermodellen präsentiert und diskutiert. Dabei hat sich die Strategie, die direkt aus der theoretischen Untersuchung in Abschnitt 5.1.2 entnommen wurde, in allen Fällen als sehr robust herausgestellt. Es konnte aber auch gezeigt werden, das andere Strategien in bestimmten Situationen effizienter sein können.

8.1.1 Ziele und Herausforderungen

In Abschnitt 1.2 wurden Herausforderungen aus der Infotainment Domäne von Fahrzeugen abgeleitet, die jeder Ansatz zur Laufzeitüberprüfung solcher Systeme lösen muss. Basierend auf diesen Herausforderungen hat Abschnitt 1.3 die Ziele der vorliegenden Arbeit formuliert. Im Folgenden wird der Zusammenhang mit den entwickelten Methoden diskutiert und wie diese beitragen, die Herausforderungen zu meistern.

Aufbau eines modularen Ansatzes zur Beschreibung von Soll-Verhalten Der in Kapitel 4 vorgestellte modulare Ansatz zur Beschreibung von Soll-Verhalten begegnet den in Ziel 1 genannten Herausforderungen. Hierbei handelt es sich um eine Kombination, Anpassung und Erweiterung bestehender Ansätze, die eine Technologie-unabhängige Beschreibung (siehe Herausforderung 1), insbesondere der Interaktionen an Schnittstellen, ermöglicht. Die Beschränkung auf präzise definierte Elemente resultiert in einer klaren Semantik für die Beschreibung (siehe Herausforderung 2). Der Ansatz lindert durch Kontexte und Instanzen eine drohende Zustandsexplosion (siehe Herausforderung 3) und ermöglicht die Beschreibung nicht-funktionaler Eigenschaften wie Zeit (siehe Herausforderung 4).

Methodik zum Finden aller Abweichungen vom Soll-Verhalten zur Laufzeit Die in Kapitel 5 eingeführte Methodik zum Finden aller Abweichungen vom Soll-Verhalten zur Laufzeit begegnet den in Ziel 2 genannten Herausforderungen. Da die Erkennung von Abweichungen (Herausforderung 5) den Kern dieser Arbeit bildet, wird die Herausforderung direkt im Ziel bereits genannt. Resumption kann die Überprüfung wiederaufnehmen und eigenständig den Zustand des SuO erkennen. Dies führt dazu, dass die Überprüfung robust gegenüber unvollständigen Beschreibungen (siehe Herausforderung 6) ist. Das nicht beschriebene Verhalten wird zwar als unerwartet markiert, der Monitor findet aber selbstständig heraus, wo die Beobachtungen wieder zum Spezifizierten passt und kann so auch danach unerwartetes Verhalten erkennen.

Einige Strategien für Resumption führen zwar allgemein zu einer exponentiellen Zunahme der Zustände, die Laufzeitüberprüfung kann aber mit linearem Aufwand für Zeit und Platz den ursprünglichen Automaten interpretieren. Die Überprüfung ist somit effizient genug, dass dies insbesondere für einfache Automaten zur Laufzeit erfolgen kann und damit eine Reaktion auf Abweichungen im Betrieb (siehe Herausforderung 7) möglich wird. Auch für die erweiterten Konzepte der Modellierung wurde gezeigt, wie die Anzahl zusätzlicher Zustände, die durch Resumption hinzukommen, reduziert werden kann.

Realisierbare Methodik Im Rahmen der Integration in die Werkzeugplattform DANA, wurden weitere Konzepte entwickelt und in Kapitel 6 vorgestellt, die eine Realisierung der Methodik ermöglichen. Auch wenn bereits bei der Beschreibung der Interaktionen an Schnittstellen auf eine Technologie-Unabhängigkeit geachtet wird (siehe Herausforderung 1), müssen die verschiedenen Technologien zum Einlesen der zu prüfenden Interaktionen abgefragt werden können. Dies wird durch die vorgestellten Konzepte für eine flexible Architektur zur Datenverarbeitung (siehe Abschnitt 6.2) und für die Konfiguration der Überprüfung (siehe Abschnitt 6.3) ermöglicht. Nachdem bereits in Kapitel 5 gezeigt wurde, dass die in Herausforderung 7 geforderte Überprüfung zur Laufzeit mit Resumption

plausibel ist, wurde in Abschnitt 6.6 die Erzeugung von ausführbarem Code aus dem Modell beschrieben.

Evaluation des Ansatzes Die Evaluation der vorgestellten Konzepte in Kapitel 7 hat den Nutzen und die Verwendbarkeit des Ansatzes in verschiedenen Anwendungsbeispielen gezeigt. Zudem wurde eine Methodik zum Vergleich verschiedener Resumption-Strategien vorgestellt. Mit der Methodik wurden verschiedene Resumption-Strategien für zufällig erzeugte Beschreibungen von Soll-Verhalten verglichen und die Ergebnisse diskutiert.

Zusammengefasst sind damit die wichtigsten Beiträge dieser Arbeit zum Finden aller Abweichungen vom Soll-Verhalten zur Laufzeit:

- ein neuer modularer Ansatz zur Beschreibung von Soll-Verhalten,
- eine Methodik zur Wiederaufnahme der Überprüfung (*Resumption*),
- ein Prototyp, der zeigt, wie die Methodik realisiert werden kann und
- ein Verfahren, um die beste Resumption-Strategie für einen Anwendungsfall zu ermitteln.

8.1.2 Fragestellungen

Neben den Zielen wurden in Abschnitt 1.3 auch Fragestellungen formuliert. Die folgenden Absätze fassen die im Rahmen dieser Arbeit gefundenen Antworten zusammen.

Wie kann das Soll-Verhalten wiederverwendbar beschrieben werden?

