TECHNISCHE
UNIVERSITÄT
DARMSTADT

Generierung von effizienten Security-/Safety-Monitoren aus modellbasierten Beschreibungen

VOM FACHBEREICH ELEKTRO- UND INFORMATIONSTECHNIK
DER TECHNISCHEN UNIVERSITÄT DARMSTADT
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
EINES DOKTOR-INGENIEURS (DR.-ING.)
GENEHMIGTE DISSERTATION

VON

DIPL.-ING. LARS PATZINA

GEBOREN AM

01. SEPTEMBER 1981 IN BAD SODEN AM TAUNUS

REFERENT: PROF. DR. RER. NAT. ANDY SCHÜRR
KOREFERENT: PROF. DR. JAN JÜRJENS
TAG DER EINREICHUNG: 31. JANUAR 2014
TAG DER MÜNDLICHEN PRÜFUNG: 10. JULI 2014

D17
DARMSTADT 2014

Lars Patzina: *Generierung von effizienten Security-/Safety-Monitoren aus modellbasierten Beschreibungen*, © Januar 2014

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe. Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 31. Januar 2014

Lars Patzina

Das einzig sichere System müsste ausgeschaltet,
in einem versiegelten und von Stahlbeton ummantelten Raum
und von bewaffneten Schutztruppen umstellt sein.

— Gene Spafford

DANKSAGUNG

Insbesondere danke ich Herrn Prof. Dr. rer. nat. Andy Schürr für die Bereitstellung des interessanten und herausfordernden Themas, die engagierte wissenschaftliche Betreuung und das intensive Interesse am Erfolg meiner Dissertation. Des Weiteren danke ich Herrn Prof. Dr. Jan Jürjens für die Übernahme des Zweitgutachtens und das Interesse an meiner Arbeit. Mein besonderer Dank gilt weiterhin Sven Patzina für die sehr gute Zusammenarbeit bei der Zusammenführung unserer Dissertationsthemen und die anregenden Diskussionen. Außerdem möchte ich mich bei allen Mitarbeitern und Mitarbeiterinnen des Fachgebiets Echtzeitsysteme für die stetige Hilfsbereitschaft, Diskussionsbereitschaft und die angenehme Arbeitsatmosphäre bedanken. Abschließend danke ich dem *Center for Advanced Security Research Darmstadt* (CASED) für die Unterstützung meiner Promotion durch ein dreijähriges Promotionsstipendium.

ABSTRACT

Small processing units with sensors for the collection of environmental data increasingly complement classical computers. These units communicate with each other and with external units to propagate information and coordinate with other units. As a result, even safety-critical embedded systems open up to the outside world. These systems are now vulnerable to direct attacks and indirect attacks that are performed involving additional non-safety-critical units.

Software that is used on resource constrained embedded systems is typically reduced to the most necessary and does not include complex security mechanisms. Measures such as testing of software cannot ensure the absence of errors. In real systems, it can be assumed that errors exist that can be exploited by attackers.

Runtime monitoring of such systems has been proven to be effective to detect previously unknown attacks and failures. Many different languages have emerged that support the description of such specifications (signatures) of allowed and prohibited behavior. These languages support diverse modelling concepts. For the generation of monitors from these specifications in software and hardware, various code generators have to be developed. Additionally, some of the commonly used specification languages have no formal definition of their syntax and semantics.

This PhD thesis presents in conjunction with [Pat14], the *Model-based Security-/Safety Monitor (MBSecMon)* development process. The process is parallel to the actual software-development process of the system that should be monitored and includes the specification, the generation and the integration of runtime monitors.

The goal of this thesis is the design of a formally defined intermediate language for the representation of highly interleaved concurrent communications. Based on existing approaches the requirements for such an intermediate language are determined. On this basis, the intermediate language *Monitor Petri nets* (MPN) is designed and formally defined. This intermediate language supports the representation of signatures that are modeled in various specification languages and the generation of efficient runtime monitors for different target platforms. The MPNs are a formalism that is based on Petri nets and is extended with concepts for the domain of runtime monitoring. The MPN language completely fulfils the determined requirements except of a hierarchical concept for events, which is out-of-scope of this thesis.

A prototypical tool has been developed that supports the generation of runtime monitors. It supports the MBSecMon specification language [Pat14] as complex input language and uses the MPN language as intermediate representation for the generation of monitors for various target platforms and languages. The generated monitors are evaluated for their runtime behavior and their memory consumption. It becomes apparent that the MPN language despite its expressiveness is suited for the straightforward generation of efficient runtime monitors for various platforms and target languages.

ZUSAMMENFASSUNG

Computer werden heute zunehmend durch kleine Recheneinheiten mit Sensoren zur Erfassung der Außenwelt ergänzt. Diese Recheneinheiten kommunizieren untereinander und mit externen Einheiten, um Informationen weiterzugeben und sich untereinander abzustimmen. Hierdurch findet auch eine Öffnung von sicherheitskritischen eingebetteten Systemen nach außen statt. Die Systeme können nun entweder direkt oder indirekt über zusätzliche Einheiten angegriffen werden.

Des Weiteren ist die auf eingebetteten Systemen eingesetzte Software durch beschränkte Ressourcen auf das Nötigste reduziert und bietet keine komplexen Sicherheitsmechanismen. Maßnahmen wie Testen von Software kann deren Fehlerfreiheit nicht sicherstellen. In realen Systemen ist zudem davon auszugehen, dass nicht bekannte Fehler existieren, die u. a. auch von Angreifern ausgenutzt werden können.

Die Laufzeitüberwachung solcher Systeme hat sich als geeignet erwiesen, um auch unbekannte Angriffe und Fehler zu erkennen. Zur Spezifikation solcher Laufzeitmonitore über Beschreibungen (Signaturen) von erlaubtem und verbotenem Verhalten haben sich viele verschiedene Spezifikationssprachen herausgebildet. Diese basieren auf verschiedensten Modellierungskonzepten. Zur Generierung von Monitoren aus diesen Spezifikationen in Software und Hardware müssen für die unterschiedlichen Sprachen verschiedenste Codegeneratoren erstellt werden. Des Weiteren besitzen einige der gewöhnlich verwendeten einfach zu verstehenden Spezifikationssprachen keine formalisierte Syntax und Semantik.

In dieser Arbeit wird zusammen mit [Pat14] der *Model-based Security/Safety Monitor (MBSecMon)*-Entwicklungsprozess vorgestellt. Dieser umfasst parallel zu dem eigentlichen Softwareentwicklungsprozess des zu überwachenden Systems die Spezifikation, die Generierung und die Einbindung von Laufzeitmonitoren.

Ziel dieser Arbeit ist die Definition einer formal definierten Zwischensprache zur Repräsentation stark verschränkter nebenläufiger Kommunikationen. Zu ihrer Entwicklung werden Anforderungen basierend auf existierenden Arbeiten aufgestellt. Auf Grundlage dieser Anforderungen wird die Zwischensprache *Monitor-Petrinetze* (MPN) entworfen und formal definiert. Diese Zwischensprache unterstützt die Repräsentation von Signaturen, die in verschiedensten Spezifikationssprachen modelliert sind, und die Generierung von effizienten Laufzeitmonitoren für unterschiedliche Zielplattformen. Die MPNs sind ein auf Petrinetzen basierender Formalismus, der um Konzepte der Laufzeitüberwachung erweitert wurde. Es wird gezeigt, dass die MPN-Sprache alle ermittelten Anforderungen an eine solche Zwischensprache, bis auf ein Hierarchisierungskonzept für Ereignisse, das in dieser Arbeit nicht behandelt wird, erfüllt.

Die MPN-Sprache wird in einem prototypischen Werkzeug zur Monitorgenerierung eingesetzt. Dieses unterstützt die MBSecMon-Spezifikationssprache [Pat14] als Eingabesprache und verwendet die MPN-Sprache als Zwischenrepräsentation zur Monitorgenerierung für verschiedenste Plattformen und Zielsprachen. Die

generierten Monitore werden auf ihr Laufzeitverhalten und ihren Speicherverbrauch evaluiert. Es hat sich gezeigt, dass sich die MPN-Sprache trotz ihrer hohen Ausdrucksstärke zur einfachen Generierung effizienter Laufzeitmonitore für verschiedenste Plattformen und Zielsprachen eignet.

VERÖFFENTLICHUNGEN

Einige in dieser Arbeit vorgestellten Ansätze wurden bereits in folgenden Veröffentlichungen publiziert:

[PPPS10] Patzina, Lars; Patzina, Sven; Piper, Thorsten; Schürr, Andy: Monitor Petri Nets for Security Monitoring. In: *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems (S&D4RCES 2010)*, ACM, 2010 (S&D4RCES). – ISBN 978–1–4503–0368–2, S. 3:1–3:6 (Zitiert auf Seite 81.)

[BPP+11] Biedermann, Alexander; Piper, Thorsten; Patzina, Lars; Patzina, Sven; Huss, Sorin A.; Schürr, Andy; Suri, Neeraj: Enhancing FPGA Robustness via Generic Monitoring Cores. In: *1st International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2011)*, 2011, S. 379–386 (Zitiert auf Seite 5.)

[PPS11] Patzina, Sven; Patzina, Lars; Schürr, Andy: Extending LSCs for Behavioral Signature Modeling. In: Camenisch, Jan; Fischer-Hübner, Simone; Murayama, Yuko; Portmann, Armand; Rieder, Carlos (Hrsg.): *Future Challenges in Security and Privacy for Academia and Industry (IFIP SEC 2011)* Bd. 354 Springer, Springer, June 2011 (IFIP Advances in Information and Communication Technology). – ISBN 978–3–642–21423–3, S. 293–304 (Zitiert auf Seite 48.)

[PP12] Patzina, Sven; Patzina, Lars: A Case Study Based Comparison of ATL and SDM. In: Schürr, Andy; Varró, Dániel; Varró, Gergely (Hrsg.): *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2012)* Bd. 7233, Springer, 2012 (LNCS). – ISBN 978–3–642–34175–5, S. 210–221

[PPPM13] Patzina, Lars; Patzina, Sven; Piper, Thorsten; Manns, Paul: Model-Based Generation of Run-Time Monitors for AUTOSAR. Version: 2013. http://dx.doi.org/10.1007/978-3-642-39013-5_6. In: Gorp, Pieter; Ritter, Tom; Rose, Louis M. (Hrsg.): *Modelling Foundations and Applications (ECMFA 2013)* Bd. 7949. DOI. – 10.1007/978–3–642–39013– 5_6. – ISBN 978–3–642–39012–8, S. 70–85. – Best Paper Arward (Zitiert auf Seite 74 und 215.)

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

AUFLISTUNGSVERZEICHNIS

Teil I

EINLEITUNG

1

EINLEITUNG

Seit der Einführung von Computern findet bis heute eine zunehmende Technisierung der Gesellschaft statt. Beginnend mit dem PC (Personal Computer), der 1981 durch IBM eingeführt wurde, begann der Siegeszug der Computer in den Privathaushalten. Dieser wird bis heute durch die Entwicklung neuer immer kompakterer Rechner wie Notebooks, Tablets und Smartphones fortgesetzt. Heute sind Computer in den verschiedensten Varianten überall im täglichen Leben vorzufinden.

Computer werden im Alltag zunehmend von diversen kleinen Recheneinheiten ergänzt. Diese sollen die Lücke zwischen realer und virtueller Welt durch die automatische Erfassung der Umwelt schließen. Die Erfassung findet dabei nicht mehr durch den Menschen selbst, sondern durch diese kleinen Einheiten mit Sensoren statt. Diese kommunizieren untereinander und sind größtenteils nur noch eingeschränkt auf Interaktion mit dem Nutzer ausgelegt. So kommunizieren u. a. Kühlschränke, Videoüberwachungsanlagen, Drucker, medizinische Geräte und sogar Windturbinen über das Internet und lokal miteinander.

Diese durch die Systeme erhobenen Informationen werden genutzt, um die Menschen bei ihrer Arbeit oder im Alltag zu unterstützen bzw. die Sicherheit zu erhöhen. So aggregiert heutzutage z. B. ein Fahrzeug viele Informationen über den aktuellen Zustand des Fahrzeugs und der Umwelt, um in Gefahrensituationen die Insassen aktiv zu schützen oder den Fahrer mit zusätzlichen Informationen zu versorgen.

Die die Informationen erfassenden und verarbeitenden Recheneinheiten werden in hoher Stückzahl in Fahrzeugen verbaut und stellen einen erheblichen Kostenfaktor bei der Produktion eines solchen Fahrzeugs dar. Die Recheneinheiten (eingebettete Systeme) besitzen deshalb zur Reduktion der Herstellungskosten eine sehr eingeschränkte Rechenleistung und begrenzte Speicherressourcen. Aus diesem Grund werden für sie möglichst effiziente, auf das Nötigste beschränkte Programme entwickelt. In vielen Domänen waren diese in einem lokalen Netzwerk zusammenarbeitenden Einheiten bisher von der Außenwelt getrennt und bezogen ihre Informationen über Sensoren. Hierdurch waren Sicherheitslücken in diesen Systemen ohne physischen Zugang nicht oder nur schwer auszunutzen. Diese sicherheitskritischen Systeme sowie die mit ihnen agierenden klassischen Computersysteme bieten jedoch ein großes Potenzial für Angriffe.

Getrieben von technischen Innovationen arbeiten eingebettete Systeme heute oft nicht mehr isoliert, sondern formen das sogenannte „Internet der Dinge“ (*engl. Internet of Things*). In diesem bilden kleine über die Zeit wechselnde Recheneinheiten, die über Kommunikationsschnittstellen verfügen, eine internetähnliche Kom-

munikationsstruktur und tauschen erhobene und vorverarbeitete Daten untereinander aus. Somit können eingebettete Systeme nicht mehr als isolierte Systeme gesehen werden, die keine Angriffsfläche von außen bieten, obwohl sie ursprünglich als solche entwickelt wurden. Für sich selbst sichere Komponenten können durch unterschiedlich wirksame Sicherheitskonzepte der Kommunikationspartner bedroht werden. Dies trifft sogar auf sicherheitskritische Anwendungsgebiete wie den Automobilbereich zu.

Nach der Einführung elektronischer Mautsysteme mit drahtloser Kommunikation zwischen Mautbrücke und On-Bord-Units sind verschiedenste Car-to-Car- und Car-to-Infrastucture-Kommunikationsanwendungen in Entwicklung, die unter anderem vom *Car2Car Communication Consortium* (C2CCC)[1] vorangetrieben werden. Durch die Verbindung eingebetteter Systeme mit der Außenwelt entstehen neue Sicherheitsbedrohungen. In der Vergangenheit wurde wenig Aufwand betrieben diese Systeme vor Angriffen zu schützen. So fehlen in aktuellen Fahrzeuge Authentifikationsmechanismen zwischen verschiedenen Komponenten und die Kopplung der Kommunikationsbusse über gemeinsame Gateways ist hochgradig anfällig für aktive und passive Angriffe [GR09]. Dies ermöglicht Angreifern mittels manipulierter Steuergeräte (ECUs) Nachrichten auf den Kommunikationsbus zu injizieren und so das System zu stören oder zu manipulieren.

Selbst bei einer Neuentwicklung eines Systems ist es unmöglich, alle vorhandenen Sicherheitslücken eines verteilten Systems zur Entwicklungszeit zu finden und zu eliminieren. Des Weiteren kann während des Betriebs die Existenz unzuverlässiger oder sogar bösartiger Kommunikationspartner bzw. Komponenten nicht generell ausgeschlossen werden. Eine zusätzliche Instanz, die die Kommunikation zwischen Komponenten des Systems und die Abläufe der Anwendungen überwacht, kann einen zusätzlichen Nutzen in Bezug auf die Sicherheit eines Systems bieten. Es wurde gezeigt, dass die Laufzeitüberwachung (*engl. runtime monitoring*) eines Systems vorher unbekannte Attacken und Fehler erkennen kann. [Kum95]

Definition 1 (Laufzeitüberwachung und Monitor). Laufzeitüberwachung *ist eine Technik zur Sicherstellung von Eigenschaften eines Systems zur Laufzeit. Basierend auf Spezifikationen, die das Referenzverhalten des Systems oder die bekanntes Fehlverhalten beschreiben, wird das System durch eine zusätzliche Instanz – einen* Monitor *– auf Basis von Ereignissen überwacht. Zur Spezifikation des Referenzverhaltens wird in der Regel eine formal definierte Sprache eingesetzt. Der Monitor läuft parallel zum System und erkennt Abweichungen von der Spezifikation bzw. bekanntes Fehlverhalten möglichst ohne das System selbst zu beeinflussen.*

1.1 Monitorgenerierung aus modellbasierten Beschreibungen

Als Ausgangspunkt der Spezifikation von Laufzeitmonitoren werden häufig Anforderungen an das zu überwachende System aus der Anforderungsanalyse und

[1] C2CCC-Website: http://www.car-to-car.org/

der Spezifikation des Systems genutzt. Die Überwachung dieser Anforderungen zur Laufzeit erhöht die Zuverlässigkeit der entwickelten Systeme [MJG+12].

Basierend auf diesen Informationen werden Monitore häufig nicht mehr auf einem niedrigen Abstraktionsniveau, sondern mittels Protokollspezifikationssprachen auf einem höheren Abstraktionsniveau spezifiziert. Dabei haben sich viele verschiedene Spezifikationssprachen herausgebildet, die eine unterschiedliche Ausdrucksstärke besitzen und verschiedene Modellierungskonzepte anbieten.

Definition 2 (Spezifikationssprache und Zwischensprache). *Der Begriff* Spezifikationssprache *beschreibt in dieser Arbeit Sprachen, die zur Modellierung von Systemen und Monitoren eingesetzt werden. Im Gegensatz hierzu dient eine* Zwischensprache *in erster Linie zur Repräsentation einer Spezifikation und deren Weiterverarbeitung. Bei einer Zwischensprache liegt der Fokus somit nicht auf die Verständlichkeit und Anschaulichkeit der Spezifikation, sondern auf einer möglichst expliziten Repräsentation mit wenigen verschiedenen Modellierungselementen.*

Eingesetzte Spezifikationssprachen sind hierbei unter anderem Dialekte von Finite State Machines (FSM), Temporalen Logiken (TL) und Message Sequence Charts (MSC), die wieder in FSMs übersetzt werden. FSMs haben den Vorteil einer fast konstanten Verarbeitungszeit der überwachten Kommunikationsereignisse. Für komplexe Sequenzen mit mehreren möglichen Alternativen steigt jedoch die Anzahl der zur Modellierung benötigten Zustände exponentiell an. Dies zeigt sich deutlich, wenn FSMs als Zwischensprache für die Codegenerierung aus Live Sequence Charts (LSC) [HKP05] genutzt werden. LSCs sind eine ausdrucksstärkere Variante der Message Sequence Charts (MSCs) [IT00], die die kompakte Spezifikation überlappender quasi-paralleler Kommunikationssequenzen unterstützen. Die Übersetzung dieser LSCs in FSMs benötigt entweder eine exponentiell wachsende Anzahl von Zuständen oder die Einführung vieler paralleler Teil-FSMs mit relativ komplexen Kommunikationsbeziehungen.

Bei der direkten Codegenerierung aus einer solchen Spezifikationssprache ohne Zwischensprache muss für jede Zielsprache und Zielplattform ein Codegenerator geschrieben werden. Dies führt zu einer Vielzahl verschiedener Codegenerierungskonzepte und zu generiertem Code, der an die Syntax und Semantik der Spezifikationssprache angelehnt ist.

1.1.1 *Notwendigkeit einer Zwischensprache für die Monitorgenerierung*

Abbildung 1.1 stellt abstrakt das in dieser Arbeit behandelte Problem dar. Das Ziel ist die Entwicklung eines Prozesses zur Monitorgenerierung aus verschiedensten Spezifikationssprachen für unterschiedliche Zielplattformen, die sich stark voneinander unterscheiden können.

Wie im vorherigen Abschnitt erwähnt und in Kapitel 4 genauer betrachtet, existieren eine Vielzahl verschiedenster Spezifikationssprachen für Laufzeitmonitore. Diese Spezifikationen müssen in einen auf der Zielplattform ausführbaren Monitor übersetzt werden. Wie in [BPP+11] festgestellt wurde, ist es für manche Ein-

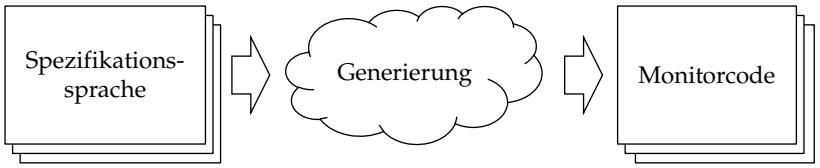


Abbildung 1.1: Monitorgenerierung aus Spezifikationen

satzgebiete sinnvoll, Monitore nicht nur in Software, sondern auch in Hardware umzusetzen. Als zusätzliche Zielplattform kommt somit für zeitkritische Aufgaben auch eine Umsetzung der Monitore in Hardware in Betracht. Hierzu muss die Quellspezifikation in eine logische Schaltung bzw. in eine Hardwarebeschreibungssprache übersetzt werden. Die möglichen Zielplattformen (PC, eingebettete Systeme, Hardware) und damit auch die Zielsprachen (Java, C, VHDL), in die die Signaturen übersetzt werden, sind vielfältig.

Diese Herausforderung unterschiedliche Zielplattformen zu unterstützen, wird bei existierenden Ansätzen häufig umgangen, indem für jede der Plattformen ein speziell angepasstes Rahmenwerk für die Ausführung von Monitorspezifikationen oder von Monitorcode entwickelt wird. Ein solches Rahmenwerk kann auch als Komponente, die die Signaturen in der Spezifikationssprache direkt interpretiert, umgesetzt sein.

Viele der existierenden Ansätze legen eine spezielle Spezifikationssprache fest und es muss für jede zu unterstützende Plattform ein Rahmenwerk oder ein Codegenerator erstellt werden. Eine Möglichkeit zur Unterstützung weiterer Spezifikationssprachen wäre die Übersetzung in die Eingabesprache des für die entsprechende Zielplattform zur Verfügung stehenden Ansatzes. Dabei eignen sich die Spezifikationssprachen, die gut zur Modellierung geeignet sind, jedoch nicht zwangsläufig auch als formale Repräsentation zur Codegenerierung.

Die Kombination aus verschiedenen Eingabesprachen und Zielsprachen bzw. -plattformen würde bei einem solchen Ansatz zu einem großen Anpassungsaufwand führen. Um die Entwicklung spezieller Codegeneratoren für alle Kombinationen aus Spezifikationssprache und Zielplattform zu umgehen, wird in vorhandenen Ansätzen für einen Teil der Spezifikation eine universelle, nicht an die Laufzeitüberwachung angepasste Zwischensprache wie endliche Automaten (FSM) eingesetzt. In diese Zwischensprache werden die Spezifikationen übersetzt und aus dieser Repräsentation in der Zwischensprache der Monitor für die entsprechende Plattform generiert.

Ein erster Ansatz, der eine gemeinsame Zwischensprache verwendet, ist im JavaMOP-Rahmenwerk [MJG+12] zu finden. Neben einer direkten Codegenerierung aus verschiedenen Spezifikationssprachen werden soweit möglich Finite State Machines (FSM) als Zwischensprache eingesetzt, in die die Spezifikationen übersetzt werden. Jedoch sind diese FSMs nicht ausdrucksstark genug, um jede der bei diesem Ansatz betrachteten Spezifikationssprachen abzudecken. Außerdem leiden die FSMs unter einer Zustandsexplosion, wenn viele alternative Abläufe repräsentiert werden sollen. Um die Ausdrucksmächtigkeit zu erhöhen, wird hier eine zusätzliche textuelle Sprache, die zielsprachenabhängig ist und in

die die Spezifikationssprachen (FSMs) eingebettet werden, zur Verfügung gestellt. Hierdurch ist nur ein Teil der Spezifikationssprache formal definiert.

Automaten (FSMs) sind als Zwischensprache, wenn z. B. die ausdrucksstarken Live Sequence Charts als Signaturmodellierungssprache unterstützt werden sollen, nicht geeignet [BS05]. Zur Repräsentation von solchen Spezifikationssprachen wird als universelle Zwischensprache eine formal definierte Sprache benötigt, die parallele bzw. nebenläufige Konzepte beinhaltet.

Bei der Suche nach besser geeigneten Alternativen ist es relativ natürlich auf einer petrinetz-ähnlichen Sprache, die für die Modellierung parallel arbeitender Systeme entwickelt wurde, aufzusetzen. *Monitor-Petrinetze* (MPNs), die in dieser Arbeit eingeführt werden, sind eine erweiterte Form der Petrinetze, die auf die Anforderungen der Laufzeitüberwachung angepasst wurden. Sie unterstützen die speziellen Anforderungen (Abschnitt 3.5), die für eine Modellierungssprache für Signaturen und für eine Zwischensprache für die Codegenerierung notwendig sind.

Eine solche formal definierte Zwischensprache bietet weitere Vorteile gegenüber der direkten Übersetzung der Spezifikation in Code. Zum einen kann die in dieser Arbeit formal definierte neue Zwischensprache *Monitor-Petrinetze* (MPN) durch Spezifikation einer Transformation von einer beliebigen Spezifikationssprache zu den MPNs eingesetzt werden, um die Semantik der Spezifikationssprache, wie in [Pat14] durchgeführt, selbst zu formalisieren. Zum anderen erleichtert eine gemeinsame Zwischensprache die in dieser Arbeit behandelte Generierung von Laufzeitmonitoren. Eine getestete und validierte Übersetzung aus der gemeinsamen Zwischensprache in den plattformspezifischen Code kann für verschiedene Spezifikationssprachen wiederverwendet werden. Durch eine Modell-zu-Modell-Transformation in die Zwischensprache kann somit aus verschiedenen Spezifikationssprachen mit reduziertem Aufwand im Gegensatz zur direkten Generierung ein lauffähiger und optimierter Monitor generiert werden. Durch die gemeinsame Zwischensprache haben die generierten Monitore eine übersichtliche grundlegend identische Struktur und sind hierdurch besser wartbar.

1.1.2 *Der MBSecMon-Entwicklungsprozess*

Die für das Monitoring neu entwickelte Zwischensprache *Monitor-Petrinetze* wird in den *Model-Based Security and Safety Monitor* (MBSecMon)-Entwicklungsprozess eingebettet, der an den *Model-Driven Engineering* (MDE)-Ansatz[2] angelehnt ist. Dieser Prozess verläuft parallel zu dem für die Systementwicklung eingesetzten Entwicklungsprozess. Im Folgenden wird der MBSecMon-Prozess zusammen mit dem Systementwicklungsprozess *V-Modell XT* [BTZ05, Bun12] betrachtet.