Die Aufteilung des Modells in Schichten (siehe Abschnitt 4.2) gibt eine mögliche Antwort auf die Frage, wie Soll-Verhalten wiederverwendbar beschrieben werden kann. Zum einen können Artefakte einzelner Schichten direkt wiederverwendet werden. Zum anderen schafft die Aufteilung den Raum, die Semantik einzelner Schichten anzupassen, beispielsweise können Ereignisse mit oder ohne Kontexte ausgewertet werden.

Wie kann die Überprüfung fortgesetzt werden, nachdem eine Anomalie festgestellt wurde, um alle erkennbaren Abweichungen zu identifizieren?

Mit Hilfe von Resumption kann für jede erkennbare Abweichung ein disjunkter Bereich des Traces abgegrenzt werden, in dem sie sich befindet (siehe Theorem 4 und Theorem 5 in Abschnitt 5.1.3). Wie die Ergebnisse des zweiten Teils der Evaluation gezeigt haben, kann

die Erkennung von unerwartetem Verhalten auch mit der Erkennung von Abweichungen zusammenfallen – dann müssen keine Bereiche betrachtet werden.

Welche Abweichungen sind erkennbar?

Um diese Frage beantworten zu können, muss definiert sein, was unter einer Abweichung verstanden wird (siehe Abschnitt 2.2.4). Wie sich Abweichungen darstellen und wie ihre Auswirkungen interpretiert werden, wird in einem Fehlermodell bestimmt. Die Fehlermodelle dieser Arbeit sind geprägt durch die Annahme, dass eine Abweichung immer ein Verstoß gegen die Spezifikation ist. In Bezug auf eine Beschreibung des erwarteten Verhaltens mit einem Zustandsautomaten bedeutet das, dass es keinen passenden, gültigen Zustandsübergang zu einem auftretenden Ereignis gibt. Dann sind alle Abweichungen nicht erkennbar, die exakt erwartetes Verhalten nachahmen – das SuO verhält sich entsprechend der Spezifikation, obwohl es intern eine Abweichung gab (siehe Theorem 6 in Abschnitt 5.1.3). Um solche Abweichungen zu erkennen, müsste die Beschreibung verfeinert werden. Umgekehrt sind alle Abweichungen erkennbar, die sich von erwartetem Verhalten unterscheiden.

Wie und wann kann die Zustands-Explosion für die Fortsetzung der Überprüfung begrenzt werden?

Wie in Abschnitt 5.1.2 gezeigt wurde, kann in einfachen Automaten unerwartetes Verhalten durch Interpretation des Automaten zur Laufzeit erkannt werden, mit einem Zeitaufwand für jedes Ereignis, der linear durch die Anzahl der Zustände des ursprünglichen Automaten begrenzt ist. Jedes der dadurch definierten Segmente des Traces kann mit einer Rückwärtssuche auf den minimalen Anteil eingeschränkt werden, der genau eine erkennbare Abweichung enthält. Theorem 5 zeigt, dass dies der kürzeste Einschluss der Abweichung ist. Da ein Zustand von allen Zuständen mit einer Transition erreicht werden könnte, bedeutet dies für die Rückwärtssuche einen Zeitaufwand für jedes Ereignis, der quadratisch von der Anzahl der Zustände begrenzt wird.

Die erweiterten Modellierungskonzepte ermöglichen eine kompakte Repräsentation von (exponentiell) vielen Zuständen. Ohne weitere Einschränkungen müssen für Resumption alle möglichen Zustände berücksichtigt werden. Für Zeitschranken wurde gezeigt, wie eine kompakte Notation auch im Synchronisierungsbaum und damit für Resumption erhalten bleiben kann. Am Beispiel von hierarchischen Automaten wurde in Abschnitt 5.3 gezeigt, dass zumindest für das spezifizierte Soll-Verhalten der kompakte Automat ohne Einschränkungen beibehalten werden kann und nur die Erweiterungen durch Resumption zusätzliche Zustände erfordern. Es wurden auch für Kontexte und Instanzen in den folgenden Abschnitten mögliche Einschränkungen des allgemeinen Fehlermodells vorgestellt, die der Zustands-Explosion entgegenwirken.

8.2 Ausblick

Wie bereits durch die Evaluation gezeigt wurde, kann der vorliegende Ansatz bereits auf eine Vielzahl von Anwendungsfällen angewendet werden. Dennoch gibt es sowohl praktisch als auch theoretisch noch Forschungspotential. Durch Integration der Konzepte in die umfangreiche Werkzeugplattform Dana wird eine Verwendung vereinfacht und kann so die Forschung auf angrenzenden Themengebieten unterstützen. Beispielsweise konnte Salvi [Sal17] bereits im Rahmen seiner Master-Arbeit eine Erweiterung entwickeln, mit der die Verhaltensbeschreibung durch maschinelle Lernverfahren aus beobachtetem Ist-Verhalten erzeugt wird.

In dieser Arbeit wurden bereits erste Untersuchungen zur Optimierung von Resumption für erweiterte Konzepte der Modellierung unternommen. Es wurden erste Strategien vorgeschlagen, wie die Konzepte in bestimmten Fällen effizienter mit Resumption behandelt werden können, eine gezielte Optimierung geht allerdings über den Rahmen dieser Arbeit hinaus. Für die verschiedenen Erweiterungen wurden Rahmenbedingungen betrachtet, unter denen Resumption vereinfacht werden kann. Effiziente Resumption für erweiterte Konzepte bleibt dennoch eine Herausforderung für die Zukunft. Hier ist eine Abwägung zwischen Genauigkeit, Vollständigkeit und Geschwindigkeit notwendig, die einen Bezug zum Anwendungsszenario voraussetzt. Denn eine Einschränkung der Kandidaten für Resumption bedeutet immer das Risiko, dass ein Monitor durch ausgeschlossene aber dennoch auftretende Abweichungen in die Irre geführt wird. Weitere Arbeiten können dies vertiefen und Modellierungsmuster identifizieren, für die Resumption besonders effizient oder ineffizient angewendet werden kann.

In dieser Arbeit wurden vier Fehlermodelle betrachtet, die basierend auf einfachen Automaten definiert wurden. Hier kann untersucht werden, wie für komplexere Zustandsdiagramme, beispielsweise mit hierarchischen und parallelen Zuständen, sich Fehlermodelle, bei denen beispielsweise Abweichungen nur Zustände im gleichen Unterzustand erreichen können, sowohl auf Qualität und Performanz auswirken. In der Praxis kann untersucht werden, wie häufig solche Fälle auftauchen und wie relevant diese für die Betrachtung in bestimmten Anwendungsgebieten sind.

Der Ansatz dieser Arbeit betrachtete nur Automaten und erweiterte Zustandsdiagramme. Für die Spezifikation von Systemen werden aber auch andere Formalismen angewendet. Formalismen wie LTL definieren häufig mehrere Merkmale, die dann instanziiert und zusammen mit Methoden zum Schneiden von Traces verwendet werden, um alle Abweichungen zu erkennen. Häufig wird dazu auch ein endlicher Automat erzeugt. Durch Resumption könnten möglicherweise mehrere dieser Automaten zusammengefasst werden, um Redundanzen zu vermeiden und dennoch alle Abweichungen erkannt werden. Zudem

kann untersucht werden, ob und wie Resumption auf andere Formalismen, beispielsweise Petri-Netze, angewendet werden kann.

Der vorliegende Ansatz unterscheidet nicht zwischen der Reihenfolge der Ereignisse im Soll-Verhalten des Systems und bei deren Beobachtung. Wenn aber beispielsweise ein System ohne eine hinreichend genaue globale Uhr über verschiedene Kanäle beobachtet wird, kann es vorkommen, dass auch eine kausal begründete Sequenz von Nachrichten in abweichender Reihenfolge beobachtet wird. Dies kann auch abhängig vom Mechanismus sein, der zur Beobachtung verwendet wird, wie z. B. im Anwendungsbeispiel der Gefahrenwarner Applikation in Abschnitt 7.1.3 beschrieben wurde. Dort wurde die Verhaltensbeschreibung des Empfängers angepasst, um nicht-deterministische Beobachtungen durch parallele Regionen auszudrücken. Hierfür könnten Möglichkeiten untersucht werden, wie kausale Zusammenhänge und der Einfluss von verschiedenen Beobachtungsmechanismen auf relatives und absolutes Zeitverhalten getrennt beschrieben werden können. Prinzipiell handelt es sich hierbei auch um ein Fehlermodell – allerdings der Beobachtung. Somit könnte auch hier Resumption verwendet werden, um dennoch die Überprüfung fortzusetzen.

Literatur

[AD94] Rajeev Alur und David L. Dill. “A Theory of Timed Automata”. In: *Theoretical Computer Science* 126.2 (Apr. 1994), S. 183–235. ISSN: 03043975. DOI: 10.1016/0304-3975(94)90010-8.

[All+05] Chris Allan u. a. “Adding Trace Matching with Free Variables to AspectJ”. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’05. San Diego, CA, USA: ACM, 2005, S. 345–364. ISBN: 978-1-59593-031-6. DOI: 10.1145/1094811.1094839.

[ALR01] Algirdas Avizienis, Jean-Claude Laprie und Brian Randell. *Fundamental Concepts of Dependability*. Report CS-TR-739. Department of Computing Science, University of Newcastle upon Tyne, 2001. URL: http://www.cs.ncl.ac.uk/publications/trs/papers/739.pdf (besucht am 12. 01. 2021).

[Alt+17] Ömer Faruk Altun u. a. “Synchronizing Heuristics: Speeding up the Slowest”. In: *Testing Software and Systems*. ICTSS 2017. Hrsg. von Nina Yevtushenko, Ana Rosa Cavalli und Hüsnü Yenigün. LNCS 10533. St. Petersburg, Russia, 2017, S. 243–256. ISBN: 978-3-319-67549-7. DOI: 10.1007/978-3-319-67549-7.

[AM08] Cyril Allauzen und Mehryar Mohri. “3-Way Composition of Weighted Finite-State Transducers”. In: *Implementation and Applications of Automata*. CIAA 2008. LNCS 5148. San Francisco, CA, USA, 2008, S. 262–273. ISBN: 978-3-540-70844-5. DOI: 10.1007/978-3-540-70844-5_27.

[AM09] Cyril Allauzen und Mehryar Mohri. *Linear-Space Computation of the Edit-Distance between a String and a Finite Automaton*. Apr. 2009. arXiv: 0904.4686.

[ApaSCX] The Apache Software Foundation. *SCXML - Commons SCXML*. URL: http://commons.apache.org/proper/commons-scxml/ (besucht am 12. 01. 2021).

[Bak+18] Alexey Bakhirkin u. a. “Online Timed Pattern Matching Using Automata”. In: *Formal Modeling and Analysis of Timed Systems*. FORMATS 2018. Hrsg. von David N. Jansen und Pavithra Prabhakar. LNCS 11022. Aug. 2018, S. 215–232. ISBN: 978-3-030-00151-3. DOI: 10.1007/978-3-030-00151-3_13.

[Bet16] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. 2nd Revised edition. Packt Publishing, 31. Aug. 2016. 426 S. ISBN: 978-1-78646-496-5.

[BJR06] Therese Berg, Bengt Jonsson und Harald Raffelt. “Regular Inference for State Machines with Parameters”. In: *Fundamental Approaches to Software Engineering*. FASE 2006. Hrsg. von Luciano Baresi und Reiko Heckel. LNCS 3922. Vienna, Austria, 1. Jan. 2006, S. 107–121. ISBN: 978-3-540-33094-3. DOI: 10.1007/11693017_10.

[BLS11] Andreas Bauer, Martin Leucker und Christian Schallhart. “Runtime Verification for LTL and TLTL”. In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (Sep. 2011), 14:1–14:64. ISSN: 1049-331X. DOI: 10.1145/2000799.2000800.