Das *V-Modell XT* (eXtreme Tailoring) ist ein Standard zur Planung und Durchführung von IT-Systementwicklungsprojekten der öffentlichen Hand in der Bundesrepublik Deutschland. Es ist eine Weiterentwicklung des *V-Modell 97* [BMI97], ist jedoch in Hinblick auf die Abfolge der Prozessschritte anpassbar und somit z. B. auch für die agile Softwareentwicklung einsetzbar. Das V-Modell XT besteht grundlegend aus den drei Phasen:

[2] OMG MDA-Website: http://www.omg.org/mda/

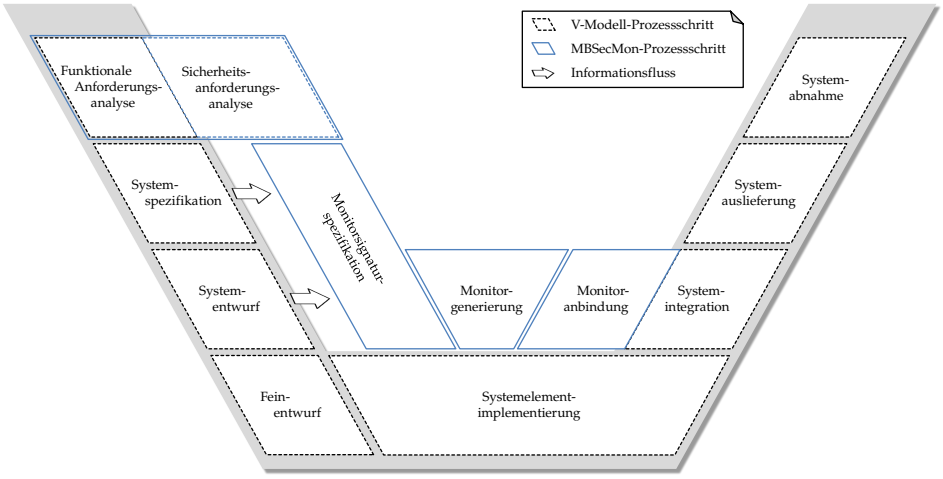


Abbildung 1.2: Der MBSecMon-Prozess im V-Modell

- Spezifikation und Zerlegung
- Realisierung
- Integration

Abbildung 1.2 zeigt den MBSecMon-Entwicklungsprozess, der in einer Variante des V-Modell XT aus der Sicht des Auftragnehmers eingebettet ist. Die Prozessschritte des V-Modells sind grau hinterlegt dargestellt, während sich die Schritte des MBSecMon-Prozesses innerhalb des V-Modells befinden. Das hier dargestellte V-Modell bezieht sich nur auf die Softwareentwicklung und spart Projektmanagement, Qualitätssicherung und Konfigurationsmanagement größtenteils aus.

SPEZIFIKATION UND ZERLEGUNG Der erste Schritt des V-Modells ist die *Anforderungsanalyse*. In dieser werden zunächst die funktionalen sowie nicht-funktionalen Anforderungen des zu entwickelnden Systems in einem Lastenheft festgehalten. Des Weiteren beinhaltet die Anforderungsanalyse die *Systemanalyse*. In ihr wird das System, in dem das zu entwickelnde System eingesetzt werden soll, spezifiziert, sowie eine Skizze des zu entwickelnden Systems erstellt. Hierzu werden die objektorientierte Analyse mithilfe von UML-Diagrammen oder die funktionale Analyse durchgeführt. Unter die objektorientierte Analyse fallen u. a. die Anwendungsfallmodellierung, die Klassen-/Objekt-Modellierung und die Interaktionsmodellierung.

In der *Systemspezifikation* wird zusätzlich zu den Anforderungen ein Grobentwurf des zu entwickelnden Systems erstellt. Hierbei werden u. a. Schnittstellen definiert und eine grobe Gesamtsystemarchitektur entwickelt. In diesem Schritt werden wiederum die angesprochenen funktionalen oder objektorientierten Techniken eingesetzt. Die Anforderungen und der Grobentwurf werden in der Gesamtsystemspezifikation (Pflichtenheft) dokumentiert.

Darauf folgt der *Systementwurf*, in dem der Grobentwurf verfeinert und auf Konsistenz überprüft wird, sodass allein auf Grundlage dessen die Entwicklung der Systemelemente (Module) durchgeführt werden kann. Um dies zu erreichen,

werden die Schnittstellen, die einzusetzenden Konzepte und das Verhalten der Systemelemente nach außen spezifiziert. Zusätzlich wird festgelegt, wie die Implementierung stattfinden soll.

REALISIERUNG Der *Feinentwurf* baut auf den bis jetzt erstellten Spezifikationen für externe sowie interne Hardware und Software auf und verfeinert diese vollständig. Anhand dieser Spezifikationen werden anschließend in der *Systemelementimplementierung* die Systemelemente realisiert und anschließend gegen die Spezifikation des Schrittes *Feinentwurf* verifiziert.

INTEGRATION Im Schritt *Systemintegration* werden die realisierten Systemelemente zusammengefasst und die Interaktion zwischen ihnen getestet. Für diese Integrationstests wird auf die Spezifikationen des Schrittes *Systementwurf* zurückgegriffen. Am Ende dieses Schrittes liegt das System in einer auslieferbaren Form vor.

In der *Systemauslieferung* findet nach Prüfung der Funktionalität des Produktes auf Basis der *Gesamtsystemspezifikation* und der Dokumentation die Auslieferung an den Auftraggeber statt. Abschließend wird in der *Systemabnahme* das entwickelte System durch den Auftraggeber abgenommen, der diese Abnahme auf Basis der im Lastenheft spezifizierten Szenarien durchführt.

MBSECMON-PROZESS Im Gegensatz zu der gewöhnlich relativ späten Betrachtung von Sicherheitsaspekten im Softwareentwicklungsprozess ist es vorzuziehen, die Monitore schon in der Anforderungs- und Designphase zu spezifizieren [MFMP06]. Hierfür wurde der MBSecMon-Entwicklungsprozess – ein modellbasierter Entwicklungsprozess für die Spezifikation und Generierung von Software- und Hardwaremonitoren – entwickelt. Der MBSecMon-Prozess ist parallel zu dem verwendeten Entwicklungsprozess spezifiziert, um eine Unabhängigkeit des Monitorentwicklungsprozesses vom Softwareentwicklungsprozess zu erreichen. Hierbei sind die vier Prozessschritte *Sicherheitsanforderungsanalyse*, *Monitorsignaturspezifikation*, *Monitorgenerierung* und *Monitoranbindung* definiert.

Die *Sicherheitsanforderungsanalyse* spezifiziert zusätzlich zu den in der *Anforderungsanalyse* erstellten funktionalen Anforderungen nicht-funktionale sicherheitsrelevante Szenarien, die z. B. Angriffe oder unerwünschtes Verhalten beschreiben. Zur Spezifikation der Monitorsignaturen z. B. als Szenarien können im Schritt *Monitorsignaturspezifikation* neben den in der Anforderungsanalyse gewonnenen Anforderungen auch verfeinerte Szenarien sowie die Systemarchitektur aus den Schritten *Systemspezifikation* und *Systementwurf* verwendet werden. Hierfür wird die MBSecMon-Spezifikationssprache [Pat14] eingesetzt. Zum einen besteht sie aus der zur Strukturierung der Spezifikation eingesetzten MUC-Sprache, in der positive Anforderungen als Usecases und negative als Misusecases modelliert werden. Diese Usecases und Misusecases können zusätzlich in Beziehung zueinander gesetzt werden. Zum anderen werden diese abstrakten Beschreibungen des erlaubten und verbotenen Verhaltens in einer erweiterten Version der Live Sequence Charts (eLSC), einer ausdrucksstärkeren Variante der Message Sequence

Charts (MSC), modelliert. Diese eLSCs dienen zur Beschreibung von Mustern (Signaturen), die erlaubtes und verbotenes Systemverhalten repräsentieren.

Definition 3 (Signatur/Signaturspezifikationssprache). *In dieser Arbeit definiert eine Signatur, die in einer Signaturspezifikationssprache modelliert ist, eine Menge von Ereignissequenzen auf einer abstrakten Modellierungsebene. Signaturspezifikationssprachen können im Gegensatz zu Protokollspezifikationssprachen, die nur positive (erlaubte) Sequenzen beschreiben – sie bieten keine Sprachkonstrukte für die explizite Beschreibung verbotener Abläufe –, auch negative (verbotene) Sequenzen beschreiben. In Kombination bilden diese Signaturen eine Menge erlaubter und verbotener Abläufe, die durch Laufzeitmonitore überwacht werden können.*

Die MUC-Sprache dient zur Strukturierung und Wiederverwendung dieser Signaturen, die als eLSCs spezifiziert sind. Durch die hohe Ausdrucksstärke dieser MBSecMon-Sprache sind die beschriebenen Signaturen zwar einfach zu verstehen und kompakt zu modellieren, jedoch für die folgende Monitorgenerierung komplex zu interpretieren.

Im nächsten Schritt findet auf Basis der Monitorsignaturen die *Monitorgenerierung* statt. Hierfür wird die Monitorspezifikation zunächst in die universelle Zwischensprache MPN transformiert und anschließend werden Optimierungen auf den übersetzten Signaturen durchgeführt. Auf Basis dieser optimierten Zwischenrepräsentation werden weitgehend unabhängig von der eigentlichen *Systemelementimplementierung* Monitore generiert. Durch Bereitstellung zusätzlicher Informationen über das Zielsystem und die Zielplattform wird die Monitorgenerierung konfiguriert.

Die *Monitoranbindung* an das entwickelte System findet im Schritt der *Systemintegration* des V-Modells statt. Hierbei wird der generierte Monitor an das zu überwachende System angebunden und somit während der Laufzeit des Systems mit Ereignissen versorgt.

1.2 Einsatzbereich des Ansatzes

Nachdem in Abschnitt 1.1.2 der MBSecMon-Entwicklungsprozess, ein paralleler Ansatz zur Codegenerierung aus Monitorspezifikationen, vorgestellt wurde, wird in diesem Abschnitt die Gültigkeit dieses Prozesses betrachtet. Der Fokus des MBSecMon-Prozesses liegt auf der Generierung von Monitoren zur Überwachung von Systemen auf einer globalen Ebene. Hierbei wird die Kommunikation über mehrere Teilkomponenten hinweg überwacht. Zusätzlich zum Einsatz robuster Protokolle und sicherer Architekturen, wie sie in [GR09, KCR+10] gefordert werden, wird durch die Laufzeitmonitore des MBSecMon-Prozesses ein erweiterter übergreifender Schutz gegen Angriffe und Fehlverhalten des überwachten Systems erreicht. Des Weiteren unterstützt dieser Ansatz die lokale Überwachung einzelner Komponenten und Schnittstellen, die nicht unter dem Aspekt der Sicherheit entwickelt wurden.

ANNAHMEN AN DIE SPEZIFIKATIONSSPRACHE UND DAS ZU ÜBERWACHENDE SYSTEM Zur Spezifikation der Signaturen wird eine ausdrucksstarke Spezifikationssprache, die Konzepte zur Beschreibung paralleler Anläufe unterstützt, eingesetzt. Zusätzlich unterstützt die Spezifikationssprache die Unterscheidung von verschiedenen Varianten eines zu überwachenden Ablaufs durch optionale Signaturanteile und Vorbedingungen.

Das zu überwachende System verfügt potenziell über stark nebenläufige Kommunikationsabläufe und parallele Kommunikationsstränge. Die dieses System überwachenden Laufzeitmonitore müssen auf verschiedensten Plattformen, auch solchen mit eingeschränkter Rechenleistung und eingeschränkten Speicherressourcen, lauffähig sein.

Es wird angenommen, dass das zu überwachende Verhalten durch Techniken wie Instrumentierung oder zusätzliche Module an Kommunikationsbussen vom Monitor erfasst werden kann. Der Monitor läuft entweder parallel zu dem zu überwachenden System oder wird durch Instrumentierung (aspektorientierte Programmiertechniken oder Wrapping) in dieses integriert.

Zum universellen Einsatz der entwickelten Zwischensprache müssen verschiedenste Zielplattformen und Zielsprachen in dem in dieser Arbeit entwickelten Prozess unterstützt werden. Aus diesem Grund muss die Zwischensprache programmiersprachenunabhängig und für verschiedenste Zielplattformen geeignet sein. Neben Software- werden auch Hardwaremonitore benötigt, um Ereignisse bei hohen Kommunikationsraten in Echtzeit überwachen zu können. Deshalb werden stellvertretend die Zielplattformen PC, Mikrocontroller und FPGA sowie die Zielsprachen Java, C und VHDL evaluiert.

ZU ERKENNENDES FEHLVERHALTEN UND ZU ERKENNENDE ANGRIFFE Die erkennbaren Angriffe und das erkennbare Fehlverhalten des in dieser Arbeit vorgestellten Ansatzes, hängen sowohl von der Ausdrucksstärke der verwendeten Signatursprache, die zur Beschreibung der Monitore verwendet wird, als auch von der eingesetzten Zwischensprache ab.

Definition 4 (Sicherheit – Safety und Security). *Sicherheit teilt sich in die Bereiche Safety und Security auf. Während* Safety *(Betriebssicherheit), die Absicherung von Fehlverhalten des Systems durch unbekannte und zufällige Fehler im System oder durch unbeabsichtigte Fehlbedienung behandelt, beschäftigt sich* Security *(Angriffssicherheit) mit der Abwehr von mutwilligen Angriffen auf ein System.*

Der Fokus dieser Arbeit liegt auf dem Bereich Safety. Durch den MBSecMon-Ansatz werden jedoch auch Teile der Sicherheitsziele der Security abgedeckt.

Umzusetzende Safety-Maßnahmen umfassen dabei Überwachungsmechanismen für Kommunikationen und Programmabläufe, jedoch keine weiteren architektonischen Maßnahmen wie z. B. redundante Ausführung und diversitäre Programmierung. Der auf Ereignissen basierende Ansatz kann die Reihenfolge und das Timing beliebiger Übergänge im Zustand des Systems und im Ablauf von Kommunikationen überwachen, wenn diese als Ereignisse dem Monitor zur Verfügung stehen. Zusätzlich können die bei einer Kommunikation übermittelten

Nachrichteninhalte und an den Monitor übergebenen Variablenwerte, die z. B. durch Instrumentierung des Systems gewonnen werden, in die Überwachung einbezogen werden.

Im Bereich Security werden von diesem Ansatz insbesondere die Teilbereiche Protokollüberwachung, System- und Datenintegrität abgedeckt. Hierbei können Abweichungen von definierten Protokollen und bekannte Angriffe erkannt werden, die als positive bzw. negative Signaturen spezifiziert wurden. Die Reihenfolge der Nachrichten und deren Timing sowie die Plausibilität der übermittelten Daten sind mögliche Überwachungsziele.

Die Überwachung der Sicherheitsziele der Security wird im MBSecMon-Entwicklungsprozess wie im Bereich Safety durch Generierung von Monitoren, die das System auf Basis von Signaturen überwachen, umgesetzt. Diese Monitore erlauben die teilweise Sicherstellung der Systemintegrität, der Datenintegrität und der Verfügbarkeit.

Systemintegrität bedeutet in diesem Fall, dass sich das System jederzeit nachvollziehbar und logisch korrekt verhält. Dieses Ziel kann durch Überwachung der Reihenfolge und des Timings der Ereignisse sowie des internen Zustands des Systems gegenüber positiven und negativen Signaturen erreicht werden.

Im Bereich der *Datenintegrität* können Daten aus dem System oder von Nachrichten, die dem Monitor zusammen mit den Ereignissen übergeben werden, plausibilisiert werden. Ein Schutz vor Manipulation kann nur bei Abweichung der Werte von der Spezifikation (Signaturen) stattfinden.

Die *Verfügbarkeit* eines Systems und die Verfügbarkeit von Datenwerten kann durch einen externen Monitor erreicht werden. Hierzu wird das zeitliche Verhalten des Systems und das Timing zwischen Nachrichten anhand der ihm übergebenen Ereignisse überwacht und es werden bei einem zeitlichen Fehlverhalten Gegenmaßnahmen eingeleitet.

Weitere Sicherheitsziele können durch Funktionalitäten externer domänenspezifischer Bibliotheken realisiert werden, die in Signaturen über Aktionen und Bedingungen eingebunden werden. Diese sind jedoch nicht im Fokus dieser Arbeit.

Nach der Erkennung von Fehlverhalten oder Angriffen sollen durch den MBSecMon-Ansatz gezielte Gegenmaßnahmen initiiert werden. Solche Gegenmaßnahmen, die auf dem Zustand der Überwachung und den auf die Erkennung eines Fehlverhaltens folgenden Ereignissen basieren, werden zur Durchsetzung der Sicherheitsziele eingesetzt.

1.3 Beitrag

Der MBSecMon-Entwicklungsprozess wurde im Rahmen zweier Promotionen entwickelt. Abbildung 1.3 zeigt den Prozess mit Fokus des in dieser Arbeit behandelten Themas der Monitorgenerierung aus Spezifikationen. Die Definition der MBSecMon-Spezifikationssprache (MBSecMonSL) und die Transformation in die Zwischensprache MPN wird durch die Arbeit von Sven Patzina in [Pat14] betrachtet. Durch die Abbildung der MBSecMonSL auf die formal definierten MPNs wird diese formalisiert.

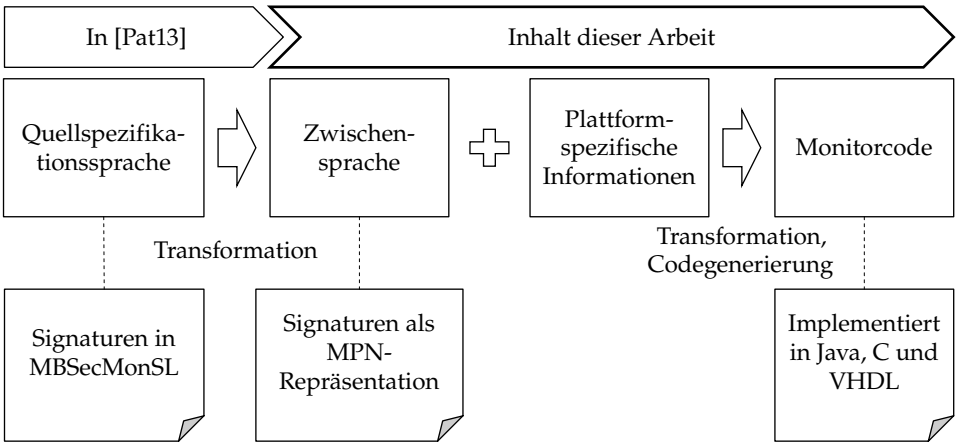


Abbildung 1.3: Überblick über den Monitorgenerierungsprozess

In dieser Arbeit steht die Definition der neuen MPN-Sprache sowie die Generierung effizienter Monitore aus dieser Zwischensprache im Vordergrund. Hierbei wird die Spezifikation in der MPN-Sprache zunächst durch eine Transformation an die Zielplattform angepasst und optimiert. Die benötigten Zusatzinformationen zu den Eigenschaften der Zielplattform werden durch plattformspezifische Informationen (PSI) dem Prozess übergeben. Anschließend wird durch einen Codegenerator, der die optimierten plattformunabhängigen MPNs und die PSI als Eingabe übergeben bekommt, der Monitor bzw. die Monitore für die entsprechende Zielplattform generiert.

Diese Arbeit beschreibt einen mehrschrittigen Entwicklungsprozess für Laufzeitmonitore. Es werden Anforderungen an eine Zwischensprache, die in einem solchen Entwicklungsprozess eingesetzt werden kann, auf Basis von existierenden Ansätzen und dem zu unterstützenden Prozess aufgestellt. Hierbei wird zur Unterstützung von beliebigen Spezifikationssprachen die neue an das Monitoring angepasste Zwischensprache MPN formal definiert und im MBSecMon-Prozess eingesetzt. Die MPN-Sprache ist eine auf Petrinetzen basierte Zwischensprache, die parallele und hoch nebenläufige Abläufe effizient überwachen kann.

Hierzu besitzen die MPNs eine formal definierte Syntax und Semantik. Im Gegensatz zu allgemeinen Formalismen, die als Zwischensprache in anderen Ansätzen eingesetzt werden, sind die MPNs an die Domäne der Laufzeitüberwachung angepasst. Die MPNs enthalten alle Konzepte, die für die Beschreibung von Monitoren benötigt werden, wobei eine möglichst explizite Repräsentation der Signaturen, eine Syntax mit wenigen verschiedenen Elementen sowie eine eindeutige Semantik im Fokus steht. Es ist nicht wie bei anderen Ansätzen notwendig, zusätzliche Beschreibungssprachen einzusetzen, die nicht formalisiert sind. Hierdurch ist es möglich, durch eine Transformation in die MPN-Sprache beliebige auch nicht formal definierte Spezifikationssprachen zu formalisieren. An den Transitionen annotierte Bedingungen und Berechnungen sind hierbei ausgenommen und müssen durch eine formalisierte Annotationssprache beschrieben werden.

Die MPNs unterstützen selbst die Abbildung komplexer Spezifikationssprachen, wie die eLSCs, die für eine übersichtliche und verständliche Modellierung von Signaturen entwickelt wurden. Eine Abbildung paralleler Abläufe ist, im Gegensatz zu Sprachen wie den FSMs, in den MPNs, da sie auf Petrinetzen basieren, mit deutlich weniger Elementen in der Repräsentation verbunden. Sie besitzen im Gegensatz zu Petrinetzen eine deterministische Ausführungssemantik, die für die Beschreibung von Monitoren ausreichend ist. Hierdurch ist die Übersetzung von der Spezifikationssprache in die MPN-Sprache durch die reduzierte Komplexität weniger fehleranfällig.

Zusätzlich zu dieser grundlegenden Version der MPN-Sprache wird ein optionales Referenzsystem eingeführt, das die Wiederverwendbarkeit von Teilen der Signaturen unterstützt. Es können Signaturen in Beziehung zum Zustand anderer Signaturen gesetzt werden und so der Überwachungszeitpunkt der Signatur definiert werden. Dieses Konzept beinhaltet die Spezifikation von Gegenmaßnahmen bei Erkennen von Abweichungen von erlaubtem oder bei Eintreten von verbotenem Verhalten.

Des Weiteren beschäftigt sich diese Arbeit mit der Generierung effizienter Monitore aus verschiedenen Spezifikationssprachen über die Monitor-Petrinetzsprache. Hierzu werden die generierten Monitore auf Skalierbarkeit analysiert und mit verwandten Ansätzen verglichen. Es wird gezeigt, dass sich der Formalismus der MPNs sowohl in Speicherverbrauch als auch in Laufzeit der generierten Monitore für den Einsatz auf PCs, ressourcenbeschränkten eingebetteten Systemen und zur Umsetzung in Hardware eignet.

Zusammenfassend sind die Beiträge dieser Arbeit:

- Die Analyse existierender Ansätze und Ableitung von Anforderungen an eine Zwischensprache
- Die Entwicklung und Formalisierung einer an das Monitoring angepassten Zwischensprache – Monitor-Petrinetze
  - Die grundlegende Definition der Monitor-Petrinetze zur Abbildung höchst nebenläufiger Prozesse
  - sowie die Definition eines optionalen Referenzsystems, durch das Teilsignaturen zueinander in Beziehung gesetzt werden können.
- Die Umsetzung des Ansatzes in einem Entwicklungsprozess
- Die Evaluation des Ansatzes und der Zwischensprache für verschiedene Zielsprachen und Zielplattformen

1.4 Gliederung der Arbeit

Diese Arbeit gliedert sich in fünf Teile, die in Abbildung 1.4 dargestellt sind. Im Folgenden wird genauer auf die Teile der Arbeit eingegangen.

Teil I: Einleitung In diesem Teil wird in die Thematik eingeführt und das durchgehende Beispiel beschrieben.

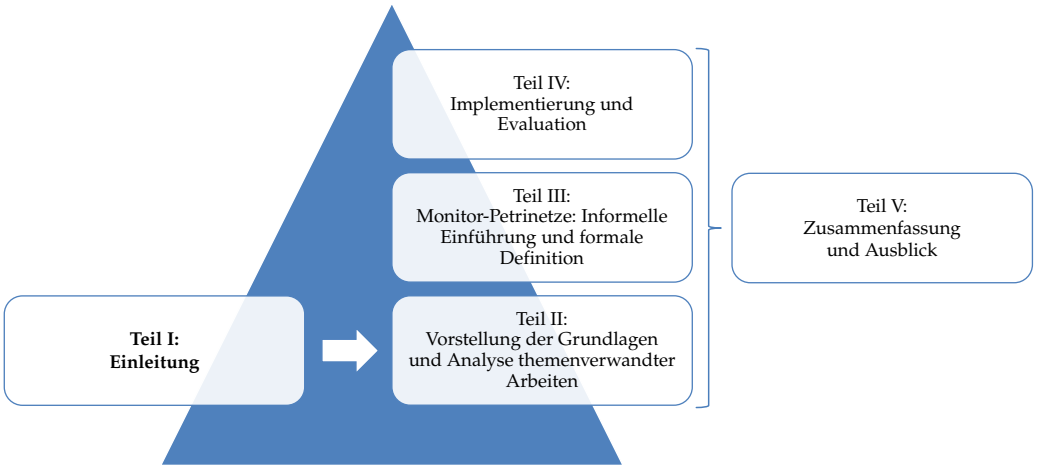


Abbildung 1.4: Strukturierung der Arbeit

Kapitel 1: Zusätzlich zu der in diesem Kapitel gegebenen Einleitung mit Motivation und der Vorstellung des entwickelten Prozesses, in den der Beitrag dieser Arbeit einfließt, wird der Beitrag dieser Arbeit beschrieben. An diesen Abschnitt, der die Gliederung der Arbeit beschreibt, anschließend wird das Szenario eingeführt, das als durchgängiges Beispiel in dieser Arbeit eingesetzt wird.

TEIL II: VORSTELLUNG DER GRUNDLAGEN UND ANALYSE THEMENVERWANDTER ARBEITEN In diesem Teil der Arbeit werden die Grundlagen in Bezug auf die Einsatzgebiete der Monitore und die verwendeten Spezifikationssprachen für Monitore betrachtet. Hieraus werden die Anforderungen an eine Zwischensprache zur Monitorgenerierung abgeleitet und die Modellierungsansätze in Bezug auf diese Anforderungen evaluiert.