[Bra04] Gilad Bracha. *Generics in the Java Programming Language*. Technical Report. Sun Microsystems, Juli 2004, S. 23.

[CD07] Michelle L. Crane und Juergen Dingel. “UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal”. In: *Software & Systems Modeling* 6.4 (1. Dez. 2007), S. 415–435. ISSN: 1619-1374. DOI: 10.1007/s10270-006-0042-8.

[CHM01] Jonathan E. Cook, Cha He und Changjun Ma. “Measuring Behavioral Correspondence to a Timed Concurrent Model”. In: *Proceedings IEEE International Conference on Software Maintenance*. ICSM 2001. Florence, Italy: IEEE, 2001, S. 332–341. DOI: 10.1109/ICSM.2001.972746.

[CK13] Mikhail Chupilko und Alexander Kamkin. “Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces”. In: *Proceedings Eighth Workshop on Model-Based Testing*. MBT 2013. Bd. 111. EPTCS. Rome, Italy, März 2013, S. 67–81. DOI: 10.4204/EPTCS.111.6.

[CTT19] Alessandro Cimatti, Chun Tian und Stefano Tonetta. “Assumption-Based Runtime Verification with Partial Observability and Resets”. In: *Runtime Verification*. RV 2019. Hrsg. von Bernd Finkbeiner und Leonardo Mariani. LNCS 11757. Porto, Portugal, 2019, S. 165–184. ISBN: 978-3-030-32079-9. DOI: 10.1007/978-3-030-32079-9_10.

[CW99] Jonathan E. Cook und Alexander L. Wolf. “Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model”. In: *ACM Trans. Softw. Eng. Methodol.* 8.2 (Apr. 1999), S. 147–176. ISSN: 1049-331X. DOI: 10.1145/304399.304401.

[DAn+18] Loris D’Antoni u. a. *Symbolic Register Automata*. 16. Nov. 2018. arXiv: 1811.06968 [cs].

[DAN12] Projekt DANA. *DANA Deliverable D2.1: Modellierungsmethodik*. Projektinterner Bericht DANA-D2.1. München: Fraunhofer ESK, 20. Dez. 2012.

[De+19] Sangita De u. a. “Towards Translation of Semantics of Automotive Interface Description Models from Franca to AUTOSAR Frameworks : An Approach Using Semantic Synergies”. In: *International Conference on Applied Electronics*. AE 2019. Pilsen, Czech Republic, Sep. 2019, S. 1–6. DOI: 10.23919/AE.2019.8867018.

[DGP15] Alessandro Danese, Tara Ghasempouri und Graziano Pravadelli. “Automatic Extraction of Assertions from Execution Traces of Behavioural Models”. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. DATE ’15. Grenoble, France: EDA Consortium, 2015, S. 67–72. ISBN: 978-3-9815370-4-8. URL: http://dl.acm.org/citation.cfm?id=2755753.2755769.

[DGR04] N. Delgado, AQ. Gates und S. Roach. “A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools”. In: *IEEE Transactions on Software Engineering* 30.12 (Dez. 2004), S. 859–872. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.91.

[DPW15] Christian Drabek, Annette Paulic und Gereon Weiss. *Reducing the Verification Effort for Interfaces of Automotive Infotainment Software*. SAE Technical Paper 2015-01-0166. Apr. 2015. DOI: 10.4271/2015-01-0166.

[Dra+13] Christian Drabek u. a. “Interface Verification Using Executable Reference Models: An Application in the Automotive Infotainment”. In: *6th International Workshop on Model Based Architecting and Construction of Embedded Systems*. ACESMB@Models 2013. CEUR-WS 1084. Miami, Florida, USA, 29. Sep. 2013. URL: http://ceur-ws.org/Vol-1084/paper7.pdf.

[DW17] Christian Drabek und Gereon Weiss. “DANA – Description and Analysis of Networked Applications”. In: *An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*. RV-CuBES 2017. Bd. 3. Kalpa Publications in Computing. EasyChair, 14. Dez. 2017, S. 71–60. DOI: 10.29007/zjn1.

[DWB17] Christian Drabek, Gereon Weiss und Bernhard Bauer. “Method for Automatic Resumption of Runtime Verification Monitors”. In: *The Third International Conference on Advances and Trends in Software Engineering*. SOFTENG 2017. Venice, Italy, 23. Apr. 2017, S. 31–36. ISBN: 978-1-61208-553-1. URL: http://www.thinkmind.org/index.php?view=article&articleid=softeng_2017_2_20_64084.

[DWB18] Christian Drabek, Gereon Weiss und Bernhard Bauer. “Resumption of Runtime Verification Monitors: Method, Approach and Application”. In: *International Journal On Advances in Software* 11 (1&2 30. Juni 2018), S. 18–33. ISSN: 1942-2628. URL: http://www.thinkmind.org/index.php?view=article&articleid=soft_v11_n12_2018_3.

[EA] SparxSystems Software GmbH. *UML, SysML, BPMN, Togaf, Updm Vereint in Enterprise Architect von Sparx Systems*. URL: https://www.sparxsystems.de/ (besucht am 01. 12. 2021).

[EATOP] Eclipse Foundation, Inc. *EATOP | The Eclipse Foundation*. URL: https://www.eclipse.org/eatop/ (besucht am 01. 12. 2021).

[EB09] Michael Eckert und François Bry. “Aktuelles Schlagwort "Complex Event Processing (CEP)"”. In: *Informatik-Spektrum* 32.2 (1. Apr. 2009), S. 163–167. ISSN: 1432-122X. DOI: 10.1007/s00287-009-0329-6.

[EclDbg] Eclipse. *Program Debug and Launch Support*. Eclipse documentation - Eclipse Neon. URL: https://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/guide/debug.htm (besucht am 12. 01. 2021).

[ECLIPSE] Eclipse Foundation Inc. *Enabling Open Innovation & Collaboration | The Eclipse Foundation*. URL: https://www.eclipse.org/ (besucht am 12. 01. 2021).

[Epp90] David Eppstein. “Reset Sequences for Monotonic Automata”. In: *SIAM Journal on Computing* 19.3 (Juni 1990), S. 500–510. ISSN: 1095-7111. DOI: 10.1137/0219033.

[Fal+18] Yliès Falcone u. a. “A Taxonomy for Classifying Runtime Verification Tools”. In: *18th International Conference on Runtime Verification*. RV 2018. Limassol, Cyprus, Nov. 2018, S. 1–18. URL: https://hal.inria.fr/hal-01882410.

[Feeder] *Vibrationswendelförderer*. In: *Wikipedia*. URL: https://de.wikipedia.org/wiki/Vibrationswendelf%C3%B6rderer (besucht am 12. 01. 2021).

[FHR13] Yliès Falcone, Klaus Havelund und Giles Reger. "A Tutorial on Runtime Verification." In: *Engineering Dependable Software Systems*. NATO Science for Peace and Security Series - D: Information and Communication Security 34 (2013), S. 141–175. DOI: 10.3233/978-1-61499-207-3-141.

[Fow11] Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley Signature Series. Addison-Wesley, 2011. 921 S. ISBN: 978-0-321-71294-3. Google Books: ri1muolw_YwC.

[Fra17] Fraunhofer ESK. *AUTOTRACE AP3.2: Modellierungskonzept für ein systemweites Soll-Verhalten Modell*. Projekt-interner Bericht AUTOTRACE-AP3.2. München: Fraunhofer ESK, 11. Juli 2017.

[Franca] itemis AG. *Franca User Guide - Release 0.12.0.1*. 2018. URL: https://github.com/franca/franca (besucht am 12. 01. 2021).

[GENIVI] *GENIVI Alliance*. URL: https://www.genivi.org/ (besucht am 12. 01. 2021).

[Hal16] Sylvain Hallé. "When RV Meets CEP". In: *16th International Conference on Runtime Verification*. RV 2016. LNCS 10012. Madrid, Spain, 27. Sep. 2016, S. 68–91. DOI: 10.1007/978-3-319-46982-9_6.

[Har87] David Harel. "Statecharts: A Visual Formalism for Complex Systems". In: *Science of Computer Programming* 8.3 (1. Juni 1987), S. 231–274. ISSN: 0167-6423. DOI: 10.1016/0167-6423(87)90035-9.

[HHS16] Thomas Herpel, Thomas Hoiss und Jan Schroeder. "Enhanced Simulation-Based Verification and Validation of Automotive Electronic Control Units". In: *Electronics, Communications and Networks V*. CECNet 2015. Hrsg. von Amir Hussain. Bd. 382. LNEE. 2016, S. 203–213. ISBN: 978-981-10-0740-8. DOI: 10.1007/978-981-10-0740-8_24.

[Hil+18] Steffen Hillemacher u. a. "Model-Based Development of Self-Adaptive Autonomous Vehicles Using the SMARDT Methodology:" In: *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*. MODELSWARD 2018. Funchal, Madeira, Portugal: SCITEPRESS - Science and Technology Publications, 2018, S. 163–178. ISBN: 978-989-758-283-7. DOI: 10.5220/0006603701630178.

[HK04] David Harel und Hillel Kugler. "The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)". In: *Integration of Software Specification Techniques for Applications in Engineering*. Bd. 3147. LNCS. 2004, S. 325–354. ISBN: 978-3-540-27863-4. DOI: 10.1007/978-3-540-27863-4_19.

[HKW17] Alexander Hagemann, Gerrit Krepinsky und Christian Wolf. “Interface Construction, Deployment and Operation – a Mystery Solved”. In: *International Journal on Advances in Software* 10 (1&2 2017), S. 61–78. ISSN: 1942-2628. URL: https://www.thinkmind.org/index.php?view=article&articleid=soft_v10_n12_2017_5 (besucht am 12. 01. 2021).

[HMF14] D. Heffernan, C. Macnamee und P. Fogarty. “Runtime Verification Monitoring for Automotive Embedded Systems Using the ISO 26262 Functional Safety Standard as a Guide for the Definition of the Monitored Properties”. In: *IET Software* 8.5 (Okt. 2014), S. 193–203. ISSN: 1751-8806. DOI: 10.1049/iet-sen.2013.0236.

[Hop71] John Hopcroft. “AN n Log n ALGORITHM FOR MINIMIZING STATES IN A FINITE AUTOMATON”. In: *Proceedings of an International Symposium on the Theory of Machines and Computations*. Theory of Machines and Computations. Hrsg. von Zvi Kohavi und Azaria Paz. Haifa, Isreal: Academic Press, Aug. 1971, S. 189–196. ISBN: 978-0-12-417750-5. DOI: 10.1016/B978-0-12-417750-5.50022-1.

[hs10] hs. *Eclipse Modeling Framework - Interview with Ed Merks*. JAXenter. 16. Apr. 2010. URL: https://jaxenter.com/eclipse-modeling-framework-interview-with-ed-merks-100007.html (besucht am 12. 01. 2021).

[JR91] Tao Jiang und B. Ravikumar. “Minimal NFA Problems Are Hard”. In: *Automata, Languages and Programming*. ICALP 1991. Hrsg. von Javier Leach Albert, Burkhard Monien und Mario Rodríguez Artalejo. LNCS 510. Madrid, Spain, 1991, S. 629–640. ISBN: 978-3-540-47516-3. DOI: 10.1007/3-540-54233-7_169.

[JSON] *JavaScript Object Notation*. In: *Wikipedia*. URL: https://de.wikipedia.org/wiki/JavaScript_Object_Notation (besucht am 12. 01. 2021).