Kapitel 2: In diesem Kapitel wird die Technik eines FPGAs, das als eine der Zielplattformen dient und die AUTOSAR-Methodik, in die der Generierungsprozess integriert wird, vorgestellt.

Kapitel 3: Ausgehend von den Grundlagen der in dieser Arbeit verwendeten Modellierungssprachen und Konzepte werden Anforderungen, die eine Zwischensprache für Monitorspezifikationen erfüllen muss, aufgestellt. Diese resultieren aus den Kategorien Ausdrucksstärke der Sprache, Modellierung der Signaturen sowie Generierung und Ausführung der Monitore für und auf einem Zielsystem.

Kapitel 4: In der Vergangenheit wurde eine Vielzahl an Modellierungssprachen und Zwischensprachen zur Monitorbeschreibung eingesetzt. Hierunter befinden sich Monitorspezifikationssprachen sowie Ansätze der Monitorgenerierung und Überwachung von Spezifikationen. Diese verwandten Ansätze und Sprachen werden kategorisiert und nach den aufgestellten Anforderungen an eine Zwischensprache für den MBSecMon-Prozess bewertet.

TEIL III: MONITOR-PETRINETZE: INFORMELLE EINFÜHRUNG UND FORMALE DEFINITION Auf Basis der im vorherigen Teil ermittelten Anforderungen an eine Zwischensprache wird in diesem Teil eine Sprache entwickelt und formalisiert, die als universelle Zwischensprache für die Monitorgenerierung dienen soll.

Kapitel 5: Es wird zunächst eine grundlegende Version der Monitor-Petrinetze eingeführt. Diese besitzt eine deterministische Semantik und unterstützt die parallele Überwachung verschiedener Kommunikationsstränge. Anschließend wird diese Formalisierung um ein Konzept zur verschränkten Überwachung einzelner Signaturen erweitert.

Kapitel 6: Aufbauend auf der MPN-Formalisierung wird ein Konzept zur Modellierung von Abhängigkeiten zwischen einzelnen MPN-Signaturen eingeführt – das Referenzsystem. Anschließend werden die Umgebung, Variablen sowie die Zeit in MPNs eingeführt. Zur Spezifikation von Gegenmaßnahmen im Referenzsystem wird beschrieben, wie ausführbare MPNs definiert werden. Für die Überwachung verteilter Systeme ist es notwendig, die ganzheitlich modellierten Monitorspezifikationen in Teilmonitore aufzuteilen. Hierzu wird besprochen, wie dies mit der MPN-Sprache umgesetzt werden kann.

TEIL IV: IMPLEMENTIERUNG UND EVALUATION Nach der Formalisierung wird die Implementierung des Prozesses und der Generierung der Monitore beschrieben. Diese werden anschließend für verschiedene Zielsprachen auf verschiedenen Zielplattformen auf ihre Einsetzbarkeit und das Verhalten der MPNs evaluiert.

Kapitel 7: Dieses Kapitel beschreibt die Umsetzung des MBSecMon-Rahmenwerks mit dem Fokus auf den MPN- und Monitorgenerierungsanteil. Hierbei wird die Anpassung des UML-Modellierungswerkzeugs Enterprise Architect sowie die Implementierung des MPN-Codegenerators betrachtet. Abschließend werden die im Prototypen umgesetzten Zielplattformen und Zielsprachen betrachtet.

Kapitel 8: Die aus der Spezifikationssprache in die MPN-Sprache übersetzten Signaturen werden anschließend in Code übersetzt. In diesem Kapitel werden das Konzept der Codegenerierung und die durchgeführten Optimierungen am Beispiel der C-Codegenerierung vorgestellt. Zusätzlich zu den MPNs werden plattformspezifische Informationen, für die Generierung von plattformspezifischem Monitorcode benötigt.

Kapitel 9: Nach Vorstellung der Codegenerierung und der Struktur des Codes findet in diesem Kapitel die Evaluation der generierten Monitore statt. Hierbei werden verschiedenste Zielsprachen (Java, C und VHDL) sowie Zielplattformen (PC, Mikrocontroller und FPGA) miteinander verglichen. Die für Java generierten Monitore werden einem sehr ähnlichen Ansatz gegenübergestellt. Die Monitorgenerierung wird außerdem als realitätsnahes Beispiel in die AUTOSAR-Methodik integriert und an diese Domäne angepasster Monitorcode generiert. Die Anbindung des Monitors an das System durch Instrumentierung des Codes als auch die Anbindung über aspektorientierte Programmierung wird betrachtet. Abschließend wird die MPN-Sprache mit endlichen Automaten verglichen, um einen

Eindruck über die Stärken und Schwächen der MPN-Sprache im Einsatz in Monitoren zu gewinnen.

TEIL V: ZUSAMMENFASSUNG UND AUSBLICK Abschließend wird ein Fazit der durch diese Arbeit erreichten Ziele gezogen und ein Ausblick auf weitere Erweiterungen und Forschungsgebiete im Bezug auf die Arbeit gegeben.

Kapitel 10: Auf Grundlage der in Teil II aufgestellten Anforderungen an eine Zwischensprache für die Monitorgenerierung wird die vorgestellte MPN-Sprache bewertet. Es wird besprochen, inwieweit die Anforderungen erfüllt werden konnten und welche in Zukunft noch zu erfüllen sind. Anschließend werden in einem Ausblick mögliche Erweiterungen der Sprache und zusätzliche Einsatzgebiete der MPNs besprochen.

1.5 DAS CAR2X-MAUTBRÜCKENZENARIO

Als durchgängiges Beispiel wird in dieser Arbeit eine Kommunikation zwischen einem Fahrzeug und einer Mautbrücke betrachtet. Zur Kommunikation wird ein Protokoll benötigt, das den Ablauf der Kommunikation festlegt.

Es gibt viele verschiedene „Electronic Fee Collection“ (EFC)-Systeme, die auf ein On-Bord-Equipment (OBE), das mit einer Mautbrücke kommuniziert, setzen. Das bekannteste und in Deutschland eingesetzte EFC-System ist Toll Collect, ein streckenbasiertes, satellitengestütztes Freeflow-System, das jedoch nicht ausreichend öffentlich dokumentiert ist. Deshalb wird in dieser Arbeit das CARDME-4 Protokoll als durchgehendes Beispiel verwendet. Es ist in EN ISO 14906 in Anhang B und in [Oeh02] spezifiziert und beschrieben.

CARDME-4 definiert ein Rahmenwerk für einen vollständig kompatiblen EFC-Service auf Basis zentraler Zahlungskonten. Es ist mit dem Ziel entwickelt worden, sich in bestehende Systeme auch über Landesgrenzen hinweg integrieren zu können. Hierzu wird eine EFC-Applikation definiert, die parallel zu bestehenden Systemen laufen kann. Der Vorteil dieser Art der Einbindung ist, dass Nutzer, die die Vorteile des CARDME-4-Protokolls, wie z. B. die grenzübergreifende Kompatibilität (Roaming), nutzen wollen, eine zusätzliche OBE in ihr Fahrzeug einbauen können. Für alle anderen lokalen Nutzer ändert sich nichts. Des Weiteren ist CARDME-4 als Basis für die Neuentwicklung von EFC-Diensten vorgesehen.

In Abbildung 1.5 sind die möglichen Einsatzgebiete des CARDME-4-Protokolls grafisch dargestellt. CARDME-4 ist unabhängig von der verwendeten Hardware und anwendbar für offene und geschlossene Systeme in Einspurumgebungen oder Freeflow-Mehrspurumgebungen. Das Protokoll sieht von sich aus keine Verschlüsselung vor, erlaubt jedoch eine Erweiterung um kryptografische Sicherheitsmaßnamen auf den OBUs, falls dies von den Betreibern gewünscht wird. Bei einem offenen Mautsystem wird bei jedem Durchfahren einer Mautbrücke (RSE) eine Gebühr abgerechnet. Hierbei spielen die gefahrene Strecke und die vorher passierten Mautbrücken keine Rolle. Im Gegensatz dazu erhält der Nutzer in einem geschlossenen System beim Einfahren in einem mautpflichtigen Bereich ein Einfahrtsticket, das in der OBE gespeichert wird. Dieses Ticket wird beim Verlas-

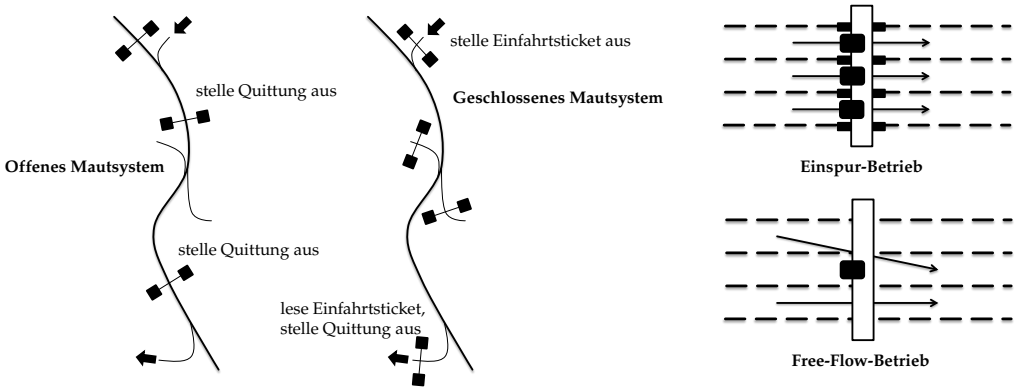


Abbildung 1.5: Einsatzgebiete des CARDME-4-Protokolls nach [Oeh02]

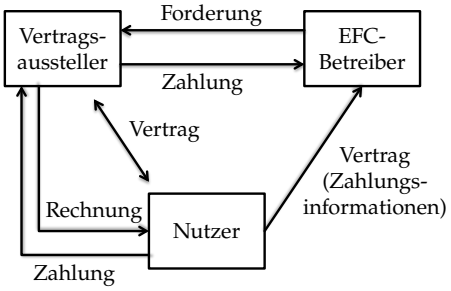


Abbildung 1.6: Architektur des CARDME-4-Protokolls nach [Oeh02]

sen dazu verwendet, die gefahrene Strecke und damit die anfallenden Gebühren zu bestimmen.

Das Protokoll sieht zwei verschiedene Domänen vor:

- Die Heimatdomäne, in der der Vertragsaussteller (Contract Issuer), mit dem der Nutzer einen Vertrag hat, das EFC-System betreibt.
- Die Fremddomäne, in der verschiedenste „fremde“ Betreiber und verschiedene EFC-Systeme, die das CARDME-Protokoll unterstützen, existieren.

Abbildung 1.6 zeigt die grundlegende Architektur des Protokolls. In der Heimatdomäne kommuniziert der Nutzer direkt mit seinem Vertragsaussteller. Ist der Nutzer in einer Fremddomäne, präsentiert er den Vertrag seines eigenen Vertragsausstellers, aus dem der „fremde“ EFC-Betreiber die Informationen extrahiert, die er benötigt, um eine Forderung an den ursprünglichen Vertragsaussteller zu stellen. Die Abrechnung findet gesammelt zwischen dem Nutzer und dem Vertragsaussteller statt.

Die Mautstation (Road Side Equipment) sendet zyklisch eine standardisierte Nachricht, auf die ein Fahrzeug (On-Board Equipment), das diese empfängt, antwortet. Das anschließende Protokoll besteht grundlegend aus fünf Phasen, wobei zwei der Phasen optional sind:

1. **Initialisierung:** Auch der Nutzer kann mehrere EFC-Verträge besitzen, von denen der EFC-Betreiber nicht alle unterstützt. In dieser Phase findet eine Einigung über den zu verwendenden EFC-Vertrag statt.

2. **Präsentation 1:** Das RSE liest die OBE-Daten, in denen unter anderem Details zum Vertrag, zum Konto, dem Fahrzeugtyp und der letzten durchgeführten Transaktion stehen.

3. **Präsentation 2 (optional):** Falls es sich um einen fremden Anbieter (nicht lokalen) handelt, findet eine zweite Präsentationsphase statt, in der Daten zum fremden Anbieter übermittelt werden.

4. **Belegphase:** Das RSE stellt eine elektronische Quittung aus, die bei späteren RSE verwendet wird, um die gefahrene Strecke und damit die Gebühren zu bestimmen.

5. **Verfolgen- und Schließenphase (optional):** Das RSE verfolgt das Fahrzeug, während es im Kommunikationsbereich ist, und schließt eventuell die Transaktion ab.

Bei jeder Belegphase werden die alten gespeicherten Quittungen gelesen und es wird eine neue Quittung geschrieben. Je nachdem, ob es sich um ein geschlossenes oder offenes System handelt, werden die alten Quittungen von der RSE ignoriert oder zur Bestimmung der gefahrenen Strecke herangezogen.

CARDME bietet folgende optionale Sicherheitsmaßnahmen:

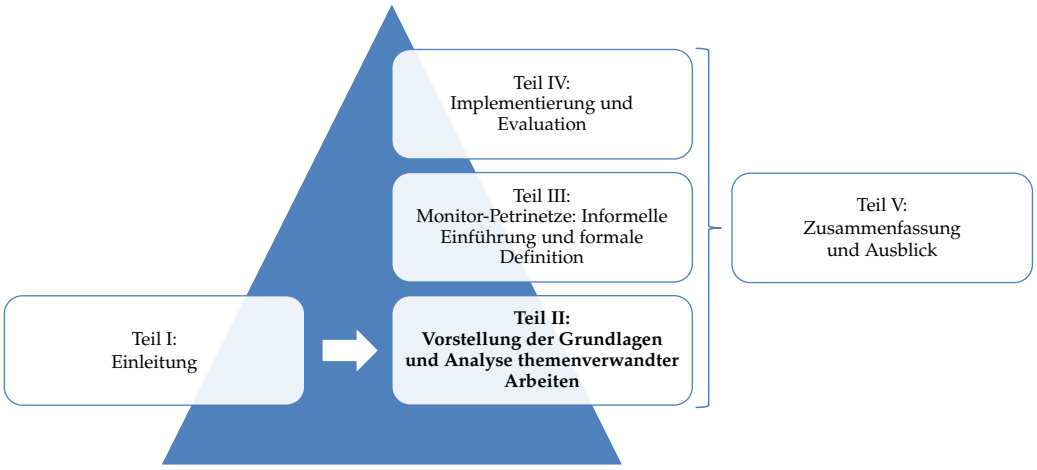
- **Integritätsservice** zur Absicherung vor unautorisierten Änderungen an den übermittelten Daten
- **Authentifikationsservice** zur Sicherstellung der Identität des Kommunikationspartners
- **Verschwiegenheitsservice** zum Verhindern unautorisierter Offenlegung von Informationen
- **Zugriffskontrollservice** zum Verhindern unautorisierter Operationen auf Informationen und Prozessen des Systems

Der EFC-Betreiber kann entscheiden, welche Sicherheitsmaßnahmen notwendig sind und wann er sie in sein System integriert.

Im Folgenden wird der allgemeine Ausdruck *Road Side Unit* (RSU) als Überbegriff für die Mautbrücke und andere statischen Einheiten an der Straße genutzt. Das OBE wird durch eine allgemeine *Vehicle*-Komponente repräsentiert.

Teil II

VORSTELLUNG DER GRUNDLAGEN UND ANALYSE THEMENVERWANDTER ARBEITEN



2

TECHNIKEN UND EINSATZGEBIETE FÜR MONITORE

Die in dieser Arbeit aus den Monitor-Petrinetzen zu generierenden Monitore sollen auf verschiedenen Zielplattformen, die in verschiedenen Programmier- bzw. Beschreibungssprachen programmiert werden, eingesetzt werden. Hierbei kommen als Programmiersprachen Java, C und VHDL als Hardwarebeschreibungssprache zum Einsatz. Als Zielplattform wird der PC, ein Mikrocontroller und programmierbare Hardware als Ziel der Monitore genutzt. Des Weiteren wird der Prozess in den internationalen Standard der Automobilindustrie für eine gemeinsame Softwarearchitektur AUTOSAR integriert. Zum besseren Verständnis wird in den folgenden Abschnitten genauer auf den AUTOSAR-Standard und die programmierbare Hardware am Beispiel eines FPGAs eingegangen.

2.1 AUTOSAR

In einem Fahrzeug existiert eine Vielzahl an Steuergeräten (*engl. Electronic Control Units*) (ECU), auf denen Anwendungen laufen. Diese übernehmen unkritische Aufgaben wie Multimediafunktionen bis zu sicherheitsrelevanten Aufgaben wie die Motorsteuerung. Diese wurden in der Vergangenheit größtenteils zielplattformspezifisch entwickelt und mussten bei Änderung der Zielhardware (ECU) aufwendig angepasst werden.

AUTomotive Open System ARchitecture (AUTOSAR) [KF09] ist ein internationaler Standard der Automobilindustrie. Er definiert eine Softwarearchitektur, die Abstraktionsebenen vorgibt und hierdurch die Erstellung plattformunabhängiger *Softwarekomponenten* (*engl. software components*) (SW-Cs) ermöglicht. Des Weiteren wird eine Methodik geboten, die Austauschformate festlegt, die als Eingabe für den Konfigurationsprozess der Basissoftware dienen und die Integration von Anwendungen auf Steuergeräten beschreibt. Es werden Anwendungsschnittstellen für häufig benötigte Komponenten definiert, um das Zusammenspiel zwischen und die Austauschbarkeit von Anwendungen zu unterstützen.

Abbildung 2.1 zeigt die generelle AUTOSAR-Schichtenarchitektur als Steuergerätesicht (ECU-Sicht). Diese besteht aus mehreren Abstraktionsebenen, die aufeinander aufbauen und die die eigentliche Hardware von der Anwendungssoftware trennen. Auf der untersten Ebene befindet sich die eigentliche ECU-Hardware. Von dieser abstrahiert die *Basissoftware* und bietet definierte Schnittstellen an das *Runtime Environment* (RTE) an. Die Basissoftware besteht aus verschiedenen Komponenten, die untereinander ebenfalls über standardisierten Schnittstellen kommunizieren. Hierdurch sind die Komponenten je nach verwendeter Hardware austauschbar.

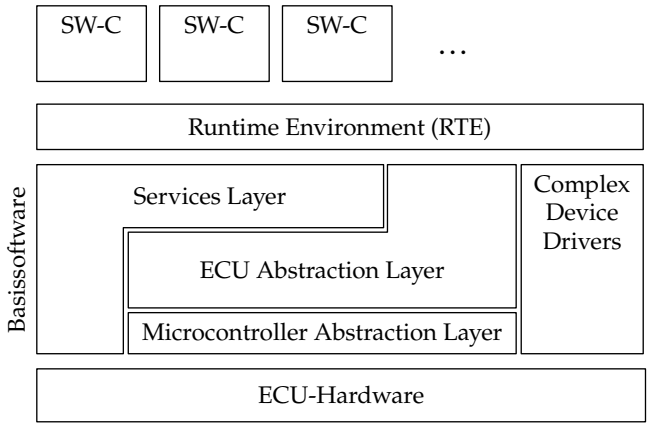


Abbildung 2.1: ECU-Sicht der AUTOSAR-Methodik (basierend auf [KF09])

Der *Microcontroller Abstraction Layer* ist mikrocontrollerspezifisch und abstrahiert von dem auf dem Steuergerät eingesetzten Mikrocontroller. Dieser umfasst die Initialisierung und die Konfiguration der Hardware sowie Treiber für die einzelnen Komponenten.

Darauf aufbauend bietet der *ECU Abstraction Layer* eine Abstraktion der Treiberschnittstellen und der eigentlichen Kommunikationsschnittstellen als standardisierte Schnittstellen an die darüberliegende RTE und den Services Layer. Der *Services Layer* bietet der RTE grundlegende Dienste, wie Betriebssystem, Speicherverwaltung und abstrakte Kommunikationsschnittstellen.

Zusätzlich zu dieser abstrahierenden Architektur existieren die *Complex Device Drivers*. Diese bieten die Möglichkeit für zeitkritische Aufgaben die einzelnen Abstraktionsschichten der Basissoftware zu umgehen und direkt aus der RTE auf die Hardware zuzugreifen. Durch den Schritt der Konfiguration der Basissoftware in der Methodik des AUTOSAR-Prozesses ist ein Austausch der ECU-Hardware möglich.

Das Runtime Environment (RTE) bindet die von der Basissoftware angebotenen Schnittstellen ein und bietet auf die Anwendungssoftware angepasste Schnittstellen zur Kommunikation der SW-Cs untereinander und mit anderen Steuergeräten.

Die Modellierung des Systems (Systemspezifikation) findet auf der Ebene des *Virtuellen Funktionsbusses* (*engl. virtual function bus*) (VFB) statt. Diese Ansicht ist schematisch auf der linken Seite der Abbildung 2.2 dargestellt. Auf dieser Ebene werden die Kommunikationsverbindungen zwischen den SW-Cs auf Modellierungsebene beschreiben. Hierbei dient eine Softwarekomponente (SW-C) als Modellierungselement des VFB. Diese SW-Cs sind Strukturelemente und besitzen *Ports*. Ports sind dabei Interaktionspunkte von SW-Cs, die ein Port-Interface zugewiesen bekommen. Es wird zwischen anbietenden Ports (*engl. provide ports*) und verlangenden Ports (*engl. require ports*) unterschieden. Als Port-Interfaces stehen drei Typen zur Verfügung. Das *Client/Server-Interface* dient dazu, dass ein Client bei einem Server Operationen ausführen kann. Dagegen bietet das *Sender/Receiver-Interface* einen asynchronen Austausch komplexer Daten zwischen SW-Cs in

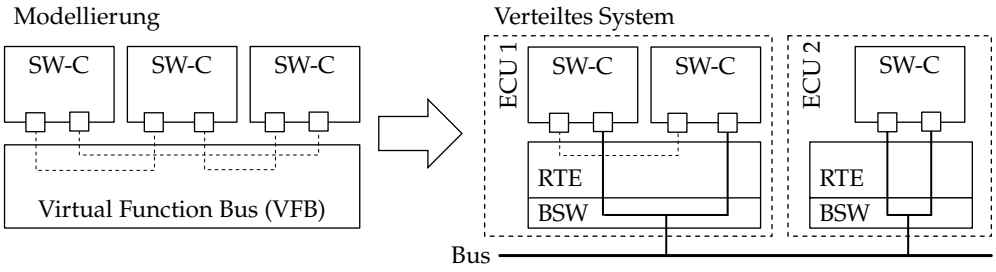


Abbildung 2.2: Modellierung mit dem VFB und Abbildung auf ein verteiltes System (basierend auf [KF09])

der Konfiguration 1-zu-m oder n-zu-1 an. Das dritte ist das *Calibration-Interface*, das ausschließlich der Übermittlung von Kalibrierungswerten dient.

Nach der Modellierung der Systemspezifikation wird die Basissoftware konfiguriert und damit an die tatsächlich verwendete ECU angepasst. Das Runtime Environment (RTE) wird auf Basis der Systemspezifikation mit einem RTE-Generator erzeugt. Dieses realisiert die auf VFB-Ebene modellierten Kommunikationsverbindungen zwischen SW-Cs und zwischen SW-Cs und der Basissoftware. Wie in Abbildung 2.2 auf der rechten Seite dargestellt ist, kann die Kommunikation, bei Verteilung der SW-Cs auf verschiedene ECUs, auch über Busse stattfinden. Für die SW-Cs entsteht durch die Kommunikation über standardisierte Schnittstellen kein Unterschied zu der Kommunikation zwischen SW-Cs, die sich auf derselben ECU befinden.

AUTOSAR wird in dieser Arbeit als Zielplattform zur Evaluation der Monitorgenerierung in einem realen Umfeld genutzt. Durch seine restriktiven Anforderungen an Schnittstellen erlaubt es die Integration des MBSecMon-Prozesses die weitgehend automatische Generierung und Einbindung von Monitoren in das AUTOSAR-System unter Einbeziehung der AUTOSAR-Systemspezifikation.

2.2 Konfigurierbare und rekonfigurierbare Logik

In der Automobilindustrie beinhalten ECUs gewöhnlich Mikroprozessoren, die die Ausführung der Softwarekomponenten und damit die Steuerung der Komponenten übernehmen.

Mikroprozessoren sind insbesondere für kontrollflussdominierte Aufgaben geeignet und universell einsetzbar. Sie können Unregelmäßigkeiten im Kontrollfluss effizient verarbeiten. Jedoch werden alle Aufgaben sequenziell verarbeitet, was die Verarbeitungsgeschwindigkeit einschränkt.

Im Gegensatz dazu finden Digitale Signalprozessoren (DSPs) und Spezialprozessoren bei datendominierten Aufgaben Einsatz. Diese sind auf die Datenverarbeitung ausgelegt und können spezielle Funktionen schnell und parallel abarbeiten.

Zur kundenspezifischen Anpassung dieser Spezialprozessoren stehen verschiedene Stufen der Flexibilität zur Verfügung. Applikationsspezifische Integrierte Schaltungen (*engl. Application-Apecific Integrated Circuits*) (ASICs) sind vollständig

nach Kundenwunsch entwickelte und gefertigte Prozessoren. Sie erreichen eine hohe Integrationsdichte, jedoch sind der Entwurfsprozess und die Fertigung sehr langwierig und teuer. Eine solche Lösung ist nur bei hohen Stückzahlen sinnvoll.

Eine deutlich günstigere Lösung sind *Semi-Custom ASICs*. Bei diesen beschränkt sich die Umsetzung auf vorgegebene Standardzellen und ein einfacheres Verbinden dieser Zellen (Routing), bei niedrigerer Integrationsdichte. Hierzu zählen auch die Gatearrays, die vordefinierte Standardgatter auf dem Chip bieten, die noch nicht verbunden sind.

Definition 5 (Gatter). *Ein Gatter ist ein elektronisches Bauteil mit booleschen Ein- und Ausgängen, das logische Operatoren wie AND, OR oder NOT realisiert und kombiniert. Es bildet eine Menge von Eingangssignalen über eine boolesche Funktion auf die Ausgänge ab.*

Die Verbindung dieser Gatter und damit die Programmierung der Funktion geschieht erst in einem zweiten Herstellungsschritt nach Kundenvorgaben. Diese Schaltungen sind somit programmierbar und bieten eine relativ hohe Integrationsdichte.

Bei *Field Programmable Logic* (FPL) kann der Kunde nicht nur selbst die Verbindung zwischen Gattern bestimmen, sondern diese auch selbst durchführen. Hier wird zwischen einfach und mehrfach programmierbaren FPLs unterschieden. Bei einfach programmierbaren FPLs werden bei der Programmierung Leitungen getrennt, um die gewünschte Funktion zu realisieren. Dieser Prozess wird auch „brennen“ genannt. Ein Nachteil hierbei ist, dass bei einem Designfehler im Entwicklungsprozess dieser auf dem FPL nicht mehr korrigiert werden kann. Mehrfach programmierbare FPLs, zu denen auch das Field Programmable Gate Array zählt, werden durch EPROM oder SRAM realisiert.

Ein *Field Programmable Gate Array* (FPGA) ist ein integrierter Schaltkreis, der logische Schaltungen implementieren kann. Er ist mehrfach programmierbar und ermöglicht somit die Integration logischer Schaltungen in einer schnellen und iterativen Art. Ein FPGA unterstützt die parallele Verarbeitung von Daten, während Mikroprozessoren komplexe Programme sequenziell abarbeiten.

Abbildung 2.3 zeigt eine abstrahierte Darstellung der grundlegenden Struktur eines FPGAs. Die programmierbaren Logikblöcke können über schnelle Signalpfade (*engl. single lines*) mit ihren Nachbarblöcken verbunden werden. Diese können über die Schaltmatrix beliebig konfiguriert werden. Um weiter voneinander entfernte Basisblöcke untereinander effizient zu verbinden, steht ein Gitter aus Busstrukturen zur Verfügung, die ebenfalls konfiguriert werden können.

Ein Logikblock besteht hauptsächlich aus Lookup-Tabellen (LUTs) und 1-Bit-Registern (D-Flip-Flops), die gewöhnlich im Verhältnis 1 zu 1 in einem FPGA vorkommen. LUTs können durch die Programmierung ihrer Wahrheitstabelle, die gewöhnlich als flüchtiger Speicherbaustein (SRAM) realisiert ist, beliebige Binärfunktionen abbilden. Die LUTs besitzen gewöhnlich zwischen 4 und 6 einzelne Bit-Eingänge und einen Ausgang, der den Wert 0 oder 1 annehmen kann. Das D-Flip-Flop dient zum Puffern des Ergebnisses bis zum nächsten Takt. Multiple-

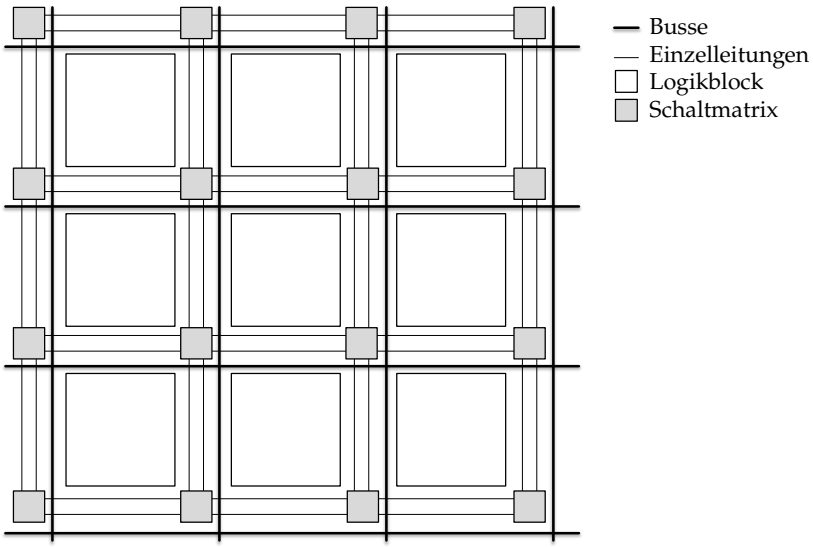


Abbildung 2.3: Grundlegende Architektur eines FPGAs

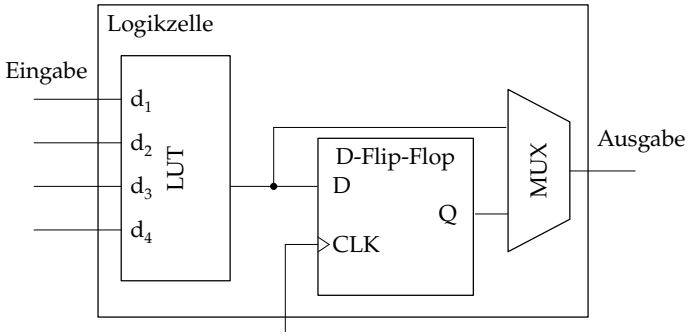


Abbildung 2.4: Einfachste Form einer Logikzelle

xer in den Basisblöcken erlauben es entweder getaktete Logik (mit Flip-Flop) oder kombinatorische Logik, in der das Flip-Flop umgangen wird, umzusetzen.

Die in Abbildung 2.4 gezeigte einfachste Form einer Logikzelle besteht aus einem LUT, einem D-Flip-Flop und einem konfigurierbaren Multiplexer um das D-Flip-Flop zu umgehen. Diese einfache Form der Logikzelle ist zwar ausreichend für die Integration beliebiger Schaltungen, jedoch für häufig verwendete Funktionen nicht effizient genug. Deshalb besitzen moderne FPGAs in ihren Basisblöcken komplexere konfigurierbare Logikschaltungen sowie dedizierte Schaltungen, wie effizientere Multiplizierer, für häufig benötigte Aufgaben.

Die einzelnen Logikzellen werden auf FPGAs zu Logikblöcken kombiniert. Diese bieten schnelle konfigurierbare Signalpfade zwischen den Elementen der Logikblöcke zur Rückkopplung von Signalen.

Hardwarebeschreibungssprachen wie die VHDL (Very High Speed Integrated Circuit Hardware Description Language) [IEE09] oder Verilog [IEE06] werden zur Beschreibung der Funktionsweise (logischen Funktionen), die auf dem FPGA abgebildet werden sollen, verwendet. Die Hardwarebeschreibungssprachen erlau-

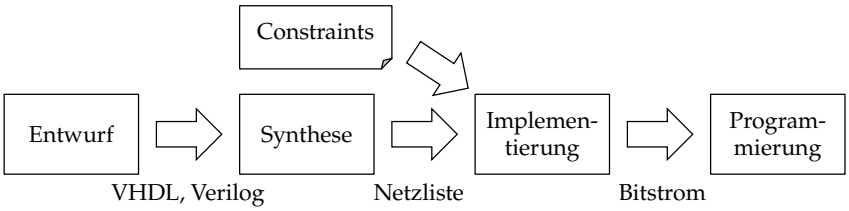


Abbildung 2.5: Ablauf des Entwurfs beginnend mit dem Register Transfer Level

ben die Spezifikation auf Ebene der funktionalen Verhaltensbeschreibung bis zur Ebene des Register Transfer Levels (RTL).

Definition 6 (Register Transfer Level). *Das Register Transfer Level beschreibt den Signalfluss zwischen Hardware-Registern und logische Operationen auf den Signalen.*

Diese Spezifikationen werden erst synthetisiert (Synthese) und dann auf die FPGA-Struktur abgebildet (Implementierung). Abbildung 2.5 zeigt die grundlegenden Schritte für die Übertragung der Hardwarebeschreibung auf eine Zielarchitektur (FPGA)

Im Schritt *Synthese* findet zunächst die Syntaxüberprüfung statt, in der die Eingabebeschreibung auf Syntaxfehler überprüft wird. Darauf folgend findet eine Übersetzung auf die RTL-Ebene statt. Abschließend wird ein *Technologiemapping* durchgeführt, das die logischen Ausdrücke (RTL) auf die Gatterebene abbildet. Hierbei spielt die Bibliothek des übersetzenden Werkzeugs eine große Rolle für die spätere Umsetzung. Die Ausgabe dieses Schrittes ist eine Netzliste.

Der zweite Schritt ist die *Implementierung*. Dieser setzt sich aus den Schritten, *Übersetzung*, *Mapping* und *Place-and-Route* zusammen. Bei der Übersetzung werden die übergebenen Netzlisten zusammen mit den Einschränkungen (*engl. constraints*), die zur Reduzierung der Komplexität dieses Schrittes dienen, in eine Gesamtnetzliste überführt. Anschließend wird im Mapping diese Gesamtnetzliste, die Elemente auf Gatterebene besitzt, auf Komponententypen der Zielarchitektur abgebildet. Abschließend werden im Place-and-Route-Schritt die Komponenten auf konkrete Komponenten der Zielarchitektur abgebildet. Die generierte Bitfile wird abschließend zur Programmierung des FPGAs eingesetzt. Der gerade beschriebene zweite Schritt ist ein NP-hartes Problem und wird durch Angabe von Einschränkungen des Entscheidungsraums durch die Constraint-Datei in vertretbarer Zeit berechenbar.

In dieser Arbeit wird VHDL als Zielsprache für die Monitorgenerierung aus der Monitor-Petrinetzsprache eingesetzt und die generierten Monitore wurden auf einem Xilinx Spartan3E Board[1] als Zielplattform zur Evaluation ausgeführt. Auf diesem FPGA nennt sich die Kombination aus vier Logikzellen (*Slice*) mit komplexerer Verschaltung *Configurable Logic Block* (CLB). Das Spartan 3E FPGAs in der Version XC3S500E besitzt 1.164 CLBs und somit 4,656 Slices.

[1] Xilinx Spartan3E Datenblatt: http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf

3

MODELLIERUNGSSPRACHEN FÜR ÜBERWACHUNGSSYSTEME

In diesem Kapitel werden die zum Verständnis dieser Arbeit benötigten Grundlagen des Themas Modellierungssprachen erläutert. Die *Unified Modeling Language* (UML) [RQJZ07] wird hierbei als bekannt vorausgesetzt.

Zunächst wird in Abschnitt 3.1 die *Model-based Security/Safety Monitor Specification Language* (MBSecMonSL) [Pat14] mit den erweiterten Live Sequence Charts und der Strukturierungssprache MUC, die als Spezifikationssprache des MBSecMon-Prozesses dienen, eingeführt. Die MBSecMonSL ist eine komplexe ausdrucksstarke Spezifikationssprache für Monitore, die alle an eine Signatursprache gestellten Anforderungen [Mei04] erfüllt. Somit dient sie als Referenzspezifikationssprache für die in dieser Arbeit entwickelte formale Zwischensprache Monitor-Petrinetze. Diese Zwischensprache muss auch Signaturen, die in einer solchen komplexen Modellierungssprache spezifiziert sind, effizient repräsentieren können.

Anschließend wird in Abschnitt 3.2 allgemein auf die Eigenschaften von Zustandstransitionssystemen, die in verwandten Ansätzen Verwendung finden, und ihre Eignung für die Abbildung von parallelem, verschränktem Kommunikationsverhalten eingegangen. Hierbei werden flache endliche Automaten, Zustandsautomaten der UML2 und Petrinetze betrachtet. Petrinetze mit ihrem ausgeprägten Parallelitätskonzept bilden im folgenden die Grundlage der in dieser Arbeit entwickelten Monitor-Petrinetze.

In dieser Arbeit wird der modellgetriebene MBSecMon-Entwicklungsprozess für Laufzeitmonitore vorgestellt. In diesem Prozess werden verschiedene Modelle spezifiziert und ineinander überführt. Abschnitt 3.3 stellt die *modellgetriebene Softwareentwicklung* vor, an die der MBSecMon-Entwicklungsprozess angelehnt ist. Hierbei werden die Variante der modellgetriebenen Entwicklung der OMG – Model Driven Architecture – und die darin eingesetzte Metamodellierung vorgestellt.

In verwandten Arbeiten und einer Variante der aus Monitor-Petrinetzen generierten Monitore wird die *Aspektorientierte Modellierung* eingesetzt. Dieses Programmierparadigma für die objektorientierte Programmierung wird am Beispiel der Implementierung AspectJ für Java in Abschnitt 3.4 vorgestellt.

Abschließend werden in Abschnitt 3.5 Anforderungen abgeleitet, die eine Zwischensprache für Monitorbeschreibungen im MBSecMon-Prozess erfüllen muss.

3.1 DIE MBSECMON-SPEZIFIKATIONSSPRACHE

Zur Modellierung von Signaturen in Form von Szenarien, die zur Überwachung von Systemen dienen, wurde die *Model-Based Safety/Security Monitor Specification Language* (MBSecMonSL) entwickelt, die auf den Live Sequence Charts und den Misusecases basiert. Die MBSecMonSL ist in [Pat14] beschrieben und dient als komplexe Spezifikationssprache für Monitorsignaturen. In diesem Abschnitt wird auf die für diese Arbeit relevanten Aspekte der MBSecMonSL eingegangen.

3.1.1 *Erweiterte Live Sequence Charts*

Erweiterte Live Sequence Charts (eLSCs) basieren auf den Live Sequence Charts, die in [DH01, HM03] eingeführt wurden. Diese dienen dem Zweck der Inter-Objekt-Modellierung von Szenarien. Im Gegensatz hierzu dienen Automaten bzw. Statecharts zur Modellierung von Intra-Objekt-Verhalten und beschreiben somit den internen Ablauf von Systemkomponenten.

Es existieren neben den LSCs viele Varianten der Message Sequence Charts (MSCs). Unter anderem sind dies die Variante der ITU [IT00], die angepasste Variante der UML2 und durch Zeitbedingungen und Coregionen erweiterte Versionen, sowie die High-Level MSCs, die eine übersichtliche Darstellung der Beziehungen zwischen MSCs erlauben. Alle diese Varianten bieten als Anforderungssprache nur eine sehr eingeschränkte Ausdrucksstärke. [HM03]

LSCs sollen diese Ausdrucksstärke durch neue Elemente und Modalitäten, die mögliche (*engl. cold*) und notwendige (*engl. hot*) Elemente unterscheiden, erhöhen. In LSCs werden Universal- und Existential-Chart unterschieden. Ein Universal-Chart kann eine Vorbedingung (Prechart) besitzen und beschreibt einen Ablauf, der auftreten muss, falls die Vorbedingung erfüllt ist. Ein Existential-Chart beschreibt einen Ablauf, der zu irgendeinem Zeitpunkt im System vorkommen kann.

Abbildung 3.1 stellt einen Usecase (positive Signatur) als Universal eLSC dar, das die wichtigsten in dieser Arbeit verwendeten Elemente enthält. Die beiden Lebenslinien symbolisieren die kommunizierenden Instanzen Road Side Unit (*RSU*), die eine Mautbrücke darstellen kann, und ein Fahrzeug (*Vehicle*).

Die Vorbedingung, die in einem *Prechart* modelliert ist, erwartet einen asynchronen Nachrichtenaustausch zur Initialisierung der Kommunikation zwischen RSU und Fahrzeug. Anschließend an diese Kommunikation wird das über die *InitializationResponse*-Nachricht als Parameter übermittelte Protokoll überprüft, ob für die weitere Kommunikation das CARDME-Protokoll verwendet wird. Ist dies der Fall, gilt im Folgenden die im Mainchart modellierte Signatur. Ansonsten wird die Überwachung des eLSCs abgebrochen (terminiert).

Im Mainchart wird zunächst ein notwendiger durch zwei heiße asynchrone Nachrichten modellierter Nachrichtenaustausch der Präsentationsphase erwartet. Darauf folgend ist eine Zuweisung mit dem Ausdruck `vehicleclass = pr.vc` modelliert, der den mit der *PresentationResponse*-Nachricht übermittelten Parameterwert in einer Variable für die spätere Verwendung speichert.

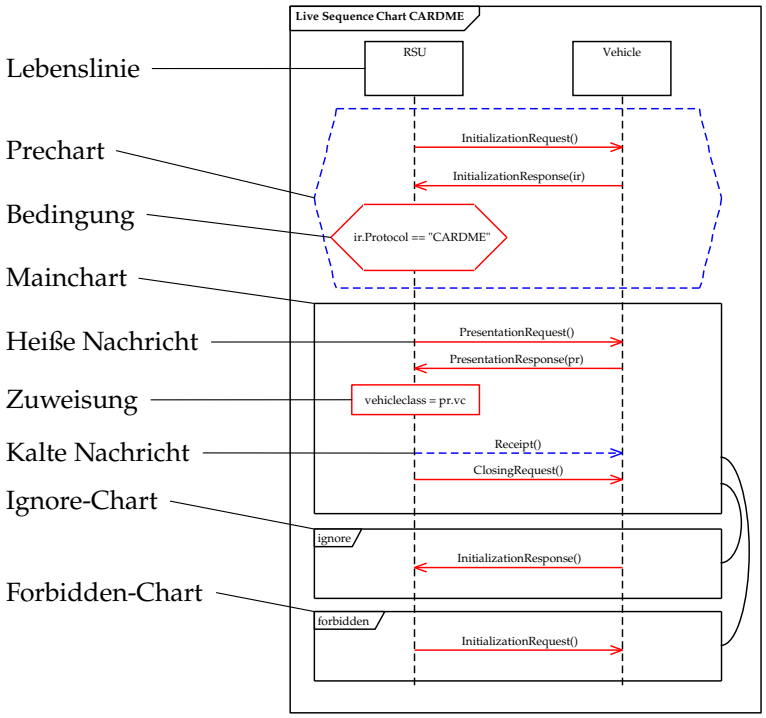


Abbildung 3.1: Das CARDME-Protokoll als Universal-Chart der eLSCs

Die RSU kann dann eine Quittung (*Receipt*) an das Fahrzeug schicken oder gleich mit einem *ClosingRequest* die Kommunikation beenden. Die *Receipt*-Nachricht ist als kalte Nachricht modelliert und hierdurch optional.

Während der Überwachung des Maincharts sind die beiden zusätzlichen Ignore- und Forbidden-Charts aktiv. Das Ignore-Chart bewirkt, dass die *InitialisationResponse*-Nachricht im Hauptteil der Signatur ignoriert wird. Die Implementierung der RSU sollte diese Nachrichten zu diesem Zeitpunkt der Kommunikation ebenfalls ignorieren. Falls jedoch eine Antwort der RSU auf diese Nachricht gesendet wird, ist dies ein Fehler und die Signatur ist fehlgeschlagen (failed).

Tabelle 3.1 zeigt eine Auflistung der Modellierungselemente der durch Harel eingeführten LSCs mit Beschreibung ihrer Bedeutung. Die erweiterten LSCs führen weitere Modellierungselemente (Tabelle 3.2) ein und passen die Semantik der LSCs an, sodass sie für die Überwachung einsetzbar sind. Diese semantischen Anpassungen können in [Pat14] genauer nachgelesen werden. In den in dieser Arbeit folgenden Beispielen, die in der eLSC-Sprache modelliert sind, wird die Semantik direkt an den verwendeten Beispielen erläutert.

Zusätzlich werden eLSCs als Use- bzw. Misusecase eingeordnet. Hierdurch ist eine Unterscheidung durch die Einordnung der eLSCs und nicht wie bei den LSCs durch einen negativen Ausgang der Signatur gegeben. Negative Signaturen unterscheiden sich somit nicht nur durch eine zu „false“ auszuwertende Bedingung am Ende des LSCs, sondern sind direkt als solche markiert. Ein als Misusecase markiertes eLSCs gilt in Gegensatz zu einem Usecase bei erfolgreicher Überwachung als fehlgeschlagen – der Misusecase wurde erkannt.

Element	Beschreibung
Lebenslinien	Lebenslinien stellen Kommunikationspartner dar, zwischen denen Nachrichten verschickt werden.
Prechart	Ein Prechart markiert den Bereich, in dem die modellierte Signatur eine Vorbedingung ist.
Mainchart	Das Mainchart ist der Hauptteil der Signatur und muss erfüllt werden, wenn das Prechart vorher erfüllt wurde.
Subchart	Ein Subchart dient der Synchronisierung von Abläufen auf mehreren Lebenslinien und kapselt somit Teile der Signatur.
Schleife	Eine Schleife ist ein spezieller Subcharttyp. Sie erlaubt eine Multiplizität (z. B. 0..*, 3..5 oder 5) zu definieren, die angibt, wie oft der Inhalt der Schleife auftreten muss bzw. darf.
Nachrichten	Nachrichten werden entweder zwischen verschiedenen Lebenslinien oder auf der eigenen Lebenslinie modelliert. Bei Nachrichten wird wie in MSCs zwischen synchronen und asynchronen Nachrichten unterschieden. Während asynchrone Nachrichten ein Sendeereignis und ein darauf folgendes getrenntes Empfangsereignis ausdrücken, fallen bei einer synchronen Nachricht Sende- und Empfangsereignis zusammen.
Zuweisungen	Zuweisungen sind Ausdrücke, die es erlauben Berechnungen auf Variablen auszuführen und das Ergebnis auf einer Variablen zu speichern.
Bedingungen	Bedingungen enthalten logische Ausdrücke (Prädikate), die zu wahr oder falsch evaluieren.
Forbidden-Chart	Das Forbidden-Chart wird dem LSC selbst oder einem der Charts zugeordnet. Es beschreibt Muster, die während der Ausführung bzw. Überwachung des entsprechenden Bereichs der Signatur nicht auftreten dürfen.

Tabelle 3.1: Syntaktische Elemente der Live Sequence Charts

3.1.2 *MUC-Sprache*

Zur Strukturierung der Signaturen in der eLSC-Sprache bietet die MBSecMonSL eine auf Usecasediagrammen basierende Misusecase-Sprache (MUC-Sprache) an. Die MUC-Sprache unterscheidet zwischen Usecases (UC) und Misusecases (MUC). Während Usecases positive Szenarien beschreiben, die in der zu überwachenden Kommunikation auftreten dürfen, beschreiben Misusecases nicht erlaubte Abläufe bzw. Angriffe auf das System.

Abbildung 3.2 zeigt eine auch im weiteren Verlauf der Arbeit wieder aufgegriffene und dort detaillierter beschriebene Strukturierung von Signaturen in der MUC-Sprache. Diese Strukturierung besteht aus insgesamt fünf Anwendungsfällen (Cases) und fünf Beziehungen zwischen diesen. Die Anwendungsfälle werden durch eLSCs detaillierter beschrieben. Das im vorherigen Abschnitt vorgestellte eLSC (Abb. 3.1) ist in einer erweiterten Form eine Signatur im Usecase CARDME.

Der Usecase CARDME beschreibt das allgemeine Kommunikationsprotokoll des CARDME-Standards. Durch die «extend»-Beziehung und den UC *CARDME Parking* wird eine Alternative zu einem Teil der Signatur beschrieben, der sich beim Einsatz des Protokolls in einem Parksystem von der Signatur in dem UC *CARDME* unterscheidet. Die «include»-Beziehung ermöglicht die Auslagerung von Teilen der Signatur und damit auch die Wiederverwendung dieser Teile in anderen UCs. Hier ist ein generischer Teil des Protokolls der die Nachverfolgungsprozedur *Tracking* beschreibt modelliert. Während der Überwachung des

Element	Beschreibung
Referenzfragment	Das Referenzfragment ist wie in MSCs ein Subchart, das andere LSCs referenziert. Hierdurch können Teilsignaturen in ein LSC eingefügt und somit wiederverwendet werden.
Par-Fragment	Das Par-Fragment besitzt beliebig viele orthogonaler Regionen, die es erlauben parallel auftretende Ereignisabfolgen zu überwachen. Hierbei müssen erst alle in den Regionen spezifizierten Muster beendet sein, bevor das Subchart verlassen wird.
Ignore-Chart	Das Ignore-Chart ist ähnlich zum Forbidden-Chart definiert. Im Gegensatz zum Forbidden-Chart beschreibt es Nachrichten, die während der Überwachung ignoriert werden.

Tabelle 3.2: Zusätzliche syntaktische Elemente der erweiterten Live Sequence Charts

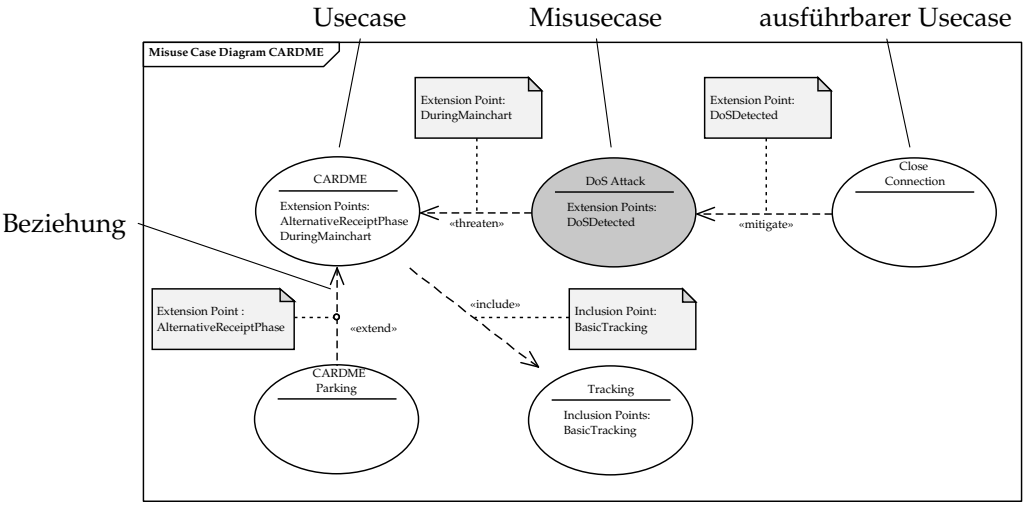


Abbildung 3.2: Beispiel für eine Misusecase-Strukturierung in der MBSecMon-Sprache

CARDME-Protokolls werden *InitializationRequest*-Nachrichten in der Hauptsignatur ignoriert. Der Misusecase *DoS Attack*, der mit dem UC über eine «threaten»-Beziehung verknüpft ist, überwacht die Frequenz dieser Nachrichten und meldet einen Angriff, wenn diese Frequenz eine Schwelle überschreitet. Falls dies geschieht, wird der UC *Close Connection*, der über eine «mitigate»-Beziehung mit dem MUC verbunden ist, ausgeführt.

Erweiterungspunkte (*engl. extension points*), die Elementen wie gesamten eLSCs oder den Charts der LSCs zugeordnet werden, erlauben über die Beziehungen Bereiche der Signaturen, in denen eine Teilsignatur überwacht werden soll, zu definieren. Die «threaten»-Beziehung ist mit dem Erweiterungspunkt mit dem Namen „DuringMainchart“ verknüpft, der an das Mainchart des eLSCs CARDME im UC CARDME zugeordnet ist.