[Kar+16] Sertaç Karahoda u. a. “Parallelizing Heuristics for Generating Synchronizing Sequences”. In: *Testing Software and Systems*. ICTSS 2016. Hrsg. von Franz Wotawa, Mihai Nica und Natalia Kushik. LNCS 9976. Graz, Austria, 2016, S. 106–122. ISBN: 978-3-319-47443-4. DOI: 10.1007/978-3-319-47443-4_7.

[KBK16] Andreas Kurtz, Bernhard Bauer und Marcel Koeberl. “Software Based Test Automation Approach Using Integrated Signal Simulation”. In: *The Second International Conference on Advances and Trends in Software Engineering*. SOFTENG 2016. Lisbon, Portugal, 21. Feb. 2016, S. 117–122. ISBN: 978-1-61208-458-9. URL: http://www.thinkmind.org/index.php?view=article&articleid=softeng_2016_5_20_65040.

[KKY18] Serta Karahoda, Kamer Kaya und Hsn Yenign. “Synchronizing Heuristics”. In: *Expert Syst. Appl.* 94.C (März 2018), S. 265–275. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2017.10.054.

[Klo+10] Kay Klobedanz u. a. “Timing Modeling and Analysis for AUTOSAR-Based Software Development: A Case Study”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’10 (Dresden, Germany). Dresden, Germany: European Design and Automation Association, 2010, S. 642–645. ISBN: 978-3-9810801-6-2. URL: http://dl.acm.org/citation.cfm?id=1870926.1871078.

[Knü+10] Christian Knüchel u. a. “Artop – an Ecosystem Approach for Collaborative AUTOSAR Tool Development”. In: *International Congress on Embedded Real Time Software and Systems*. ERTS2010. Toulouse, France, 2010. URL: https://hal.archives-ouvertes.fr/hal-02267845/.

[Lan+09] Agnes Lanusse u. a. “Papyrus UML: An Open Source Toolset for MDA”. In: *Fifth European Conference on Model-Driven Architecture Foundations and Applications: Proceedings of the Tools and Consultancy Track*. ECMDA-FA ’09. Enschede, The Netherlands, Juni 2009, S. 1–4.

[Lau+19] Henrich Lauko u. a. “Extending DIVINE with Symbolic Verification Using SMT”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2019. Hrsg. von Dirk Beyer u. a. LNCS 11429. Prague, Czech Republic, 2019, S. 204–208. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_14.

[LM01] Gerald Lüttgen und Michael Mendler. “Statecharts: From Visual Syntax to Model–Theoretic Semantics”. In: *Workshop on Integrating Diagrammatic and Formal Specification Techniques (IDFST 2001)*. Informatik 2001: Wirtschaft Und Wissenschaft in Der Network Economy - Visionen Und Wirklichkeit 1 (2001), S. 615–621.

[LS09] Martin Leucker und Christian Schallhart. “A Brief Account of Runtime Verification”. In: *The Journal of Logic and Algebraic Programming* 78.5 (Mai 2009), S. 293–303. ISSN: 1567-8326. DOI: 10.1016/j.jlap.2008.08.004.

[LS11] David Luckham und W. Roy Schulte. *EPTS Event Processing Glossary v2.0*. Event Processing Technical Society, Juli 2011. URL: http://www.complexevents.com/2011/08/23/event-processing-glossary-version-2/ (besucht am 12. 01. 2021).

[LY96] D. Lee und M. Yannakakis. “Principles and Methods of Testing Finite State Machines-a Survey”. In: *Proceedings of the IEEE* 84.8 (Aug. 1996), S. 1090–1123. ISSN: 0018-9219. DOI: 10.1109/5.533956.

[Mal04] Andreas Malcher. “Minimizing Finite Automata Is Computationally Hard”. In: *Theoretical Computer Science* 327.3 (Nov. 2004), S. 375–390. ISSN: 03043975. DOI: 10.1016/j.tcs.2004.03.070.

[MATLAB] *MATLAB - MathWorks - MATLAB & Simulink*. URL: https://de.mathworks.com/products/matlab.html (besucht am 12. 01. 2021).

[Mea55] George H. Mealy. “A Method for Synthesizing Sequential Circuits”. In: *The Bell System Technical Journal* 34.5 (Sep. 1955), S. 1045–1079. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1955.tb03788.x.

[Mer+11] Patrick O’Neil Meredith u. a. “An Overview of the MOP Runtime Verification Framework”. In: *International Journal on Software Tools for Technology Transfer* 14.3 (23. Apr. 2011), S. 249–289. ISSN: 1433-2787. DOI: 10.1007/s10009-011-0198-6.

[MLS97] Erich Mikk, Yassine Lakhnechi und Michael Siegel. “Hierarchical Automata as Model for Statecharts (Extended Abstract)”. In: *Annual Asian Computing Science Conference*. ASIAN’97. LNCS 1345. Kathmandu, Nepal: Springer, 1997, S. 181–196. ISBN: 978-3-540-69658-2. DOI: 10.1007/3-540-63875-X_52.

[MNE15] Alexander Maier, Oliver Niggemann und Jens Eickmeyer. “On the Learning of Timing Behavior for Anomaly Detection in Cyber-Physical Production Systems.” In: *26th International Workshop on Principles of Diagnosis*. DX@ Safeprocess. CEUR-WS 1507. Paris, France, 2015, S. 217–224. URL: http://ceur-ws.org/Vol-1507/dx15paper28.pdf.

[Mub+17] Saad Mubeen u. a. “Supporting Timing Analysis of Vehicular Embedded Systems through the Refinement of Timing Constraints”. In: *Software & Systems Modeling* 18.1 (2017), S. 39–69. ISSN: 1619-1374. DOI: 10.1007/s10270-017-0579-8.

[Nas16] Ahmed Nassar. “Specification and Runtime Verification of Distributed Multiprocessor Systems: Languages, Tools and Architectures”. Dissertation. Irvine: University of California, 2016. 200 S. URL: https://escholarship.org/uc/item/2f63n8px (besucht am 12. 01. 2021).