Die in den LSCs notwendige Kategorisierung in Use- und Misusecases ist durch die Verwendung der MUC-Sprache nicht mehr notwendig, da die Zuordnung der Signaturen zu den Anwendungsfällen, dies schon festlegt.

3.2 Zustandstransitionssysteme

Zustandstransitionssysteme werden u. a. zur Beschreibung und Verifikation von zustandsbasierten Systemen in der Informatik eingesetzt. Sie bestehen aus einer Menge an Zuständen und Transitionen, die die Übergänge zwischen Zuständen beschreiben. Den Transitionen kann eine Beschriftung (*engl. label*) aus einer Menge an Beschriftungen zugeordnet sein, die auch an mehreren Transitionen vorkommen kann.

Weder die Anzahl an Zuständen, noch die Anzahl der Transitionen müssen hierbei beschränkt sein. Zustandstransitionssysteme können als gerichtete Graphen dargestellt werden. Zu dieser Kategorie gehören u. a. Endliche Automaten, UML2 State Machines und Petrinetze, die im Folgenden vorgestellt werden. Diese Formalismen besitzen einen unterschiedlichen Umfang an Modellierungskonzepten in Bezug auf die Modellierung von parallelem Systemverhalten.

3.2.1 *Endliche Automaten*

Zu der Kategorie der Zustandstransitionssysteme gehören unter anderem auch die Endlichen Automaten [Sam08]. Sie besitzen in Gegensatz zu Zustandstransitionssystemen spezielle Start- und Endplätze, die den Beginn bzw. das Ende eines Automaten repräsentieren. Ein flacher Endlicher Automat besteht grundlegend aus Zuständen, Transitionen und Aktionen.

Ein System besitzt verschiedene Zustände, wenn es sich in verschiedenen Situationen auf dieselben Eingaben unterschiedlich verhält. Funktionen, die unabhängig von vorherigen Eingaben immer dasselbe Ergebnis zurückliefern, sind dagegen nicht zustandsbehaftet. Endliche Automaten (*engl. finite state machines*) (FSMs) eignen sich, um zustandsbehaftete Systeme abzubilden. Ein Endlicher Automat kann nur eine endliche Anzahl an verschiedenen Zuständen einnehmen. Jederzeit ist immer genau ein Zustand aktiv. Wenn ein System in einem Zustand ist, reagiert es nur auf Teile der möglichen Eingaben, liefert vom Zustand abhängige Ausgaben zurück und wechselt durch Eingaben in neue Zustände.

Ein *Zustand* bildet in einer vom eigentlichen System abstrahierten Form die Folge der vorausgegangenen Ereignisse und damit relevanten Aspekte des Systemzustands ab.

> Ein Zustand modelliert eine Situation während der einige (normalerweise implizite) invariante Bedingungen gelten. Die Invariante kann eine statische Situation wie ein Objekt, das auf das Auftreten eines externen Ereignisses wartet, repräsentieren. Des Weiteren kann ein Zustand dynamische Bedingungen modellieren, wie dass der Prozess ein bestimmtes Verhalten ausführt. [OMG11]

Übergänge zwischen Zuständen werden durch Transitionen modelliert, die annotiert sein können. Im einfachsten Fall sind dies Bedingungen, die sich auf Systemvariablen beziehen. Sind diese erfüllt, findet ein Übergang von Quell- zum Zielzustand statt. Zusätzlich hierzu können Ereignisse an den Transitionen als Bedingung annotiert werden, die den Übergang zwischen Zuständen auslösen.

Wenn eine Eingabe verarbeitet wird, kann der Automat mit einer Aktion reagieren, die z. B. eine Variable inkrementiert.

Beispiel *Flacher endlicher Zustandsautomat eines Mautbrückenmoduls*

Abbildung 3.3 zeigt die Betriebszustände des Mautbrückenmoduls im Fahrzeug als Endlicher Automat. Der Automat startet im Startzustand und wechselt sobald das Modul gestartet wird den Zustand, der das Starten des Systems darstellt. Sobald das Modul hochgefahren ist (*started*), wechselt es in den Betriebszustand, in dem es senden kann (*send*). Wird es abgeschaltet (*ignitionOff*), wechselt der Automat wieder in seinen Startzustand.

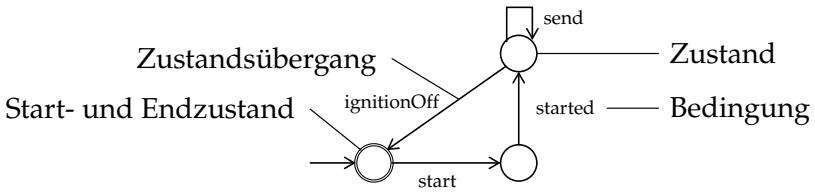


Abbildung 3.3: Endlicher Automat des Mautbrückenmoduls im Fahrzeug

Grundsätzlich existieren zwei von ihrer Ausdrucksstärke identische Ausprägungen der Endlichen Automaten. Bei Mealy-Automaten hängt die Ausgabe vom aktuell aktiven Zustand und der aktuellen Eingabe ab. Im Gegensatz hierzu hängt die Ausgabe des Moore-Automaten ausschließlich von den verlassenen und betretenen Zuständen ab. Die genutzte Transition ist für die Ausgabe unerheblich.

Parallele oder nebenläufige Aktivitäten eines Systems führen bei Abbildung auf einen flachen Endlichen Automaten zu einer exponentiellen Zunahme an zu modellierenden Zuständen.

3.2.2 *Zustandsautomaten der UML2*

Die Zustandsautomaten (*engl. State Machines*) der UML2 [OMG11] sind eine objektorientierte Variante der Statecharts von Harel [Har87]. Neben vielen neuen Modellierungselementen werden zusätzlich zu den flachen Endlichen Automaten in den UML2-Zustandsautomaten Konzepte zur Hierarchisierung (eingebettete Zustände) und zur Modellierung von Parallelität (orthogonale Regionen) eingeführt. Die Zustandsautomaten der UML2 vereinen die Eigenschaften von Mealy- und Moore-Automaten und bieten hierdurch Aktionen, die auf dem Zustand des Systems und der aktuellen Eingabe basieren, sowie Entry- und Exit-Aktionen der Zustände. Hierbei wird zwischen *Protocol* und *Behavioral State Machines* unterschieden, die für verschiedene Einsatzgebiete vorgesehen sind. Protocol State Machines beschreiben legale Einsatzszenarien für Protokolle und dienen der Überwachung der Einhaltung. Behavioral State Machines sind dagegen für die Modellierung des Verhaltens einer Instanz des Systems vorgesehen.

Annotationen an den Transitionen werden in den UML2-Zustandsautomaten folgendermaßen geschrieben: `Ereignis[Bedingung]/Aktion`

Das Konzept der orthogonalen Regionen erweitert die Endlichen Automaten (FSMs) um ein Konzept zur Modellierung paralleler Abläufe. Hierdurch wird die Übersichtlichkeit der Spezifikation erhöht und dem exponentiellen Zuwachs an Zuständen entgegengewirkt. Jedoch hat dieses Konzept auch Einschränkungen. Orthogonale Regionen müssen immer vollständig verlassen werden, da nur ein Zustand in UML2-Zustandsautomaten auf einmal aktiv sein kann.

Beispiel *UML2-Zustandsautomat der Kommunikationsmodule eines Fahrzeugs*

In Abbildung 3.4 ist ein kleines Beispiel eines UML2-Zustandsautomaten zu sehen. Dieser abstrahiert die Betriebszustände zweier Steuergeräte, die für die Kommunikation mit außerhalb des Fahrzeugs befindlichen Einheiten zuständig sind. Nachdem ein *WakeUp*-Signal an die beiden Steuergeräte gesendet wurde, starten diese nebenläufig zueinander. Dies ist durch einen Zustand mit den zueinander orthogonalen Regionen *TB_Module_Start* und *Enterprise_Module_Start* modelliert. Wird die Zündung während der Initialisierungsphase abgeschaltet (*[IgnitionOff]*), wird der Automat beendet. Haben hingegen beide Zustandsautomaten in den orthogonalen Regionen den Endzustand erreicht, wird der übergeordnete Initialisierungszustand verlassen und der Zustand *Ready* aktiv. In diesem Zustand kann beliebig gesendet werden (*send*). Wenn die Zündung abgeschaltet wird (*[IgnitionOff]*) beendet sich der Automat.

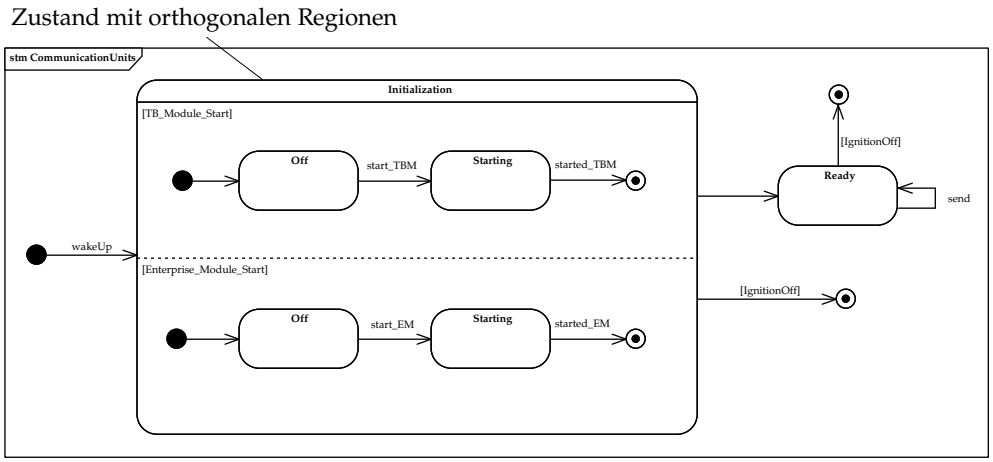


Abbildung 3.4: UML2-Zustandsautomat der Kommunikationsmodule im Fahrzeug

3.2.3 *Petrinetze*

Petrinetze wurden 1962 zum ersten Mal in der Dissertation von Carl Adam Petri, die am Fachbereich Mathematik und Physik der Technischen Universität Darmstadt eingereicht wurde, veröffentlicht und in [Pet62] vorgestellt. Das erste Einsatzgebiet der Petrinetze war die Beschreibung chemischer Prozesse. Die Erforschung und Erweiterung der Petrinetze fand größtenteils in Deutschland und Europa statt. Einen Überblick über die Entwicklung und die Einsatzgebiete von Petrinetzen bis 1988 gibt Murata in [MZ88].

Petrinetze sind ein grafisches und mathematisches Werkzeug, das durch seine Allgemeinheit in den verschiedensten Einsatzgebieten anwendbar ist. Es dient der Modellierung und der Analyse von Systemen, bei denen parallele oder nebenläufige Aktivitäten beschrieben werden müssen. Einsatzgebiete sind unter anderem die Modellierung und Analyse von Kommunikationsprotokollen, verteilten Systemen, parallelen oder nebenläufigen Programmen, programmierbarer Logik, aber auch formaler Sprachen.

Der größte Schwachpunkt der Petrinetze ist das Komplexitätsproblem, das bei der Analyse von Petrinetzen auftritt. Selbst für nicht sehr große Systeme tendieren Petrinetze zu groß für die Analyse zu werden. Dem kann durch Einschränkungen der Modellierungsfreiheiten, angepasst auf das zu lösende Problem, entgegengewirkt werden.

In Kapitel 5 werden die Place/Transition-Netze [DR98] als Grundlage für die formale Definition der Monitor-Petrinetze genutzt. Sie bieten eine grafische Notation für schrittweise Prozesse mit Entscheidung (Choice), Iteration und nebenläufiger Ausführung. Im Folgenden werden die Grundbegriffe der Petrinetze informell eingeführt.

Statische Elemente: Abbildung 3.5 zeigt ein einfaches Place/Transition-Netz. Es besteht grundlegend aus Plätzen, Transitionen und Kanten, die statische Elemente genannt werden.

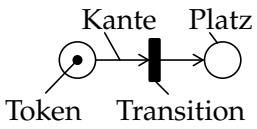


Abbildung 3.5: Grundlegendes Petrinetz

Definition 7 (Plätze). *Plätze werden durch Kreise oder Ellipsen dargestellt. Sie modellieren passive Komponenten und können Dinge enthalten und dadurch sich in einem Zustand befinden.*

Definition 8 (Transitionen). *Transitionen werden durch Quadrate oder Rechtecke symbolisiert. Sie modellieren aktive Komponenten und können Dinge erzeugen, verbrauchen und transportieren.*

Definition 9 (Kanten). *Plätze und Transitionen werden durch gerichtete Kanten, die grafisch als Pfeil dargestellt werden, verbunden. Sie modellieren abstrakte Beziehungen zwischen Komponenten.*

Die statischen Elemente werden zu einem Netz zusammengefügt und beschreiben somit Abläufe eines Systems.

Definition 10 (Netzstruktur). *Plätze werden durch gerichtete Kanten mit Transitionen verbunden und umgekehrt. Plätze bzw. Transitionen direkt miteinander zu verbinden ist nicht möglich. Auf eine passive folgt immer eine aktive Komponente.*

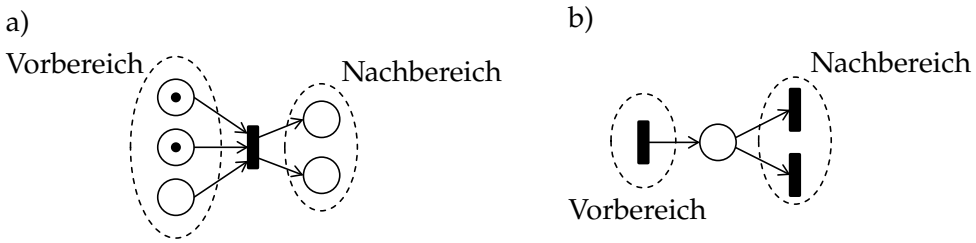


Abbildung 3.6: Vor- und Nachbereich einer Transition bzw. eines Platzes

Abbildung 3.6 a) zeigt anhand eines Teilnetzes den Vor- und den Nachbereich einer Transition, und Abbildung 3.6 b) den Vor- und Nachbereich eines Platzes. Hierbei befindet sich im Vor- und Nachbereich einer Transition eine Menge von Plätzen und in dem eines Platzes eine Menge von Transitionen. Der Vor- und Nachbereich einer Transition bzw. eines Platzes ist dabei folgendermaßen definiert:

Definition 11 (Vorbereich/Nachbereich). *Als Vorbereich einer Komponente versteht man alle Komponenten, die durch eine in die Komponente eingehende Kante mit dieser verbunden sind. Zum Nachbereich einer Komponente gehören entsprechend alle durch eine ausgehende Kante verbundenen Komponenten. Hierbei sind die Mengen nicht zwingend disjunkt.*

Dynamische Elemente: Neben diesen statischen Elementen besitzen Petrinetze auch dynamische Elemente. Diese dienen zur Darstellung des Zustandes des Netzes und beschreiben, wie sich dieser verändern kann.

In Abbildung 3.5 ist die Anfangsmarkierung durch ein Token im linken Platz als runder gefüllter Kreis dargestellt. Die hier betrachteten Petrinetze besitzen nur abstrakte Token, die keinen Typ besitzen. Allgemein können sich in Petrinetzen [Rei10] hingegen verschiedene Typen von Token in einem Netz befinden, die von Transitionen unterschiedlich behandelt werden.

Definition 12 (Token). *Token (Marken) sind dynamische Elemente eines Petrinetzes und werden durch Symbole in den Plätzen dargestellt. Sie repräsentieren Elemente der realen Welt und können entweder einen Typ besitzen oder auch abstrakt sein.*

Die sich in einem Netz befindenden Token bilden zusammen die Markierung, die den Zustand eines Netzes repräsentiert.

Definition 13 (Markierungen). *Markierungen sind eine Verteilung von Token auf die Plätze eines Netzes. Die Markierung repräsentiert dabei den aktuellen Zustand des durch das Petrinetz modellierten Systems.*

Für die Zustandsänderung des Netzes können Transitionen unter gewissen Bedingungen schalten.

Definition 14 (Schritte). *Eine Transition wird aktiviert, wenn alle Plätze in ihrem Vorbereich belegt sind. Eine aktivierte Transition schaltet, wenn das ihr zugewiesene Ereignis auftritt.*

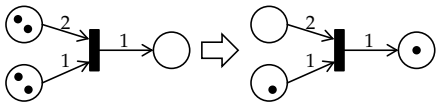


Abbildung 3.7: Kantenbeschriftung

In Abbildung 3.7 ist ein einfaches Petrinetz mit Kantenbeschriftungen zu sehen.

Definition 15 (Kantenbeschriftung). *Die Kantenbeschriftungen geben an, wie viele Token in den Plätzen des Vorbereichs der Transition liegen müssen, damit diese aktiviert wird. An ausgehenden Kanten zeigt sie an, wie viele Token beim Schalten der Transition in den nachfolgenden Plätzen erzeugt werden.*

Das Petrinetz kann in diesem Fall einmal schalten, da genügend Token auf den Plätzen im Vorbereich der Transition liegen. Aus dem oberen Platz im Vorbereich werden zwei Token und aus dem unteren wird ein Token entfernt. Im Platz des Nachbereichs der Transition wird ein Token erzeugt. Sind keine Kantenbeschriftungen angegeben, wird beim Schalten einer Transition immer nur ein Token pro Platz gelöscht bzw. erzeugt. Diese Kantenbeschriftungen können neben der Anzahl der Token auch den Typ der gelöschten bzw. der erzeugte Token definieren.

Zusätzlich zu den Kantenbeschriftungen kann durch das Festlegen von Platzkapazitäten eine Höchstgrenze für bestimmte Plätze spezifiziert werden.

Definition 16 (Kapazitäten von Plätzen). *Kapazitäten legen in Petrinetzen fest, wie viele Token in einem Platz gleichzeitig liegen können. Würde diese Grenze durch Schalten einer Transition verletzt werden, führt dies dazu, dass sie nicht aktiviert ist und somit die Transition nicht schalten kann.*

Abbildung 3.8 stellt ein Beispielnetz dar, in dem der untere Platz im Nachbereich der Transition eine Kapazität von 2 besitzt. Ohne die Kapazität könnte die

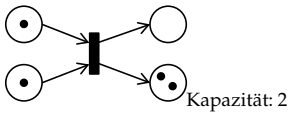


Abbildung 3.8: Kapazitäten von Plätzen in einem Petrinetz

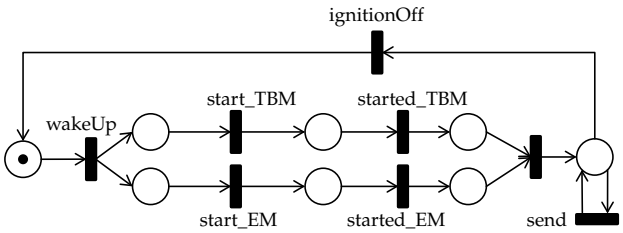


Abbildung 3.9: Petrinetz der Kommunikationsmodule im Fahrzeug

Transition, da die Plätze im Vorbereich belegt sind, schalten. Hier verhindert nun der Platz mit Kapazität, dass die Transition aktiviert wird, da auf ihm schon zwei Token liegen und damit beim Schalten die Kapazität verletzt würde.

Abbildung 3.9 zeigt das im vorherigen Abschnitt für einen UML2-Zustandsautomaten eingeführte Beispiel der Betriebszustände zweier Steuergeräte. Das während der Initialisierung mögliche Abschalten der Zündung wurde in diesen Fall nicht mit modelliert, da Transitionen für alle Kombinationen der Zustände während des Hochfahrens der Komponenten modelliert werden müssten. Dieses Abräumen des Netzes kann die Anzahl der zu modellierenden Transitionen deutlich erhöhen und die Übersicht verschlechtern. Die parallel ablaufenden Stränge sind in einem Petrinetz unabhängig und werden erst an Ende wieder synchronisiert. Hierdurch lassen sich auch beliebig verschachtelte Abläufe modellieren.

Nichtdeterminismus in Petrinetzen: Bei der Ausführung von PNs mit abstrakten Token und ohne Kantenbeschriftungen gilt Folgendes: Eine Transition kann schalten, wenn alle Plätze im Vorbereich der Transition belegt sind. Können in einem Schritt mehrere Transitionen schalten, wird eine der Transitionen nichtdeterministisch ausgewählt. Somit schaltet in jedem Schritt genau eine Transition. Abbildung 3.10 zeigt die einfachste Version eines solchen Konflikts (Verzweigungskonflikt). In diesem Beispiel sind beide Transitionen aktiviert und könnten schalten. Da beide Transitionen sich einen Platz in ihrem Vorbereich teilen, kann, wenn eine der beiden schaltet, die andere nicht mehr schalten. Eine zweite Art des Konflikts, der Wettbewerbskonflikt, kann in Petrinetzen, in denen die Plätze Kapazitäten besitzen, auftreten. Wenn ein gemeinsamer Platz im Nachbereich der Transitionen eine obere Grenze besitzt, kann es vorkommen, dass nur eine Transition der beiden schalten kann, da die Kapazität mit dem Schalten der anderen Transition überschritten würde. Diese Eigenschaften der Petrinetze führen zu einem möglichen nichtdeterministischen Verhalten. Dieser Nichtdeterminismus gilt ebenfalls für Petrinetze mit Kantenbeschriftungen und symbolischen Token. Die Free-Choice-Netze genannte Variante der Petrinetze schließt die Konstruktion von Netzen mit Konkurrenz aus.

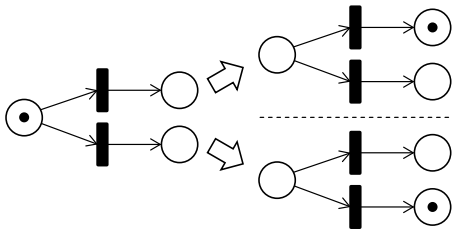


Abbildung 3.10: Verzweigungskonflikt in einem Petrinetz

Analysen in Petrinetzen: Im Gegensatz zu anderen Industriestandards wie den UML-Aktivitätsdiagrammen, der Business Process Modeling Notation (BPMN) und der Event-driven Process Chain (EPC) besitzen Petrinetze eine sehr stark entwickelte mathematische Theorie. Diese mathematische Basis erlaubt eine Vielzahl von Analysen. Viele dieser Analysen auf Petrinetzen basieren auf dem Konzept des Markierungsgraphen.

Definition 17 (Markierungsgraph). *Ein Markierungsgraph bildet die möglichen Zustände (Markierungen) eines Petrinetzes (PN) ab. Er wird aus der Anfangsmarkierung des PNs gewonnen, indem Transitionen, die aktiviert sind, geschaltet werden und die entstehende Markierung des PNs als Knoten in einem gerichteten Graphen gespeichert wird. Die schaltende Transition, die den Übergang von der vorherigen zur aktuellen Markierung auslöst, wird an den Kanten des Markierungsgraphen annotiert. Hierbei werden alle Pfade durch das Petrinetz genommen.*

Markierungsgraphen können nur für beschränkte Netze, d. h., es existiert eine obere Schranke für die Anzahl der Token im Netz, berechnet werden. Selbst bei diesen Netzen wächst die Anzahl der Markierungen im Markierungsgraphen exponentiell mit der Größe des Netzes (Plätze und Transitionen) an. Dieses Phänomen wird Zustandsraumexplosion genannt. Bei unbeschränkten Netzen wird die Technik des Überdeckungsgraphen angewendet, bei der jeder Zustand, unabhängig von der Anzahl der Token in den Plätzen, nur einmal im Graphen erfasst wird. Hierzu werden regelmäßige sich wiederholende endliche Teilgraphen zusammengeführt. Obwohl diese Technik nicht bei jeder Anwendung auf ein Petrinetz identische Graphen produziert, sind diese für die auf den Markierungsgraphen aufsetzenden Analysen gleichwertig. Diese Techniken erlauben die Analyse von Petrinetzen auf Eigenschaften wie Lebendigkeit, Terminierung und Beschränktheit.

3.3 Modellgetriebene Entwicklung

Der in dieser Arbeit beschriebene MBSecMon-Entwicklungsprozess orientiert sich an dem Konzept der modellgetriebenen Softwareentwicklung. Modellgetriebene Entwicklung (*engl. Model-Driven Engineering*) (MDE) bezeichnet einen Prozess, der es unterstützt aus formalen Spezifikationen (Modellen) lauffähige Software zu generieren. Ziel des MDEs ist es, eine durch Verwendung von Programmiersprachen

nicht immer erreichbare Abstraktion des zu modellierenden Problems zu erzielen. Hierzu werden domänenspezifische Sprachen (*engl. Domain Specific Languages*) (DSL) entwickelt, die eine passende Abstraktion des zu beschreibenden Problems darstellen. Aus den in diesen DSLs verfassten Beschreibungen wird automatisch entweder der lauffähige Code erzeugt oder das Modell interpretiert.

Durch den höheren Abstraktionsgrad der DSLs gegenüber von Code wird eine einfachere Verständlichkeit der Spezifikation erzielt. Die Entwicklung der DSL und die Bereitstellung einer Werkzeugkette für die Generierung des Codes führt jedoch zu einem erhöhten Initialaufwand gegenüber der klassischen Programmierung. Ziel ist es, durch Wiederverwendung der Modelle, eine vereinfachte Testfallerstellung und bessere Wartbarkeit der Modelle eine mindestens gleichwertige wenn nicht höhere Qualität des Softwareprodukts bei mindestens gleichbleibender Gesamtproduktivität zu erreichen.

3.3.1 *Model Driven Architecture*

Die Model Driven Architecture (MDA)[1] ist eine Umsetzung des MDE der OMG. Grundidee der MDA ist die Generierung von Software aus einem Modell, das in klar getrennte Schichten aufgeteilt ist. Das Modell wird in der MDA in folgende vier Schichten aufgeteilt, die von einem sehr abstrakten Modell bis zum Codemodell reichen.

COMPUTATION INDEPENDENT MODEL (CIM) Das CIM beschreibt Anforderungen an das System und ist unabhängig von der Realisierung dieser Anforderungen sowie von der Zielsprache und der Zielplattform des zu entwickelnden Systems. Als Sprache können hierfür in der UML z. B. Usecase-Diagramme oder Interaktionsdiagramme eingesetzt werden.

PLATFORM INDEPENDENT MODEL (PIM) Das PIM beschreibt das zu entwickelnde System, zeigt jedoch keine Details der Plattform auf dem das System eingesetzt wird.

PLATFORM SPECIFIC MODEL (PSM) Das PSM beschreibt dasselbe System wie das PIM. Es enthält zusätzlich Informationen, die festlegen, wie das System die Zielplattform nutzt.

CODEMODELL Das Codemodell ist eine Repräsentation des PSM in einer Zielsprache wie Java oder C, die auf der Zielplattform lauffähig ist.

Grundsätzlich wird in der MDA das auf der höheren Abstraktionsebene liegende Modell mit genaueren Informationen angereichert, um das darunterliegende weniger abstrakte Modell zu erhalten. Die Abbildung einer Schicht auf die darunterliegende Schicht soll möglichst durch eine automatische Transformation stattfinden und rückführbar sein. Hierbei werden Elemente des Modells auf Elemente des darunterliegenden Modells abgebildet. Der Übergang von PSM zu Code findet durch eine Codegenerierung statt. Eine Möglichkeit der direkten Codegenerierung aus einem PIM ohne die Verwendung eines PSM ist in der MDA ebenfalls vorgesehen.

[1] OMG MDA-Website: http://www.omg.org/mda/

Beispiel *Abstraktionsschichten der MDA am Beispiel des MBSecMon-Prozesses*

Abbildung 3.11 zeigt die vorgestellten Abstraktionsschichten am Beispiel des MBSecMon-Prozesses. Auf oberster Ebene (CIM) befindet sich eine Beschreibung der durch die Signaturen zu erkennenden Fehler und Angriffe. Diese werden z. B. durch umgangssprachliche Beschreibungen oder durch die MUC-Sprache mit einfachen, abstrahierten eLSCs beschrieben.

Auf der PIM-Ebene befinden sich zwei verschiedene Modelle, die auseinander hervorgehen. Das CIM wird soweit verfeinert, dass explizite Signaturen, die sich für die automatische Weiterverarbeitung eignen, in der MBSecMon-Sprache, vorliegen. Diese werden dann durch eine Transformation in die explizitere MPN-Repräsentation gebracht.

Auf Basis dieses MPN-Modells werden die Signaturen optimiert und Transformationen durchgeführt, die die Signaturen an die Zielplattform anpassen. Dies führt zu einem PSM. Zusätzlich kommen in diesem Schritt weitere plattformspezifische Informationen hinzu.

Als letzte Transformation werden die Signaturen mit den plattformspezifischen Informationen in einen Monitor in Java, C oder VHDL überführt. Diese Transformation ist eine Codegenerierung.

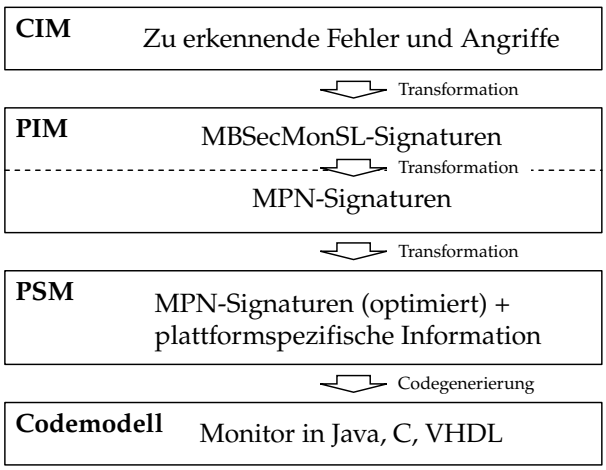


Abbildung 3.11: Abstraktionsschichten der MDA am Beispiel des MBSecMon-Prozesses

Die Sprachen MBSecMonSL und MPN, in der die Modelle der verschiedenen Schichten beschrieben werden, werden durch Metamodelle definiert.

3.3.2 *Metamodellierung*

Die Metamodellierung bildet die Grundlage der modellgetriebenen Softwareentwicklung. Sie ist die Basis für die Definition domänenspezifischer Sprachen, die Modelltransformation und die Werkzeugentwicklung.

Meta kommt aus dem Griechischen und beschreibt, dass sich etwas auf einer höheren als der aktuellen Ebene befindet. Metamodelle sind damit als Modelle von Modellen zu sehen. Die OMG [OMG06] hat hierzu eine Architektur entwi-

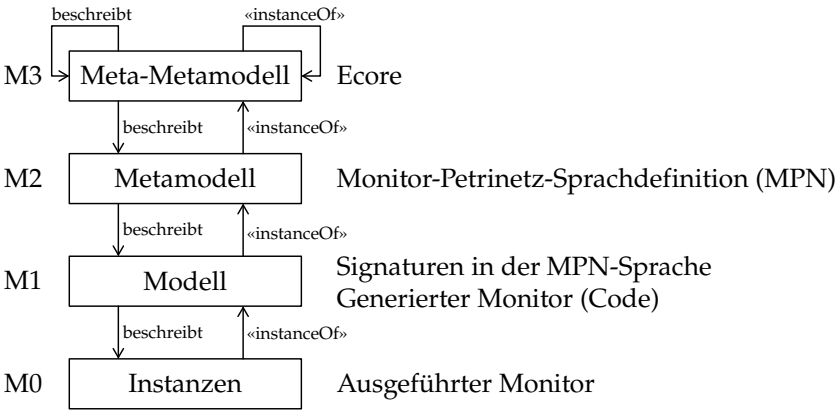


Abbildung 3.12: Metaschichten-Architektur der OMG

ckelt, die mindestens vier Ebenen umfasst, die mit M0 bis M3 gekennzeichnet sind.

Die Modellierung beschreibt eine Domäne mittels eines Modells. Für dieses Modell wird eine Modellierungssprache benötigt, die eine festgelegte Syntax und Semantik besitzt. Zur Beschreibung der Syntax und statischen Semantik dieser Modellierungssprache selbst wird wiederum eine Sprache benötigt, die Metamodellierungssprache. Diese Metamodellierungssprache besitzt nun wieder eine festgelegte Syntax und Semantik, die definiert werden muss. Zum Aufbrechen dieses Zyklus wird eine Meta-Metamodellierungssprache benötigt, die durch sich selbst beschrieben werden kann. Die OMG schlägt als Meta-Metamodellierungssprache die *Meta Object Facility* (MOF) vor.

Abbildung 3.12 zeigt am Beispiel der MPN-Sprache die Metaschichten-Architektur der OMG. Im Gegensatz zu der OMG-Spezifikation wird in diesem Beispiel wie auch in der Umsetzung der MPNs in dieser Arbeit *Ecore* [SBPM08], die die essenziellen Kernbeschreibungselemente der MOF enthält, als Metamodellierungssprache eingesetzt. Als Meta-Metamodell der M3-Ebene wird die Sprache Ecore verwendet. Mit dieser wird das Metamodell der MPN-Sprache (M2) beschrieben. Mit der MPN-Sprache können dann Signaturen (M1) modelliert werden, aus denen ausführbarer Code (M1) generiert wird. Die Instanzen der Monitore zur Laufzeit überwachen das System (M0).

Beispiel *MDA-Metaschichten am Beispiel der MPN-Sprache*

Abbildung 3.13 zeigt am konkreten Beispiel die Metaebenen M1 bis M3 der vereinfachten MPN-Sprache. Auf der M3-Ebene ist die Metamodellierungssprache Ecore zu finden. Dieser Kern der Ecore-Sprache besteht aus *EClass*, *EAttribute*, *EReference* und *EDataType*. Alle diese Klassen sind wiederum Instanzen der *EClass*. Die Klasse *EClass* repräsentiert eine Klasse von Instanzen. *EAttribute* stellt Namen-Wert-Paare dar, die mit einer *EClass* verknüpft sind. Diese Attribute haben einen Typ, der durch die Klasse *EDatatype* repräsentiert wird. Schließlich stellt *EReference* ein Ende einer Assoziation dar, die sich zwischen zwei Klassen befindet. Die *EReference* kann eine Besitztbeziehung (*engl. containment*) darstellen und zeigt auf eine Klasse. Des Weiteren wird ihr eine Multipli-

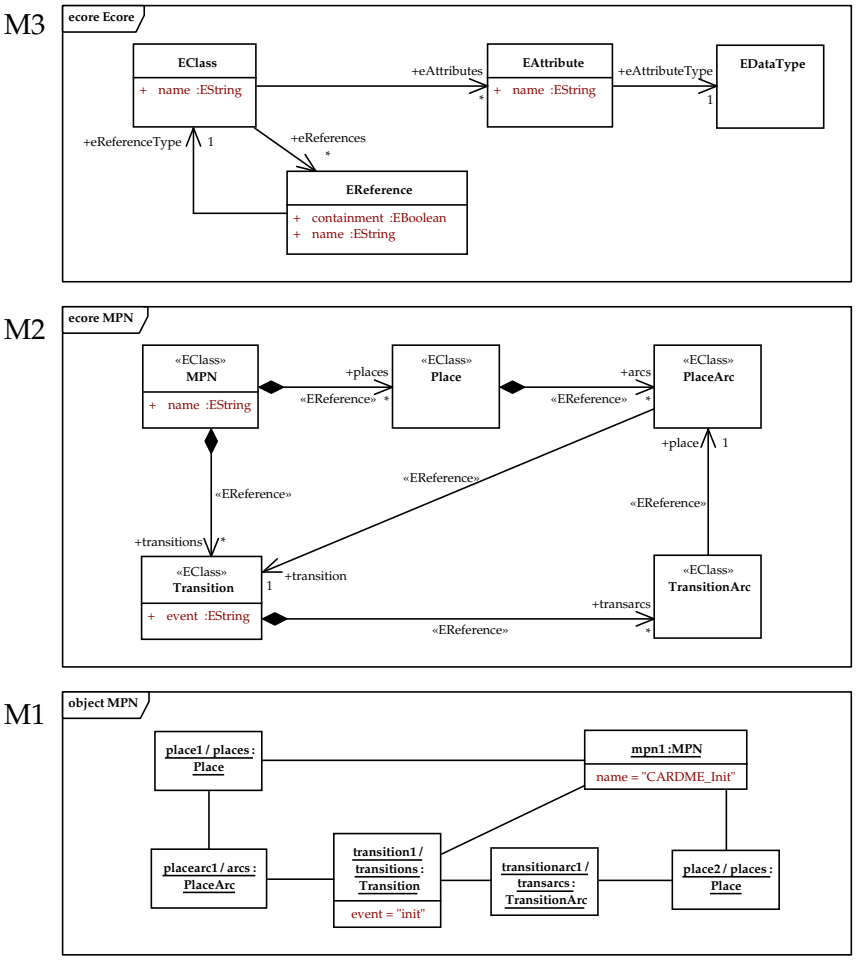


Abbildung 3.13: Metaschichten am konkreten Beispiel der MPN-Sprache

zität zugewiesen, die festlegt, wie viele Instanzen der referenzierten Klasse auf der darunterliegenden Ebene existieren dürfen. Die Instanzen der *EAttribut*s auf dieser Ebene, die in den Klassen *EClass*, *EReference* und *EAttribute* annotiert sind, sind vom Typ *EString* bzw. *EBoolean*.

Basierend auf dieser Metamodellierungssprache kann nun auf der Ebene M2 das Metamodell der zu definierenden Sprache erstellt werden. Hier ist eine vereinfachte Version des MPN-Metamodells gezeigt. Die Klassen wie *MPN*, *Place* und *Transition* sind Instanzen der Metaklasse *EClass*. Die Referenzen zwischen den Klassen basieren auf der Metaklasse *EReference* und die Attribute in den Klassen auf der Metaklasse *EAttribute*.

Ein MPN besitzt Transitionen und Plätze. Es existieren zwei Typen von Kanten: *PlaceArc*s, die von Plätzen besessen werden, und *TransitionArc*s, die zu den Transitionen gehören. Von einem Platz gehen *PlaceArc*s zu Transitionen und *TransitionArc*s zeigen auf Plätze. An Transitionen sind Ereignisse (*event*) annotiert und MPNs besitzen Namen. Beide sind vom Typ *EAttribute* und vom in Ecore als *EDataType* definierten Typ *EString*.

Auf der Ebene M1 ist nun ein einfaches Petrinetz mit zwei Plätzen und einer Transition in abstrakter Syntax zu sehen. Dieses Modell ist eine Instanz des

MPN-Metamodells auf Ebene M2. Der Platz *place1* ist über die Kante *placearc1* mit der Transition *transition1* verbunden. Die Verknüpfungen finden auf der M1-Ebene über Links statt, die Instanzen der Referenzen der darüber liegenden Ebene sind. Die Transition *transition1* ist über die Kante *transitionarc1* mit dem zweiten Platz *place2* verbunden. Alle Plätze und Transitionen sind über eine Besitztbeziehung (Link mit dem Attribut *containment* auf „true“) mit dem MPN verknüpft.

Bei der modellbasierten Entwicklung dient das Metamodell als Richtlinie wie ein Modell auszusehen hat. Des Weiteren kann aus dieser Spezifikation ein Modell-Repository generiert werden, in dem Modelle gespeichert werden können. Auf Basis dieser Modelle in den Repositories ist es dann möglich Modelltransformationen bzw. Codegenerierungen durchzuführen.

3.4 Aspektorientierte Programmierung

Aspektorientierte Programmierung (AOP) ist ein Programmparadigma für objektorientierte Sprachen wie zum Beispiel Java, C++ und C#. Ziele von AOP sind eine bessere Wartbarkeit und Wiederverwendung von Funktionalitäten durch stärkere Modularisierung der Implementierung. AOP wurde in [KLM+97] für die Realisierung einer besseren Aufteilung querschnittlicher Belange vorgeschlagen. Es dient der Auslagerung von Implementierungsteilen, die modulübergreifend eingehalten werden müssen (Cross-Cutting Concern). Ein häufiger Einsatzzweck ist hierbei die Umsetzung generischer Funktionen, wie zum Beispiel das Logging. Diese müssten ansonsten als Methodenaufrufe an vielen Stellen manuell in den Quelltext eingebracht werden und können mit dieser Technik nun in einen *Aspekt* ausgelagert werden. Zur Kompilierzeit werden diese in einem Aspekt gekapselten Funktionalitäten automatisch in den Quelltext eingewoben. Es ist das Ziel einen möglichst hohen Anteil des Overheads durch Vorberechnungen und angepasste Intrumentierungen schon zu diesem Zeitpunkt zu verursachen, um den späteren Laufzeitoverhead zu verringern.

Aspektorientierte Programmierung wird nicht direkt von jeder Sprache unterstützt. Auch die beispielhaft aufgeführten Programmiersprachen Java, C++ und C# unterstützen diese nicht von sich aus, weshalb AOP durch Einbindung eines Rahmenwerks hinzugefügt werden muss. Ein solches Rahmenwerk für Java ist zum Beispiel AspectJ [KHH+01].

Im Folgenden wird auf die Benennung der Konzepte in AspectJ zurückgegriffen, wobei diese sich in den Rahmenwerken für verschiedene Programmiersprachen unterscheiden können. Grundsätzlich existieren zwei Typen des Crosscuttings. *Static Crosscutting* beschreibt neue Operationen auf statischen Typen und *Dynamic Crosscutting* definiert zusätzlich auszuführende Operationen an klar definierten Punkten des Programms. Im Folgenden wird das in diese Arbeit eingesetzte dynamische Crosscutting betrachtet.

Zur Umsetzung der aspektorientierten Programmierung in Java führt AspectJ nur eine kleine Menge an neuen Konstrukten ein:

Join Points sind klar definierte Punkte im Ablauf des Programms.

POINTCUTS sind ein Mittel zur Referenzierung von Mengen von Join Points und bestimmten Werten an diesen Punkten.

ADVICES sind methodenähnliche Konstrukte zur Beschreibung von zusätzlichem Verhalten an Join Points, die durch zugeordnete Pointcuts referenziert werden. AspectJ unterstützt grundlegend die Advices `before`, `after` und `around`, die den Zeitpunkt angeben, wann das Advice ausgeführt werden soll. Für das `after`-Advice existieren zusätzlich noch die Spezialfälle `after returning` sowie `after throwing`.

ASPECTS sind Einheiten modularer querschnittlicher Impementierungseinheiten, die aus Join Points, Advices und normalem Javacode bestehen.

Beispiel *Aspekt zur Übergabe von Ereignissen an einen Monitor*

Im Folgenden ist ein kleines Beispielprogramm zu sehen, das aus je einer Methode in der Klasse RSU zum Senden und zum Empfang einer Nachricht besteht.

```
public class RSU{
 public void sendInitRequest(int reqno){
  ...
 }

 public void receiveInitResponse(int respno){
  ...
 }
 ...
}
```

Die Methoden haben jeweils einen Parameter `reqno` bzw. `respno` der beim Aufruf der Methoden gesetzt wird.

Jedes Mal wenn die Methode `sendInitRequest` aufgerufen wird, soll dem Monitor das Ereignis `initRequest` mit dem übergebenen Parameterwert zur Verarbeitung übergeben werden. Der folgende Aspekt in der Sprache AspectJ übernimmt diese Aufgabe.

```
public aspect RSUMonitor{
 pointcut monitorMethod(int reqno) : args(reqno) && calls(* *.RSU.
     sendInitRequest(int));

 //Advice
 before(int reqno) : monitorMethod(reqno) {
  monitor.dispatchEvent("initRequest", reqno);
 }
}
```

Hierbei wird ein `before`-Advice, der den Monitor vor der Ausführung der beschriebenen Methode aufruft, verwendet. Der Advice besitzt den Parameter `reqno`, der ihm vom Pointcut `monitorMethod` übergeben wird. Das Argument wird im Pointcut über das `args`-Schlüsselwort aus dem Join Point dem benutzerdefinierten Pointcut-Bezeichner `monitorMethod` übergeben. Der Pointcut selbst beschreibt den Aufruf (`calls`) der Methode `sendInitRequest` der Klasse

RSU mit einem Parameter des Typs `int`. Der Rückgabewert der Methode und die Paketstruktur der Klasse sind hierbei durch Wildcards (*) freigelassen.

Für objektorientierte Sprachen werden Aspekte häufig als Zielsprache bzw. Modellierungssprache für Monitore oder zur Integration von Monitoren in vorhandene Programme verwendet. Das *dynamische Crosscutting* dieses Programmierparadigmas wird u. a. in dieser Arbeit zur Einbindung von Monitoren in objektorientierte Sprachen wie Java verwendet. Somit können die Instrumentierungen, die der Einbindung der Monitore dienen, unabhängig vom zu überwachenden Programm erstellt und gewartet werden.

3.5 ANFORDERUNGEN AN EINE ZWISCHENSPRACHE FÜR DIE MONITORGENERIERUNG

Ziel dieser Arbeit ist die Definition einer universellen Zwischensprache zur Abbildung von Spezifikationssprachen für Monitore, wie die MBSecMonSL, auf plattformspezifischen Monitoringcode. Eine Sprache, die in einem Prozess als allgemeine Zwischenrepräsentation für verschiedene Spezifikationssprachen dienen soll, muss die in diesem Abschnitt vorgestellten Anforderungen erfüllen.

Es gibt zwei Kategorien von Anforderungen, die für die Zwischensprache relevant sind. Zum einen sind das direkte Anforderungen an die Ausdrucksstärke einer Signatursprache. Diese beschreiben, welche Mustertypen mit der Sprache modelliert werden können und gelten auch für die Spezifikationssprachen. Zum anderen sind es Anforderungen, die aus dem Einsatz als Zwischensprache in einem Entwicklungsprozess für Monitore mit anschließender Codegenerierung entstehen. Diese umfassen auch domänenspezifische Anforderungen, die sich aus der Laufzeitüberwachung an sich und den Zielplattformen auf denen der Monitor laufen soll ergeben.

Als erstes werden hier die Anforderungen der ersten Kategorie betrachtet. Sie sind wie die Anforderungen der *Event Description Language* (EDL) [Sch04] an den in [Mei04] veröffentlichten Anforderungen an eine Ereignisbeschreibungssprache angelehnt. Da die im Prozess vorgesehene Modellierungssprache für Signaturen, die erweiterten LSCs [PPS11], die folgenden Anforderungen erfüllen, müssen diese auch durch die Zwischensprache erfüllt werden. Folgende Anforderungen sind somit an die Semantik einer Signatursprache zu stellen:

AS1) SEQUENZTYPEN

SEQUENZ Mehrere Ereignisse mit einer festen sequenziellen Ordnung

KONJUNKTION Mehrere Ereignisse/Ereignisabfolgen, die in beliebiger Reihenfolge auftreten können. Dabei enden die Abfolgen mit einem gemeinsamen Ereignis.

NEGATION Ein Ereignis bzw. eine Ereignisabfolge, die nicht auftreten darf.

DISJUNKTION Ein Ereignis bzw. eine Ereignisabfolge von mehreren möglichen darf auftreten.

SIMULTAN Zwei Ereignisse, müssen gleichzeitig auftreten.

AS2) ITERATIONSTYPEN

EXAKT Ein Muster muss n-mal auftreten.

MINDESTENS Ein Muster muss mindestens n-mal auftreten.

HÖCHSTENS Ein Muster darf höchstens n-mal auftreten.

AS3) KONTINUITÄT

KONTINUIERLICH Jedes Ereignis, das auftritt, muss in der Signatur modelliert sein. Dabei muss die spezifizierte Reihenfolge eingehalten werden.

NICHT-KONTINUIERLICH Eine Signatur wird erkannt, wenn alle in ihr modellierten Ereignisse in der richtigen Reihenfolge aufgetreten sind. Ereignisse, die zusätzlich zu den modellierten auftreten, werden ignoriert.

AS4) NEBENLÄUFIGKEIT

NICHT-ÜBERLAPPEND Zwei oder mehr Ereignissequenzen müssen nacheinander auftreten. Sie dürfen keine Ereignisse während des Matchings teilen.

ÜBERLAPPEND Zwei oder mehr Ereignissequenzen dürfen sich während des Matchings Ereignisse teilen. Hierdurch werden Ereignissequenzen verschränkt überwacht.

AS5) KONTEXTBEDINGUNGEN

INTRA-SCHRITT-BEDINGUNG Die Bedingung besteht aus einem einfachen booleschen Ausdruck, der auf den Eigenschaften eines Ereignisses, zum Zeitpunkt des Auftretens dieses Ereignisses, basiert.

INTER-SCHRITT-BEDINGUNG Eine komplexe Bedingung zwischen Eigenschaften mehrerer Ereignisse.

AS6) MATCHING-REGELN

ERSTES Das erste auftretende Ereignis eines Ereignistyps wird gebunden.

LETZTES Das letzte Ereignis bei mehrfachem Auftreten eines Ereignistyps wird gebunden.

ALLE Jedes passende Auftreten eines Ereignistyps wird gebunden.

AS7) KONSUM

KONSUMIEREND In einer Signatur wird ein aufgetretenes Ereignis nur für einen Match im modellierten Ereignismuster verwendet.

NICHT-KONSUMIEREND In einer Signatur wird ein aufgetretenes Ereignis für alle möglichen Matches in einem Ereignismuster verwendet.

Diese Anforderungen an die Ausdrucksstärke einer Signatursprache müssen auch von der Zwischensprache erfüllt werden, um alle beliebigen Signatursprachen, die diese oder einen Teil dieser Sprachkonstrukte beinhalten, weiterverarbeiten zu können. In [Pat14] wird durch die Abbildung auf die hier entwickelte Zwischensprache (MPN) gezeigt, dass die Anforderungen an die Sprache erfüllt

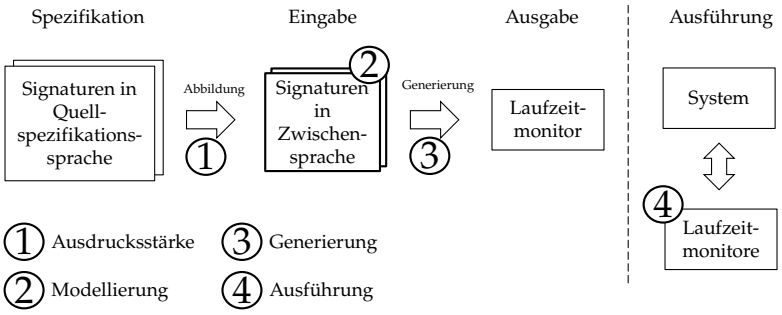


Abbildung 3.14: Quellen der Anforderungen

sind. Für die Anforderung AS1 (*Simultan*) gilt im Fall der MBSecMon-Sprache einschränkend, dass eine Zeitspanne definiert wird, in der die Ereignisse als gleichzeitig angesehen werden. AS1 *Konjunktion* und AS4 *Nebenläufigkeit* unterscheiden sich insoweit, dass eine Konjunktion zusätzlich eine Zusammenführung über ein Ereignis am Ende der simultanen Abfolge verlangt.

Die Anforderungen der zweiten Kategorie, die aus dem Einsatzgebiet als Zwischensprache in einem Monitorgenerierungsprozess entspringen, werden im Folgenden betrachtet. Hierbei spielen insbesondere der Monitorgenerierungsprozess, der Einsatz der Sprache zur Spezifikation von Laufzeitmonitoren sowie die Anforderungen der möglichen Zielplattformen und Programmiersprachen eine wichtige Rolle.

Tabelle 3.3 zeigt die Kategorisierung der vorgestellten Anforderungen an die Zwischensprache nach Ursprung im Prozess, der in Abbildung 3.14 dargestellt ist. Hierbei entstehen viele der Anforderungen aus mehreren Schritten des Prozesses der Monitorgenerierung. Der Schritt *Modellierung* steht hierbei für Eigenschaften der Signatur an sich in der Zwischensprache nach der Übersetzung aus einer Spezifikationssprache oder der direkten Modellierung. Die *Ausdrucksstärke* der Zwischensprache baut auf den Anforderungen aus der ersten Kategorie der Anforderungen auf und konkretisiert diese auf die Anwendung im Generierungsprozess für Monitore. Die Kategorie *Generierung* steht für die Übersetzung einer Spezifikation in der Zwischensprache auf eine Zielplattform und in eine Programmiersprache. Unter *Ausführung* sind Anforderungen zusammengefasst, die von der Ausführung auf dem Zielsystem herrühren. Diese Ausführung kann u. a. auf eingebetteten Systemen, auf Hardware oder durch einen Interpreter stattfinden. Hierbei sind ein niedriger Speicherverbrauch und eine niedrige Laufzeit des Monitors wichtige Kriterien.

Die aus diesen Aspekten abgeleiteten Anforderungen an eine Zwischensprache basierend auf dem Prozess der Monitorgenerierung aus modellbasierten Spezifikationen sind im Folgenden spezifiziert.