[Nik+19] Mihajlo Nikolić u. a. “Utilization of Pattern Generators in Adaptive AUTOSAR Platform”. In: *27th Telecommunications Forum*. TELFOR 2019. Belgrade, Serbia, Nov. 2019, S. 1–4. DOI: 10.1109/TELFOR48224.2019.8971306.

[OMG-FUML] OMG. *Semantics of a Foundational Subset for Executable UML Models, v1.4*. Specification FUML/1.4/PDF. Object Management Group, Dez. 2018, S. 396. URL: https://www.omg.org/spec/FUML/1.4/.

[OMG-MOF] OMG. *Meta Object Facility, v2.5.1*. Specification MOF/2.5.1/PDF. Object Management Group, Nov. 2016, S. 80. URL: https://www.omg.org/spec/MOF/2.5.1/.

[OMG-PSSM] OMG. *Precise Semantics of UML State Machines (PSSM), v1.0*. Specification PSSM/1.0. Object Management Group, Mai 2019, S. 272. URL: https://www.omg.org/spec/PSSM/1.0.

[OMG-UML] OMG. *Unified Modeling Language, v2.5.1*. Specification UML/2.5.1/PDF. Object Management Group, 5. Dez. 2017, S. 796. URL: https://www.omg.org/spec/UML/2.5.1.

[Per+12] M. Peraldi-Frati u. a. "A Timing Model for Specifying Multi Clock Automotive Systems: The Timing Augmented Description Language V2". In: *IEEE 17th International Conference on Engineering of Complex Computer Systems*. ICECCS 2012. Paris, France, Juli 2012, S. 230–239. URL: https://hal.inria.fr/hal-00687562.

[PL05] Alexander Pretschner und Martin Leucker. "Model-Based Testing – A Glossary". In: *Model-Based Testing of Reactive Systems*. LNCS 3472. Springer, Berlin, Heidelberg, 2005, S. 607–609. ISBN: 978-3-540-32037-1. DOI: 10.1007/11498490_27.

[Pow11] David M. W. Powers. "Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation". In: *Journal of Machine Learning Technologies* 2.1 (2011), S. 37–63. arXiv: 2010.16061. URL: http://hdl.handle.net/2328/27165.

[Pra+13] Thomas Pramsohler u. a. "Control Flow Analysis of Automotive Software Components Using Model-Based Specifications of Dynamic Behavior". In: *SAE International Journal of Passenger Cars - Electronic and Electrical Systems* 6.2 (8. Apr. 2013), S. 425–436. DOI: 10.4271/2013-01-0435.

[Rab+17] Rick Rabiser u. a. "A Comparison Framework for Runtime Monitoring Approaches". In: *Journal of Systems and Software* 125 (März 2017), S. 309–321. ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.12.034.

[RBR15] Giles Reger, Howard Barringer und David Rydeheard. "Automata-Based Pattern Mining from Imperfect Traces". In: *SIGSOFT Softw. Eng. Notes* 40.1 (Feb. 2015), S. 1–8. ISSN: 0163-5948. DOI: 10.1145/2693208.2693220.

[Reg15] Giles Reger. "Suggesting Edits to Explain Failing Traces". In: *Runtime Verification*. RV 2015. LNCS 9333. Vienna, Austria, 2015, S. 287–293. ISBN: 978-3-319-23820-3. DOI: 10.1007/978-3-319-23820-3_20.

[RHAPSODY] *IBM Engineering Systems Design Rhapsody - Overview*. URL: https://www.ibm.com/products/systems-design-rhapsody (besucht am 12. 01. 2021).

[RS15] Adam Roman und Marek Szykuła. “Forward and Backward Synchronizing Algorithms”. In: *Expert Systems with Applications* 42.24 (30. Dez. 2015), S. 9512–9527. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2015.07.071.

[RS93] Ronald L. Rivest und Robert E. Schapire. “Inference of Finite Automata Using Homing Sequences”. In: *Machine Learning: From Theory to Applications*. Springer, Berlin, Heidelberg, 1993, S. 51–73. ISBN: 978-3-540-47568-2. DOI: 10.1007/3-540-56483-7_22.

[Sal17] Aniket Salvi. “Design and Prototyping of Self-Learning Methods for Semi-Automated Model Creation as a Reference Behavior in Distributed Embedded System”. Master Thesis. Magdeburg: Magdeburg, Univ., 2017. 86 S.

[Sam16] Miro Samek. *Introduction to Hierarchical State Machines (HSMs)*. Barr Group. 4. Mai 2016. URL: https://barrgroup.com/Embedded-Systems/How-To/Introduction-Hierarchical-State-Machines (besucht am 12. 01. 2021).

[San05] Sven Sandberg. “Homing and Synchronizing Sequences”. In: *Model-Based Testing of Reactive Systems*. Hrsg. von Manfred Broy u. a. LNCS 3472. Springer Berlin Heidelberg, 2005, S. 5–33. ISBN: 978-3-540-26278-7. DOI: 10.1007/11498490_2.

[SBS20] Stefan Schlichthaerle, Klaus Becker und Sebastian Sperber. “A Domain-Specific Language Based Architecture Modeling Approach for Safety Critical Automotive Software Systems”. In: *Combined Proceedings of the Workshops at Software Engineering 2020*. ASE 2020: 17th Workshop on Automotive Software Engineering. CEUR-WS 2581. Innsbruck, Austria: CEUR Workshop Proceedings, März 2020, S. 6. URL: http://ceur-ws.org/Vol-2581/.

[Sch+02] B. Schätz u. a. “Model-Based Development of Embedded Systems”. In: *Advances in Object-Oriented Information Systems*. OOIS 2002. Hrsg. von Jean-Michel Bruel und Zohra Bellahsene. Bearb. von Gerhard Goos, Juris Hartmanis und Jan van Leeuwen. Bd. 2. LNCS 2426. Montpellier, France, 2002, S. 298–311. ISBN: 978-3-540-46105-0. DOI: 10.1007/3-540-46105-1_34.