AP1) *Formale Spezifikation der Syntax und Semantik:* Zum Einsatz einer Sprache als Zwischensprache in einem Monitorgenerierungsprozess muss diese formal definiert sein. Die hierdurch festgelegte Ausführungssemantik verhindert falsche Interpretation der Signaturen und erlaubt die Formalisierung der Spezifikationssprache durch Abbildung auf die Zwischensprache.

Anforderung	Kategorie			
	Ausdrucksstärke	Modellierung	Generierung	Ausführung
AP1 Formale Spezifikation		x	x	x
AP2 Determ. Semantik		x		
AP3 Keine Zustandsr.-Expl.			x	
AP4 Effizient ausfürbar				x
AP5 Parallele Kommun.	x			
AP6 Verschränkte Überw.	x			
AP7 Einfache Generierung			x	
AP8 Kompakte Repräsent.		x	x	
AP9 Zeitaspekte		x		
AP10 Aufteilbarkeit		x	x	
AP11 Bedingte Überwachung	x			x
AP12 Gegenmaßnahmen	x			x
AP13 Plattformunabh.	x		x	
AP14 Erl. u. verb. Verhalten	x			
AP15 Deontische Bedingung	x			
AP16 Ereignisklassen		x		x
AP17 Lok./glob. Variablen	x			

Tabelle 3.3: Kategorisierung der Anforderungen an die Zwischensprache

AP2) *Deterministische Ausführung/Semantik:* Die Sprache sollte eine deterministische Auswertung der ihr übergebenen Ereignisse ermöglichen. Dies könnte auch durch Einschränkungen der Modellierung gewährleistet werden, birgt jedoch ein hohes Fehlerpotenzial, wenn diese nicht von sich aus deterministisch ist. Eine nicht-deterministische Ausführung der Signaturen ist für das Monitoring und die Ausführung von Gegenmaßnahmen nicht sinnvoll, da die Entscheidungen durch das System selbst getroffen werden und Gegenmaßnahmen keine zufälligen Entscheidungen beinhalten sollten.

AP3) *Keine Zustandsraumexplosion:* Da die Signaturen u. a. zur Codegenerierung für eingebettete Systeme genutzt werden sollen, darf ihre Modellierung keine sehr großen Modelle erzeugen. Dies würde zu hohem Speicherbedarf für die Speicherung des Monitoringcodes führen.

AP4) *Effizient ausführbar:* Die in der Zwischensprache repräsentierten Signaturen werden zur Laufzeit eines Systems überwacht. Hierzu muss der generierte Monitor parallel zum System ausgeführt werden, was zu einem Laufzeitoverhead führt. Der als Zwischensprache verwendete Formalismus muss effizient ausführbar sein, um den Overhead in einem zu überwachenden System niedrig zu halten.

AP5) *Abbildung paralleler zu überwachender Kommunikationen/Prozesse:* Die Überwachung vieler verschiedener Instanzen eines Teilsystems sollte schon in der Semantik der Zwischensprache als integriertes Konzept enthalten sein.

AP6) *Verschränkte Überwachungsversuche von Signaturen:* Eine Signatur (besonders Fehlverhalten) muss nicht nur sequenziell überwacht werden, sondern auch verschränkt. Sequenzielle Überwachung bedeutet dabei, dass erst die Überwachung einer Signaturinstanz beendet sein muss, bevor diese erneut überwacht wird. Verschränkte Überwachung startet mit jedem Auftreten des ersten in einer Signatur erwarteten Ereignisses einen neuen Überwachungsversuch, während die vorherigen noch laufen. Hierdurch werden zueinander verschränkt mehrere Überwachungsversuche durchgeführt.

AP7) *Einfache Generierung von Code für verschiedene Plattformen:* Die Zwischensprache sollte eine möglichst einfache und explizite Syntax mit wenigen essenziellen Elementen und eine eindeutige Semantik besitzen. Durch wenige Konzepte, die keine aufwendige Analyse ihrer impliziten Semantik erfordern, kann eine einfache Codegenerierung für verschiedene Plattformen, umgesetzt werden.

AP8) *Kompakte Repräsentation der Quellspezifikation:* Ziel der Zwischensprache muss eine explizite aber dennoch kompakte Repräsentation verschiedener komplexer Quellspezifikationen sein. Sie sollte in ihrer Semantik Konzepte beinhalten, die es erlauben den Umfang der Spezifikation zu reduzieren und daraus folgend den Speicherbedarf des generierten Monitors bzw. der zu interpretierenden Spezifikation möglichst klein zu halten.

AP9) *Abbildung/Modellierung von konkreten Zeitaspekten:* In der Zwischensprache müssen zeitliche Aspekte modellierbar sein. Bei der Laufzeitüberwachung von Signaturen müssen zeitliche Abbruchkriterien modelliert werden, da ansonsten bei nicht auftretenden Ereignissen keine Entscheidung getroffen werden kann, ob ein Fehler aufgetreten ist. Des Weiteren müssen häufig auch zeitliche Aspekte, wie die Frequenz des Auftretens eines Ereignisses, überwacht werden.

AP10) *Aufteilbarkeit auf verschiedene Teilsysteme:* Nicht in allen zu überwachenden Systemen besteht die Möglichkeit einen zentralen Monitor einzusetzen. Sei es durch eingeschränkte Verfügbarkeit der benötigten Ereignisse und Daten oder durch einen geschützten Kommunikationskanal. Die modellierten Monitore müssen deshalb ggf. zur Einsparung von Speicherplatz und zur Reduzierung der Laufzeit auf die miteinander kommunizierenden Komponenten frei aufgeteilt werden. Bei der Generierung von Monitoren aus Spezifikationssprachen, die die Unterscheidung zwischen verschiedenen Komponenten unterstützen, sollte in der Zwischensprache eine Aufteilung der Signaturen in Teilmonitore vorgesehen werden, um eine verteilte Überwachung zu unterstützen. Diese Teilmonitore sollten auch als Wrapper um einzelne Komponenten einsetzbar sein.

AP11) *Bedingte Überwachung:* In Signaturspezifikationssprachen existieren häufig Konzepte zur Formulierung von Vorbedingungen, die die Anwendbarkeit einer Signatur einschränken. Erst wenn diese Vorbedingung erfüllt wurde, muss der Rest der Signatur eingehalten werden. Diese bedingte Überwachung in den Spezifikationssprachen muss auch durch die Zwischenspra-

che abgedeckt werden können. Zur möglichst kompakten und laufzeiteffizienten Repräsentation der Signaturen in einer Zwischensprache sind Beziehungen zwischen Signaturen sinnvoll, die bestimmen zu welcher Zeit eine Signatur auftreten kann und diese auch nur dann überwachen.

AP12) *Ausführung von Gegenmaßnahmen (policy enforcement):* Wenn durch den Monitor eine Abweichung zum erlaubten Ablauf oder ein modellierter Fehler gefunden wurde, kann es notwendig sein eine Gegenmaßnahme durchzuführen, die das System wieder in einen sicheren Zustand versetzt. Die Zwischensprache muss eine Modellierung von ausführbaren Gegenmaßnahmen unterstützen. Diese sollten in der Zwischensprache selbst spezifiziert werden können.

AP13) *Plattform- und Sprachunabhängigkeit:* Die Zwischensprache muss plattform- und programmiersprachenunabhängig sein. Durch diese Eigenschaft kann die Sprache universell in verschiedensten Generierungsprozessen für Monitore eingesetzt werden. Sie sollte weder von der Spezifikationssprache noch von der Zielsprache/-pattform abhängig sein. Erst zur Generierungszeit sollten zielplattformspezifische Informationen eingewoben werden.

AP14) *Erlaubtes und verbotenes Verhalten:* Die Zwischensprache muss die Modellierung von erlaubtem und verbotenem Verhalten unterstützen. Hierdurch kann neben der Erkennung von Fehlverhalten, wie bei der klassischen signaturbasierten Überwachung stattfindet, auch eine Anomalieerkennung, die auf positiven Signaturen basiert, durchgeführt werden. Durch die Beschreibung von erlaubtem Verhalten werden auch Angriffe erkannt, die vorher nicht bekannt sind. Die Reaktion darauf ist jedoch im Allgemeinen nicht spezifisch möglich.

AP15) *Abbildung von deontischen Bedingungen:* Viele Ansätze verwenden nur klassische Aussagenlogik zur Formulierung von Signaturen. Eine Zwischensprache, die eine möglichst große Anzahl verschiedener Spezifikationssprachen repräsentieren soll, muss jedoch auch erweiterte Logiken wie deontische Logik abbilden können. Die deontische Logik ergänzt die klassische Aussagenlogik durch Begriffe wie die Erlaubnis und die Verpflichtung zur Formulierung normativer Aussagen. Diese sind z. B. durch die Signatursprache eLSC, die die Modallogik unterstützt, abgedeckt [SES07] und müssen auch auf die Zwischensprache abbildbar sein.

AP16) *Ereignisklassen zur abstrakteren Beschreibung von Szenarien und Zusammenfassung von Ereignissen:* Bei der Beschreibung von Angriffsszenarien können viele bis auf die annotierten Ereignistypen identische Signaturen entstehen. Zur effizienten Repräsentation solcher Signaturen in einer Zwischensprache können diese durch abstraktere Signaturen mit allgemeinen Ereignisklassen spezifiziert werden. Hierzu ist es nötig Ereignisse auch in Ereignisklassen zu gruppieren. Dies führt zu weniger zueinander ähnlichen Signaturen, die sich nur in den überwachten Ereignissen unterscheiden, und spart Speicherplatz auf der Zielplattform.

AP17) *Lokale und globale Variablen:* Zur Unterstützung der Überwachung von Daten, die durch Ereignisse übertragen werden und Berechnung von Werten, benötigt die Zwischensprache ein Konzept zur Zwischenspeicherung von diesen Daten zur Laufzeit. Diese Variablen müssen in Bedingungen zur Evaluation des Monitorergebnisses herangezogen werden können. Hierbei muss zwischen Variablen, die an eine Signaturinstanz gebunden sind und Variablen, die global für eine Signatur gelten unterschieden werden können. Zusätzlich sind Variablen, die durch verschiedene Signaturen verwendet werden können, wünschenswert.

Diese Anforderungen dienen als Ausgangspunkt für den Vergleich von verwandten Arbeiten sowie der Entwicklung und Definition einer an die Laufzeitüberwachung angepassten Zwischensprache für Werkzeugketten zur Monitorgenerierung.

4

VERWANDTE ARBEITEN – ÜBERWACHUNGSTECHNIKEN FÜR SYSTEME

In diesem Kapitel werden verwandte Arbeiten zum Thema Überwachungstechniken behandelt. Hierbei werden Sprachen zur Spezifikation von Monitoren, Werkzeuge und Zwischensprachen, die zur Generierung und Beschreibung von Monitoren verwendet werden, betrachtet.

Bei der Überwachung von Systemen, unabhängig davon, ob es sich um „Intrusion Detection"-Systeme oder um Laufzeitmonitore handelt, werden zu erkennende Muster als Signaturen in einer Spezifikationssprache beschrieben. Diese Signaturen definieren entweder positive oder negative Muster, die dementsprechend interpretiert werden. Sie werden während der Überwachung auf der Basis von zur Laufzeit des Systems auftretenden Ereignissequenzen ausgewertet.

Im Folgenden werden existierende Ansätze gruppiert nach ihren Einsatzgebieten betrachtet. Hierfür werden in Abschnitt 4.1 Ansätze der Modellierung von negativen Anwendungsfällen (Misusecases) betrachtet, die auch als Basis der in [Pat14] vorgestellten MUC-Sprache dienen. Diese *Misusecases* werden in den betrachteten Ansätzen zur Absicherung von Systemen vor Angriffen eingesetzt. Abschnitt 4.2 beschreibt anschließend verschiedenste Ansätze und Sprachen aus dem Bereich der *„Intrusion Detection"-Systeme*. Der Bereich der *Laufzeitüberwachung* und *Laufzeitverifikation* wird in Abschnitt 4.3 betrachtet. Hierbei werden Ansätze, die auf verschiedenen Varianten von *Temporalen Logiken*, *Automaten* und *Petrinetzen* basieren, analysiert. Des Weiteren werden auf *UMLsec* und *SecureUML* basierende Ansätze betrachtet und es wird als spezielle Bereiche der Laufzeitüberwachung auf *Design by Contract* und den Einsatz der *aspektorientierten Programmierung* eingegangen. Als spezielles Einsatzgebiet zur Integration des MBSecMon-Prozesses wurde in dieser Arbeit die *AUTOSAR-Methodik* gewählt, die ebenfalls grundlegende Überwachungstechniken unterstützt. Diese Ansätze werden in Abschnitt 4.4 besprochen. Abschließend findet in Abschnitt 4.5 eine Bewertung einer Auswahl der vorgestellten Ansätze in Bezug auf die in Abschnitt 3.5 aufgestellten Anforderungen an eine Zwischensprache für den MBSecMon-Prozess statt.

4.1 Misusecases zur Beschreibung von negativen Signaturen

Usecases [RQJZ07] (UCs), wie sie in der UML2 verwendet werden, werden zur Beschreibung positiver Szenarien (Teilabläufe), die in einem System auftreten, eingesetzt. Sie ermöglichen die Beschreibung funktionaler Anforderungen an das System. Diese Usecases werden z. B. durch Interaktionsdiagramme der UML2 oder Message Sequence Charts genauer spezifiziert. Sie werden u. a. in der Anforde-

rungsphase zur Beschreibung des Systemverhaltens und zur Testfallbeschreibung verwendet.

Sindre et al. [SO01] führt zusätzlich zu den positiven Usecases negative Szenarien (*engl. Misusecases*) ein. Diese Misusecases werden zur Modellierung von Fehlverhalten eingesetzt und über spezielle, neue Beziehungen mit Usecases in Verbindung gesetzt. Die hierdurch modellierbaren nicht-funktionalen Anforderungen an das System [Ale03] beschreiben auch bösartige Akteure, die negativ auf das System einwirken.

Whittle et al. [WWH08] beschreibt einen Ansatz zur Modellierung und Ausführung von Misusecases, die mit Extended Interaction Overview Diagrams (EIODs) verfeinert werden. EIOD ist eine formalisierte Variante der UML2-Interaktionsübersichtsdiagramme, die um Beziehungen von Use- und Misusecases erweitert sind. Ziel dieses Ansatzes ist nicht die Erzeugung von Laufzeitmonitoren, sondern EIODs als ausführbare Spezifikation zur Unterstützung bei der Entwicklung eines sicheren Systems zu verwenden. Hierbei werden sie in eine Menge von untereinander kommunizierenden Finite State Machines (FSM) übersetzt [WJ06]. Die Abläufe der einzelnen Lebenslinien der Sequenzdiagramme werden in jeweils einen Teilautomaten übersetzt, der sich über Ereignisse mit den anderen Teilautomaten synchronisiert. Spezielle Mitigationsszenarien[1] beschreiben Gegenmaßnahmen zu bekannten Angriffen in Form von Sequenzdiagrammen. Sie werden als aspektorientierte Szenarien modelliert, die in den Ablauf der Usecase-Szenarien eingewoben werden [WJ08]. Misusecases dienen zur Beschreibung von Angriffen auf das spezifizierte System. Mit diesen Angriffsmustern kann die durch die Mitigationsszenarien erweiterte ausführbare Spezifikation (FSM) stimuliert werden, um sicherzustellen, dass die Gegenmaßnahme den Angriff verhindert.

Systeme, die mithilfe dieses Ansatzes entwickelt wurden, werden gegen die zur Entwicklungszeit bekannten Angriffe abgesichert. Abweichungen vom erwünschten Verhalten oder unbekannte Angriffe können vom System nur abgewehrt werden, wenn sie eine identische Manifestation (sie werden durch Sequenzdiagramme bekannter Angriffe erkannt) im Systemablauf aufweisen. Die verwendete Technik könnte jedoch auch für die Entwicklung von Monitoren eingesetzt werden, indem kommunizierende FSMs bzw. High-Level FSMs als Zwischensprache zur Monitorgenerierung verwendet werden. Hierzu müssten Misusecases jedoch so modelliert werden, dass sie potenzielle Attacken nicht nur auslösen, sondern erkennen können.

4.2 Intrusion Detection Systeme

„Intrusion Detection"-Systeme (IDS) werden eingesetzt, um Angriffe auf Computersysteme oder -netzwerke zu erkennen. Auf Basis von im System oder Netzwerk auftretenden Ereignissen werden Muster erkannt, die auf einen Angriff oder Missbrauch schließen lassen.

[1] Mitigation bedeutet Schadensminderung. Mitigationsszenarien beschreiben Gegenmaßnahmen für einen Angriff und somit Änderungen an den das System beschreibenden Sequenzdiagrammen (Usecases). Diese Änderungen werden ggf. auch an mehreren Stellen der Systembeschreibung vorgenommen.

IDS existieren in zwei unterschiedlichen Ausprägungen, die zur Laufzeit des Systems oder auf aufgezeichneten Protokoll- bzw. Auditdaten arbeiten. Der verbreitetste Typ ist die *Signaturanalyse* (*engl. misuse detection*), die auf Basis vorgegebener Muster bekannte Angriffe erkennt. Hierbei werden Muster modelliert, die auf einen Angriff schließen lassen. Im Gegensatz dazu wertet die *Anomalieerkennung* (*engl. anomaly detection*) das Normalverhalten eines Systems statistisch aus und meldet auf Basis dieser Daten durch Verwendung von Metriken Abweichungen vom Normalverhalten. Ein Indiz für eine „Denail of Service"-Attacke, die das System überlasten soll, kann zum Beispiel eine erhöhte Anzahl von Verbindungsversuchen in einem festgelegten Zeitintervall sein. Die Anomalieerkennung erlaubt es im Gegensatz zur Signaturanalyse auch vorher unbekannte Angriffe zu erkennen, ist jedoch deutlich komplexer zu konfigurieren und führt durch ihre niedrigere Erkennungsgenauigkeit zu einer erhöhten „False-Positive"-Rate (Fehlalarme). Durch die genaue Einordnung einer Attacke sind bei IDS mit Signaturanalyse auch gezielte Gegenmaßnahmen möglich.

IDS werden zusätzlich nach der Komplexität der zu erkennenden Angriffsmuster kategorisiert. Hierbei findet eine Unterscheidung zwischen Einschritt- und Mehrschrittattacken statt. Einschritt-IDS basieren zum Großteil auf Stringvergleichen, die verdächtige Byte-Sequenzen im Auditeintrag erkennen. Mehrschritt-IDS sind in der Lage auch komplexere Analysen auf den Auditdaten auszuführen und damit komplexe Mehrschrittattacken zu erkennen.

Ein weiteres Unterscheidungskriterium ist die Datenbasis, auf die das IDS zurückgreifen kann. Hier wird zwischen netzwerk- und hostbasierten IDS unterschieden. Netzwerkbasierte IDS arbeiten auf den protokollierten Daten des Netzwerks und können an zentraler Stelle des Netzwerks den Verkehr überwachen. Hostbasierte IDS sind im Allgemeinen leistungsfähiger in Hinsicht auf die Einbruchserkennung, beeinflussen jedoch die Leistungsfähigkeit des zu überwachenden Systems. Dies resultiert daraus, dass sie auf Betriebssystemereignissen auf einem Zielsystem in einer gesicherten Umgebung arbeiten. Des Weiteren existieren auch hybride IDS, die netzwerk- und hostbasierte Komponenten kombinieren.

IDS existieren in den verschiedensten Ausführungen, die für bestimmte Einsatzgebiete die beschriebenen Techniken kombinieren. Hierbei existieren viele kommerzielle und einige freie IDS. Zu den freien gehören u. a. *SNORT*[2], *Samhain*[3] und *Prelude*[4]. Das weitverbreitete IDS *SNORT* ist ein signaturbasiertes, netzwerkbasiertes Einschritt-IDS. *Samhain* ist im Gegensatz dazu ein host-basiertes System, das nur den lokalen Computer überwacht und den Netzwerkverkehr ignoriert. *Prelude* ist ein hybrides IDS, das andere IDS und damit netzwerk- und hostbasierte Komponenten integriert. Zu den kommerziellen IDS gehören die Werkzeuge zum Bau von Expertensystemen *CLIPS* [GR98] und *P-BEST* [LP99]. Hierbei stellt CLIPS drei verschiedene Programmierparadigmen zur Verfügung: regelbasiert, objektorientiert und prozedural. P-BEST unterstützt den Bau von angepassten bzw. optimierten Inferenzengines, die Signaturen (Regeln) von Expertensystemen auswerten.

[2] SNORT-Website: http://www.snort.org/
[3] Samhain-Website: http://la-samhna.de/samhain/
[4] Prelude-Website: http://www.prelude-ids.com/

Bei der Umsetzung der Signaturen in ausführbare Repräsentationen können grundlegend zwei Typen unterschieden werden. Die einen übersetzen die einzelnen Signaturen in Programmmodule, während die anderen *Expertensysteme* [GR98] zur Ausführung nutzen.

Zur ersten Kategorie gehören dabei Systeme wie STAT [VEK00] und IDIOT [Kum95]. Diese Systeme übersetzen die modellierten Signaturen in einzelne C++-Programmmodule, die voneinander unabhängig die aufgezeichneten Ereignissequenzen (Auditdaten) analysieren. Redundante Spezifikationen werden durch diesen Ansatz zur Laufzeit mehrfach ausgewertet.

Im Gegensatz dazu arbeiten Expertensysteme wie CLIPS [GR98] und *Production-Based Expert System Toolset* (P-BEST) [LP99] mit Inferenzmechanismen. Sie basieren auf deklarativen Inferenzregeln mit einer Wenn-Dann-Struktur. Wenn die Vorbedingung einer Regel erfüllt ist, wird eine Aktion auf den Fakten bzw. Daten, die den aktuellen Zustand des zu überwachenden Systems beschreiben, ausgeführt. Diese Aktion kann zur Erfüllung weiterer Vorbedingungen anderer Regeln führen. Die Basis der Fakten wird somit von dem System selbst und den Inferenzregeln verändert, was wiederum Aktionen auslösen kann. Des Weiteren können zeitliche Regeln über den Fakten spezifiziert werden. Den Vorteil von der Abbildung der Spezifikation auf Regeln eines Expertensystems stellen die einfache Struktur der Regeln und die Verwendung optimierter Inferenzalgorithmen basierend auf RETE- [For82] und Markovalgorithmen [GR98] zur Ausführung dar. Hierbei wählt die Inferenzengine eine passende Regel aus, führt den Dann-Teil aus und wiederholt den Vorgang, bis keine der Regeln mehr anwendbar ist. Die Expertensysteme werden dann in die IDS integriert.

Der Vorteil dieser Art der Spezifikation ist die Trennung von Problembeschreibung und Kontrollalgorithmus, der die Regeln auswertet und anwendet. Ein großer Nachteil ist, dass die linke Seite der Produktionsregeln (Wenn-Teil) keine vorgegebene Sequenz besitzt und damit eine effiziente Reihenfolge der Anwendung der Regeln für den Inferenzalgorithmus schwer zu bestimmen ist. Expertensystemregeln können somit in einem Monitorgenerierungsprozess als eine Zielsprache gesehen werden, da diese durch Inferenzalgorithmen zur Laufzeit des IDS ausgewertet werden und aus ihnen im Normalfall keine weiteren Übersetzungen in andere Zielsprachen durchgeführt wird.

In signaturbasierten „Intrusion Detection“-Systemen werden bekannte Misusecases (Attacken) definiert, die erkannt werden sollen. Diese müssen jedoch zur Modellierungszeit bekannt sein. Zur Erkennung unbekannter Attacken werden signaturbasierte Ansätze mit der Anomalieerkennung wie statistisches Profiling [Den87], Neural Networks [DBS92], und Sequenzanalyse [FHSL96] kombiniert, die das System auf Basis von vorher analysiertem Verhalten auf Abweichungen zu diesem überwachen.

Im Gegensatz zu den Monitor-Petrinetzen agieren Expertensysteme durch ihre einfachen Regeln auf einer relativ komplexen Faktenbasis, die zur Laufzeit im Datenspeicher vorgehalten werden muss. Des Weiteren müssen Informationen zu den durchgeführten Regelanwendungen gespeichert werden. Dies führt bei der Überwachung komplexer Signaturen (vieler aufeinander aufbauender Regeln) dazu, dass der auf eingebetteten Systemen beschränkt zur Verfügung ste-

hende Datenspeicher stark belastet wird. Monitor-Petrinetze speichern nur die aktuelle Markierung des Netzes, da viele der Zusammenhänge durch das Netz beschrieben sind und somit statisch im Codespeicher vorgehalten werden können.

Das Ziel der Monitor-Petrinetze ist es zwischen erwünschtem und verbotenem Verhalten bei der Signaturbeschreibung zu unterscheiden. Hierdurch ist es auch möglich bei zur Modellierungszeit nicht bekannten Attacken den gewünschten Ablauf (Usecase) zu modellieren und Abweichungen von diesem als Fehler zu erkennen. Das Zurückgreifen auf Anomalieerkennungsansätze ist in der Praxis nicht sinnvoll, da diese Ansätze schwer zu konfigurieren sind und eine hohe False-Positive-Rate besitzen. Somit können die Monitore basierend auf der Anomalieanalyse bei Erkennung eines Angriffs keine eigenständigen Gegenmaßnahmen auslösen. Im Fall von IDS wird bei solchen Ansätzen häufig nur eine Warnung erzeugt, die durch einen Menschen eingeschätzt werden muss.

Im Folgenden werden deshalb Sprachen zur signaturbasierten Spezifikation von IDS betrachtet. Hierbei werden die zu dieser Arbeit ähnlichsten Ansätze zur Modellierung und Generierung von Monitoren genauer vorgestellt.