[SCXML] *SCXML*. In: *Wikipedia*. URL: https://en.wikipedia.org/wiki/SCXML (besucht am 12. 01. 2021).

[SCXML15] Jim Barnett u. a. *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. W3C Recommendation. W3C, Sep. 2015. URL: https://www.w3.org/TR/2015/REC-scxml-20150901/.

[SÇZ05] Michael Stonebraker, Uğur Çetintemel und Stan Zdonik. “The 8 Requirements of Real-Time Stream Processing”. In: *ACM SIGMOD Rec.* 34.4 (Dez. 2005), S. 42–47. ISSN: 0163-5808. DOI: 10.1145/1107499.1107504.

[Sta+07] Thomas Stahl u. a. *Modellgetriebene Softwareentwicklung : Techniken, Engineering, Management*. 2. Auflage. Heidelberg, GERMANY: dpunkt.verlag, 2007. 458 S. ISBN: 978-3-89864-881-3.

[Ste+08] David Steinberg u. a. *EMF: Eclipse Modeling Framework*. 2nd Revised edition. Upper Saddle River, NJ: Addison Wesley, 16. Dez. 2008. 704 S. ISBN: 978-0-321-33188-5.

[Sto+12] Scott D. Stoller u. a. “Runtime Verification with State Estimation”. In: *Second International Conference on Runtime Verification*. RV 2011. Hrsg. von Sarfraz Khurshid und Koushik Sen. LNCS 7186. San Francisco, CA, USA, 1. Jan. 2012, S. 193–207. ISBN: 978-3-642-29860-8. DOI: 10.1007/978-3-642-29860-8_15.

[SWH11] M. Staats, M. W. Whalen und M. P. E. Heimdahl. “Programs, Tests, and Oracles: The Foundations of Testing Revisited”. In: *33rd International Conference on Software Engineering*. ICSE ’11. Waikiki, Honolulu, HI, USA, Mai 2011, S. 391–400. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985847.

[TIM09] TIMMO. *TIMMO D6: Timing Model*. Projektbericht Deliverable D6. 5. Okt. 2009. URL: http://adt.cs.upb.de/timmo-2-use/timmo/pdf/D6_TIMMO_TADL_Version_2_v12.pdf (besucht am 12. 01. 2021).

[TIM12] TIMMO-2-USE Consortium. *TIMMO-2-USE D11: Language Syntax, Semantics, Metamodel V2*. Projektbericht Deliverable D11. 30. Aug. 2012. URL: http://adt.cs.upb.de/timmo-2-use/deliverables/TIMMO-2-USE_D11.pdf (besucht am 12. 01. 2021).

[TTCN3] TTCN-3.org Editorial Team. *Standardized Test Suites*. ETSI’S OFFICIAL TTCN-3 HOMEPAGE. URL: http://www.ttcn-3.org/index.php/downloads/publicts (besucht am 12. 01. 2021).

[URI] *Uniform Resource Identifier*. In: *Wikipedia*. URL: https://de.wikipedia.org/wiki/Uniform_Resource_Identifier (besucht am 12. 01. 2021).

[vdAAvD12] Wil van der Aalst, Arya Adriansyah und Boudewijn van Dongen. “Replaying History on Process Models for Conformance Checking and Performance Analysis”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2.2 (1. März 2012), S. 182–192. ISSN: 1942-4795. DOI: 10.1002/widm.1045.

[VModellXT] Verein zur Weiterentwicklung des V-Modell XT e.V. (Weit e.V.) *V-Modell XT - Das deutsche Referenzmodell für Systementwicklungsprojekte*. URL: www.v-modell-xt.de (besucht am 12. 01. 2021).

[VMP14] V. Viyović, M. Maksimović und B. Perisić. "Sirius: A Rapid Development of DSM Graphical Editor". In: *IEEE 18th International Conference on Intelligent Engineering Systems*. INES 2014. Tihany, Hungary, Juli 2014, S. 233–238. ISBN: 978-1-4799-4615-0. DOI: 10.1109/INES.2014.6909375.

[Voe13] Markus Voelter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. ISBN: 978-1-4812-1858-0. URL: http://www.dslbook.org.

[vZij04] Lynette van Zijl. "Generalized Acceptance, Succinctness and Supernondeterministic Finite Automata". In: *Theoretical Computer Science* 313.1 (Feb. 2004), S. 159–172. ISSN: 03043975. DOI: 10.1016/j.tcs.2003.10.013.

[WA19] Masaki Waga und Étienne André. "Online Parametric Timed Pattern Matching with Automata-Based Skipping". In: *NASA Formal Methods*. NFM 2019. Hrsg. von Julia M. Badger und Kristin Yvonne Rozier. LNCS 11460. Houston, TX, USA, Mai 2019, S. 371–389. ISBN: 978-3-030-20652-9. DOI: 10.1007/978-3-030-20652-9_26. arXiv: 1903.07328.

[XTEXT] *Xtext - Language Engineering Made Easy!* URL: https://www.eclipse.org/Xtext/ (besucht am 12. 01. 2021).

[Yan00] Mihalis Yannakakis. "Hierarchical State Machines". In: *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*. TCS 2000. Hrsg. von Jan van Leeuwen u. a. LNCS 1872. Sendai, Japan, 2000, S. 315–330. ISBN: 978-3-540-44929-4. DOI: 10.1007/3-540-44929-9_24.

[Yin+16] Sorrachai Yingchareonthawornchai u. a. "Precision, Recall, and Sensitivity of Monitoring Partially Synchronous Distributed Systems". In: *International Conference on Runtime Verification*. RV 2016. LNCS 10012. Madrid, Spain, Sep. 2016, S. 420–435. ISBN: 978-3-319-46982-9. DOI: 10.1007/978-3-319-46982-9_26.