STAT (STATE TRANSITION ANALYSIS TECHNIQUE) Das Rahmenwerk STAT [VEK00] zur Entwicklung von IDS verwendet die Spezifikationssprache STATL, die auf Zustands-Transitionssystemen basiert. In STAT werden Angriffe auf Computer als Sequenzen von Ereignissen (Aktionen) repräsentiert, die Übergänge im Sicherheitszustand des Systems auslösen. STAT wurde in host-basierten (USTAT) und netzwerkbasierten IDS (NetSTAT) verwendet. Jede Signatur besitzt einen Startzustand und mindestens einen Endzustand, der die Kompromittierung des Systems signalisiert. Als Annotationssprache wird die Programmiersprache C eingesetzt. In einem Vorbereich können Konstanten und Variablen deklariert werden, die in den Signaturen genutzt werden können. Globale Variablen können von allen Instanzen einer Signatur verwendet werden, während die Sichtbarkeit lokaler Variablen auf die entsprechende Instanz eingeschränkt ist. Die Signaturen besitzen eine einfache Struktur, die aus Zuständen und Transitionen besteht. Zustände werden als Invarianten beschrieben und Transitionen als Bedingungen, Ereignisse und Aktionen. Transitionen haben verschiedene Typen wie konsumierend, nichtkonsumierend und rückabwickelnd (*engl. unwinding*), die es ermöglichen, dass mehrere Zustände im Netz gleichzeitig aktiv sind. Um die Interpretation der Signaturen zu erleichtern, existieren keine Aufspaltung (Fork) und keine Zusammenführung (Join), wodurch es nur einen Vorgänger und einen Nachfolger einer Transition gibt. Aus dieser Einschränkung der Sprache resultiert eine Zustandsexplosion der Signaturen, da bei parallelen Abläufen jede Permutation als eigener Zustand modelliert werden muss. STATL unterstützt ein Timerkonzept. Die aus den Signaturen generierten Programmmodule werden in der STATL-Toolsuite eingebunden und überwacht.

IDIOT (INTRUSION DETECTION IN OUR TIME) IDIOT [Kum95] verwendet eine angepasste Form der ausdrucksstarken gefärbten Petrinetze (CPNs) – die Colored Petri Automata (CPA) –, die mit einer funktionalen General Purpose Lan-

guage annotiert werden. Eine Signatur, die Misusecases beschreibt, besteht aus beliebig vielen Startplätzen und einem Endplatz. Die Pfade zwischen ihnen beschreiben die zu findenden Ereignissequenzen. Der Hauptunterschied zu CPNs [Jen91] ist die Abwesenheit von Nebenläufigkeit. Hierdurch kann für ein Ereignis nur eine Transition im CPA schalten. CPAs unterstützen keine lokalen Transitionsvariablen, sondern nur dem gesamten CPA zugewiesene Variablen. Jedes Token trägt dabei die Werte der dem CPA zugewiesenen Variablen und ist damit gefärbt. Eine Signatur muss ein zusammenhängendes Netz bilden. Des Weiteren ist es möglich Vor- und Nachbedingungen, sowie Invarianten für Signaturen zu definieren. Vorbedingungen und Invarianten ermöglichen die Spezifikation von Bedingungen, die nicht oder schwer mit CPAs ausdrückbar sind und Nachbedingungen können zum rekursiven Aufruf derselben Signatur genutzt werden.

Ziel dieses Ansatzes ist ein möglichst allgemeiner Formalismus zur Beschreibung von Ereignissequenzen, der in IDS eingesetzt werden kann. Die Verwendung von CPAs führt zu einer ausdrucksstarken, aber schwer verständlichen (unübersichtlichen) Spezifikation. Zur Überwachung dieser systemunabhängigen Signaturen wird eine Matching-Engine eingesetzt. Durch die Einschränkung der CPA des nicht-nebenläufigen Schaltens von Transitionen kommt es im Gegensatz zu den CPNs bei komplexen Spezifikationen zu größeren Netzen. Bei der Abbildung von alternativen Abläufen, die teilweise die selben Ereignistypen beinhalten, kommt es wie bei STAT zu einer Explosion der Größe des Netzes.

EDL (EVENT DESCRIPTION LANGUAGE) Die Signaturbeschreibungssprache *Event Description Language* (EDL) [MS05, FM12] ist als ganzheitliche Modellierungssprache für „Intrusion Detection"-Systeme entwickelt worden. Sie wird genutzt um Signaturen zu modellieren (*engl. signature nets*), die Muster bestehend aus mehreren Schritten beschreiben. Ziel bei der Entwicklung dieser Sprache war die Verbesserung der Modellierungsfreundlichkeit im Gegensatz zu STATL und IDIOT. Der Modellierungsansatz von EDL basiert syntaktisch wie STATL auf gefärbten Petrinetzen, wobei die Sprache die in [Mei04] für Signaturen geforderten notwendigen semantischen Konzepte ausdrücken kann (siehe Abschnitt 3.5). Die Semantik unterscheidet sich jedoch von Petrinetzen. Die EDL definiert eine deterministische Schaltregel, die alle Transitionen, die schalten können, auch im Falle eines Verzweigungskonflikts auslöst. Die Beschreibung der Signaturen findet textuell statt, jedoch können die Signaturen auch grafisch dargestellt werden. Transitionen beschreiben hierbei Aktionen und Plätze Systemzustände. Es existieren vier Platztypen – Initial-, Final-, Escape- und Interior-Plätze. Initialplätze besitzen zu Beginn der Überwachung ein Token, das während der Überwachung über Interior-Plätze zu Exit- bzw. Escape-Plätzen wandert. In einer Signatur darf es nur einen Exit-Platz, der für die Erkennung einer Signatur steht und beliebig viele Escape-Plätze, die das Token aus dem Netz entfernen und damit die Überwachung abbrechen, geben. Zur Abstraktion des zur Überwachenden Systemzustandes besitzen Plätze Merkmale (Sets von Variablen), die den Systemzustand, soweit wie zur Spezifikation der Attacke benötigt, beschreiben. Dabei werden bei diesem Variablenkonzept die Datentypen Bool, Float, String und Integer unterstützt. Token, die durch das Netz wandern, tragen Werte, die den Merkmalen der

Plätze zugeordnet sind. Durch schaltende Transitionen, die einen Merkmalsbindungsblock besitzen, können die Wertebelegungen der Token verändert werden. Ob Token durch das Schalten der Transition von ihren Vorplätzen entfernt werden, wird durch die Konfiguration der Kanten zwischen Plätzen und Transitionen als konsumierend bzw. nicht-konsumierend festgelegt. Zur Spezifikation komplexer Ereignisse (Ereignissequenzen) bietet die EDL ein Hierarchisierungskonzept an, das auf der der Sprache SHEDEL [MBH02] basiert. In diesem Konzept können Transitionen selbst als Folgen von Transitionen (signature net) beschrieben werden. Die Bindung von Variablenwerten an einzelne Token ist ein mächtiges Konzept, jedoch komplex in Monitoren umzusetzen. Für die Speicherung dieser Werte der einzelnen Token wird zur Laufzeit Datenspeicher, der von der Anzahl der Token im Netz und der maximalen Anzahl verschiedener Variablenbindungen der Token abhängt, benötigt. Für den Einsatz auf eingebetteten Systemen ist hierzu eine komplexe Analyse auf den EDL-Signaturen notwendig, um eine obere Grenze des benötigten Datenspeichers zu bestimmen.

Auch bei diesen auf Zustands-Transitionssystemen basierenden IDS-Ansätzen werden, wie bei IDS üblich, nur negative Signaturen, die eine Attacke beschreiben, unterstützt. Die Signaturen in der Spezifikationssprache EDL werden zur Ausführung auf einer Analyseeinheit auf Regeln eines Expertensystems abgebildet. Die als Aktionen und in Bedingungen verwendbaren Funktionen basieren auf den von der Analyseeinheit (Zielplattform) angebotenen Funktionen. Hierbei sind auch Reaktionen (Gegenmaßnahmen), die auf den durch die Analyseeinheit angebotenen Funktionen basieren, beim Auslösen einer Transition möglich. Jedoch müssen im Gegensatz zu der in dieser Arbeit eingeführten MPN-Sprache auch bei der EDL alle möglichen Pfade durch das Netz (positive und negative) explizit modelliert werden. Des Weiteren ist das Konzept der gefärbten Token, deren Variablenbindungen von den belegten Plätzen abhängt relativ komplex für die Codegenerierung.

SCHLUSSFOLGERUNG „Intrusion Detection"-Systeme sind für den Einsatz auf PCs/Servern und den Netzwerken zwischen ihnen entwickelt worden. In signaturbasierten IDS werden Angriffe modelliert, die erkannt werden sollen. Es zeigt sich, dass der Formalismus der Petrinetzdialekte für die Spezifikation systemunabhängiger Ereignissequenzen geeignet ist. Jedoch eignen sich die hier vorgestellten Ansätze nur eingeschränkt für die Verwendung auf eingebetteten Systemen, da diese nicht auf die eingeschränkten Speicher- und Rechenzeitanforderungen hin entwickelt wurden. Bei allen diesen Ansätzen müssen alle möglichen Pfade durch die Signatur explizit modelliert werden, sodass alle Token aus dem Netz abgeräumt werden. Dies führt bei komplexen Signaturen zu relativ umfangreichen Netzen. Reaktionen bzw. Gegenmaßnahmen basieren in den betrachteten Signaturbeschreibungssprachen immer auf dem Aufruf von Funktionen der Analyseeinheit. Die Unterscheidung mehrerer Kommunikationssequenzen und die verschränkte Überwachung einer Signatur ist nicht Teil dieser Sprachen, sondern wird durch mehrfache Instanziierung der Signaturen durch die ausführende Einheit erreicht. Auch dies führt zu einem erhöhten Speicherverbrauch.

4.3 Laufzeitverifikation und Laufzeitüberwachung

Die Laufzeitüberwachung (*engl. runtime monitoring*) und die Laufzeitverifikation (*engl. runtime verification*) haben zum Ziel Programme zur Laufzeit gegen formale Spezifikationen zu überprüfen. Ein Laufzeitverifikationssystem generiert hierfür Überwachungscode, der durch Instrumentierung in das zu überwachende System integriert wird. Hierfür muss ein Laufzeitverifikationssystem folgende Schritte unterstützen:

1. Spezifikation der zu überwachenden Aspekte in einem Formalismus.
2. Generierung von Code aus dieser Spezifikation, der parallel zum Zielsystem oder im Zielsystem eingesetzt werden kann. Für die Transformation von formaler Spezifikation zu ausführbaren Überwachungscode ist ein Monitorsynthesealgorithmus notwendig.
3. Verbinden des Zielsystems mit dem Monitorcode durch Instrumentierung.

Hierbei bestimmt die Wahl des Spezifikationsformalismus die Ausdrucksstärke des Monitorgenerierungssystems.

In diesem Abschnitt werden Ansätze aus dem Bereich der Laufzeitverifikation betrachtet. Hierbei wird insbesondere auf die verwendeten Sprachen der Ansätze eingegangen. Darunter sind Sprachen, die zur Spezifikation der Monitore, und Sprachen, die als Zwischensprachen zur Generierung eingesetzt werden.

Im Folgenden werden existierende Ansätze der Laufzeitverifikation genauer betrachtet. Die Unterteilung der Unterabschnitte findet nach dem Typ der verwendeten Spezifikationssprache und ihrem Einsatzgebiet statt. Zunächst werden die weitverbreiteten Spezifikationssprachen aus dem Bereich der *Temporalen Logiken* anhand einiger Beispielsprachen betrachtet. Darauf folgend werden modellbasierte Ansätze, die *Automaten* bzw. *Petrinetze* zur Spezifikation einsetzen, vorgestellt. Der Abschnitt *Design by Contract* stellt anschließend eine zur Laufzeitverifikation verwendete Art der Spezifikation mittels Vor-, Nachbedingungen und Invarianten (Assertions) direkt im Programmcode vor. Eine modellbasierte Variante für den Einsatz von Assertions wird im Abschnitt *UMLsec und SecureUML* besprochen. Die *Aspektorientierte Programmierung* wird auf ähnliche Weise zur Spezifikation von Bedingungen, die im Quelltext zur Laufzeit überprüft werden sollen, verwendet. Zusätzlich wird dieser Ansatz u. a. im *Monitor-oriented Programming* auch zur Anbindung von Laufzeitmonitoren an die zu überwachenden Programme (Instrumentierung) eingesetzt.

4.3.1 *Temporale Logiken*

Eine Verifikation realer Systeme ist aufgrund ihrer Größe oft nicht mehr möglich. Des Weiteren wird bei der Verifikation von Systemen ein formales Modell des Systems auf Korrektheit geprüft und nicht die Implementierung selbst, die viel detaillierter sein kann und sich nicht zwingend in allen Details an die formale Spezifikation halten muss. Aus einer solchen Spezifikation können Testfälle generiert werden, um die Implementierung zu testen. Jedoch kann hierbei nur

eine (repräsentative) Untermenge der möglichen Eingaben überprüft werden. Die Verifikation der formalen Spezifikation (statische Analyse) und das Testen der Implementierung können somit nicht garantieren, dass ein implementiertes System sich zur Laufzeit korrekt verhält. Um dieses Problem zu umgehen, werden in der *Runtime Verification* Laufzeitmonitore aus formalen Spezifikationen synthetisiert. Diese stellen sicher, dass die aktuelle Ausführung korrekt ist. Danach können sie entweder offline für Debuggingzwecke auf gesammelten Daten oder online zur Überprüfung von Eigenschaften eines Programms oder einer Kommunikation zur Laufzeit eingesetzt werden. Beispiele für Runtime-Verification-Rahmenwerke für logikbasiertes Monitoring sind *Monitoring and Checking* (MaC) [KKL+01], *PathExplorer* (PaX) [HR01], *Eagle* und seine Abwandlung *RuleR* [BRH10]. Alle diese Ansätze bzw. ihre Umsetzungen verwenden ihre eigene festgelegte Spezifikationssprache aus dem Bereich der Temporalen Logiken. Eine Übersicht über verschiedene Varianten der Temporalen Logiken ist in [Var01] zu finden.

- **MaC** bietet zwei Spezifikationssprachen: *Primitive Event Definition Language* (PEDL), die für die Beschreibung von Ereignissen zuständig ist und die *Meta Event Definition Language* (MEDL) zur Beschreibung von High-Level Anforderungen (Invarianten). Die Sprache basiert auf der Interval Temporal Logic und unterstützt die Modellierung anhand von Ereignissen und Bedingungen. MEDL erlaubt zusätzlich die Definition von Hilfsvariablen, die zum Beispiel zum Zählen von Ereignissen genutzt werden können.

- **Java PaX** bietet für logikbasiertes Monitoring ein Rahmenwerk zur Definition von High-Level-Anforderungen und ihrer darunterliegenden Logik in der Sprache *Maude*. Bei Maude handelt es sich um eine Rewriting-Logik, die es erlaubt Logiken zu definieren. Unterstützt werden Future-Time (LTL) und Past-Time-Linear-Temporal Logic (PTLTL). Das Maude-Rahmenwerk wird anschließend zum Abgleich zwischen beobachteten Ereignissen und den temporalen Formeln verwendet.

- **Eagle** unterstützt Future-Time- und Past-Time-Logiken. Die Eagle-Logik ist eine eingeschränkte Fixpunkt-LTL erster Ordnung mit Verkettung (*engl. chop*) über endliche Ereignissequenzen (Traces).

- **RuleR** definiert ein Subset der Eagle-Sprache, mit dem Ziel einer effizienteren Ausführung.

Die beiden kommerziellen Werkzeuge Temporal Rover [Dru00] und sein Nachfolger DBRover erwarten als Eingabe ein Programm, das mit LTL/MTL-Formeln, die als Kommentare im Quellcode annotiert sind, versehen ist. Diese LTL-Formeln werden in Quelltext überführt, der in das Programm eingebettet wird. Sie haben festgelegte Spezifikationsformalismen (LTL) und können für verschiedenste Programmiersprachen Code generieren.

Ein weiterer Ansatz, der auf Linear Temporal Logic (LTL) zur Spezifikation von Laufzeitmonitoren setzt, wird in [BLS06] beschrieben und in [BJ08, BJ10] auf seine Anwendbarkeit auf Safety- und Co-Safety-Eigenschaften evaluiert. Die verwendete LTL unterstützt eine dreiwertige Logik für Wahrheitswerte. Diese ermöglicht es, bei Verwendung der Standardsematik der LTL, endliche Ereignissequenzen zur

Laufzeit zu überwachen. In dieser Logik wird, wenn ein Präfix erkannt wurde und dessen Postfix nicht mehr fehlschlagen kann, der Wert „true“ und wenn die Formel nicht mehr erfüllt werden kann „false“ zurückgegeben. In allen anderen Fällen wird ein „uneindeutig“ (*engl. inconclusive*) zurückgegeben.

Die Formeln, die entweder in LTL oder in Timed LTL formuliert sind, werden in einen Monitor übersetzt. Für LTL-Formeln werden nach Konstruktion der negativen LTL-Formel, die beiden Formeln (positive und negative Formel) einzeln über verschiedene nicht-deterministische Automaten und abschließender Potenzmengenkonstruktion zu deterministischen Automaten transformiert. Anschließend wird aus ihnen eine FSM mit dreiwertiger Logik konstruiert. Diese durch die einzelnen Transformationsschritte im Verhältnis zur Formel exponentiell gewachsene FSM wird im Weiteren durch Standardtechniken reduziert. Für TLTL-Formeln wird kein deterministischer Automat erzeugt, sondern der Monitor aus einem *Region-Automat*, der einem Büchi-Automaten [Büc66] ähnelt, direkt generiert.

Die Abbildung unendlicher Ereignissequenzen, die durch LTL beschrieben werden, haben dazu geführt, dass die Autoren der Ansätze diese durch unterschiedliche Interpretation der LTL-Formeln auf endlichen Systemtraces abbilden mussten. Somit unterscheiden sich die Eingabesprachen der unterschiedlichen Ansätze trotz eines gemeinsamen TL-Formalismus. Quellspezifikationen, die in diesen Ansätzen eingesetzt werden sollen, müssen zunächst in die Eingabesprache des Ansatzes übersetzt werden. Dieser Einsatz verschiedener Temporaler Logiken in den Werkzeugen erschwert die Nutzung bzw. den Wechsel eines solchen Rahmenwerks, wenn vorhandene Spezifikationen wiederverwendet werden sollen. Ein Ansatz, der sich zum Ziel gesetzt hat verschiedenste Eingabesprachen in einem Rahmenwerk zu unterstützen, ist das *Monitor-oriented Programming* (MOP) Rahmenwerk [MJG+12], das in Abschnitt 4.3.7 näher vorgestellt wird.

SCHLUSSFOLGERUNG In den vorgestellten Ansätzen dienen Temporale Logiken zumeist als Spezifikationssprache und nicht als Zwischensprache, die dann in Code oder vorher in Automaten, die als Zwischensprache dienen, übersetzt werden. Diese Automaten, sind zumindest während der Generierung der Monitore von einer Zustandsraumexplosion betroffen. Selbst wenn diese Automaten durch Reduktionstechniken vor der Codegenerierung optimiert werden, besitzen sie die Nachteile der im nächsten Abschnitt vorgestellten Ansätze, die Automaten zur Spezifikation einsetzen. Im Gegensatz zu der auf Petrinetzen basierenden MPN-Sprache kommt es bei der Repräsentation von nebenläufigen verschränkten Signaturen, wie sie in der MBSecMon-Sprache vorkommen, auch bei der Abbildung auf Temporale Logik zur Explosion der Größe der Formeln und in den optimierten Automaten zu einer Explosion der Größe des Netzes.

4.3.2 *Automaten*

Zur Spezifikation von Signaturen, die überwacht werden sollen, werden in der Literatur auch Automaten eingesetzt. Hierbei werden in den folgenden Ansätzen, die ein generisches Security-Monitoring unterstützen sollen, positive Signaturen

spezifiziert, die das gewünschte Verhalten beschreiben. Security-Policies verhindern damit Systemverhalten, das als nicht akzeptabel (vom spezifizierten Verhalten abweicht) spezifiziert ist.

Einer der ersten Ansätze zur Beschreibung einer Klasse durchsetzbarer Security-Policies sind die Security Automata [Sch00]. Sie werden in einer Klasse von Büchi-Automaten [Büc66] spezifiziert. Bei diesem Ansatz werden Sicherheitsrichtlinien durchgesetzt, indem ein zum Programm parallel laufender Monitor die Aktionen des Programms gegen die spezifizierten Security-Policies überwacht und diese durchsetzt, indem er bei Erkennung einer Abweichung von den Security-Policies das System terminiert. Ein solcher Automat (Truncation Automata) besteht aus Zuständen und Eingaben. Eine Policy kann beliebig viele Initialzustände enthalten. Die Übergänge zwischen den Zuständen sind durch eine Übergangsfunktion beschrieben, die aus Kombination von Zustand und Eingabe auf die Potenzmenge der Zustände abbildet. Eingabesymbole können dabei u. a. Systemzustände, atomare Aktionen oder Zustandsaktionspaare sein.

Diese Security Automata besitzen nur eine sehr eingeschränkte Reaktionsmöglichkeit auf unerwünschtes Verhalten. *Edit Automata* [LBW05a, LBW05b] sind ein Rahmenwerk, das es erlaubt Aktionen von nicht vertrauenswürdigen Programmen zur Laufzeit abzufangen und zu modifizieren. Programmmonitore (Edit Automata), die auf Security Automata basieren, werden als Finite-State- oder abzählbare Infinite-State-Automaten (*engl. abstract machines*) spezifiziert. Diese Monitore untersuchen die Sequenz von Programmaktionen bevor sie ausgeführt werden und greifen in diese Sequenz ein, wenn sie von der spezifizierten Policy abweichen. In der Literatur werden zusätzlich zu den *Truncation Automata* (TA) der Security Automata zwei Klassen von Monitoren definiert: *Suppression Automata* (SA) und *Insertion Automata* (IA), deren Möglichkeiten des Eingriffs in das Systemverhalten (den Aktionsstrom) ansteigt. Während der TA bei Abweichung von der Spezifikation das überwachte System unterbricht, kann der SA die nicht erlaubte Aktion unterdrücken und der IA kann die verbotene Aktion durch eine andere ersetzen, die das System wieder in einen stabilen Zustand überführt. Edit Automata sind eine Kombination des IA und des SA.

Eine neue Art der Monitorbeschreibung in diesem Bereich sind *Service Automata* [GMS12], die auf den Security bzw. Edit Automata aufbauen. Die Service Automata können zusätzlich verteilte Systeme in einer dezentralisierten, koordinierten Art sichern, indem sie verteilte miteinander agierende Monitore beschreiben. Diese überwachen die Ausführung verteilter Programme und setzen Gegenmaßnahmen durch, bevor eine Verletzung einer spezifizierten Security-Policy eintreten kann. Im Gegensatz zu Security Automata sind sie in der Lage auch nicht-lokale Sicherheitsanforderungen durchzusetzen. Service Automata sind mittels Kommunizierenden Sequenzielle Prozesse (*engl. Communicating Sequential Processes*) (CSP) formalisiert. Die Umsetzung findet hierbei als Handimplementierung in Java statt. Ziel ist es, diese Handimplementierung der Monitore durch die in CSP beschriebenen Spezifikationen zu verifizieren.

SCHLUSSFOLGERUNG Die hier vorgestellten Formalismen werden zur Spezifikation von Monitoren, die Policies durchsetzen, verwendet. Hierbei ist es nicht

das Ziel dieser Ansätze aus den Spezifikationen Monitore zu generieren, sondern die Policies formal zu beschreiben und die implementierten Monitoringmechanismen zu verifizieren.

4.3.3 *Petrinetze*

Zur direkten Spezifikation von Monitoren werden Petrinetze außerhalb vom IDS, wie sie in Abschnitt 4.2 vorgestellt wurden, nur selten eingesetzt. Eine der wenigen Ansätze ist die von Frankowiak et al. [FGP09] eingeführte leicht erweiterte Version der Petrinetze. Diese besitzt spezielle Plätze, die Token generieren (*engl. token generators*) und Plätze in denen Token abgeräumt werden (*engl. bins*). Die Tokengeneratoren erzeugen basierend auf Ereignissen außerhalb des Netzes neue Token, die dann durch das Netz wandern. Es gibt je Petrinetz nur einen Tokengenerator aber ein bis beliebig viele Endplätze. Einzelne Petrinetze beschreiben nur Teilabläufe des zu überwachenden Systems und können über ein Kontrollnetz verknüpft werden. Sogenannte *Output Transitions*, die es ermöglichen Ergebnisse nach außen zu schicken, können dazu verwendet werden je nach Implementierung des Systems einen Alarm auszulösen.

SCHLUSSFOLGERUNG Frankowiak zeigt, dass Petrinetze dazu geeignet sind, Monitore zu spezifizieren und daraus für eingebettete Controller Quelltext zu generieren. Im Gegensatz zu den in dieser Arbeit vorgestellten MPNs führen komplexe Spezifikationen mit der in [FGP09] vorgestellten Version der Petrinetze zu großen Netzen, da alle möglichen Ereignissequenzen und alle Abbrüche der Überwachung mittels Transitionen modelliert werden müssen. MPNs besitzen in ihrer Semantik ein implizites Abbruchkriterium der Überwachung einer Signatur. Ein Endplatz stellt bei Frankowiak zudem immer nur das Ende eines Prozesses dar und ermöglicht keine Unterscheidung zwischen positiven und negativen Enden eines Petrinetzes. Ansätze wie der von Kumar et al. [Kum95], der gefärbte Petrinetze als Modellierungssprache für Monitore mit der funktionalen Programmiersprache ML einsetzt, führt zwar zu einer kompakten Repräsentation der Signaturen jedoch in Folge der Komplexität der Sprache zu einer aufwendigen Codegenerierung. Hierdurch ist die Unterstützung neuer Plattformen und Zielsprachen sehr aufwendig. Für den Einsatz auf einem eingebetteten System muss für eine Zwischensprache ein Kompromiss zwischen ihrer Komplexität (Anzahl der Konzepte) und ihrer Kompaktheit (Größe der Spezifikation) gefunden werden.

4.3.4 *Design by Contract*

Eine weitere Möglichkeit der Laufzeitüberwachung eines Programms ist das Prinzip des *Design by Contract* (DbC) [Mey88]. Hierbei wird der Quelltext, ähnlich wie bei den auf Temporaler Logik basierenden Ansätzen, durch semantische Spezifikationen erweitert. Diese werden während der Kompilierung zu Invarianten und Assertions übersetzt und während der Laufzeit des Programms überprüft. Das

Konzept wurde erstmals in der Programmiersprache *Eiffel*[5] eingeführt, wobei es das Ziel ist, das Zusammenspiel verschiedener Programmmodule durch diese semantischen Spezifikationen sicherzustellen. Dies soll die Korrektheit eines Systems sicherstellen und die Robustheit erhöhen. Hierzu werden Vorbedingungen formuliert, die den Zustand des Systems beschreiben, in dem z. B. eine Methode korrekt arbeitet. Nachbedingungen überprüfen nach der Ausführung der Methode, ob sie sich korrekt verhalten hat. Des Weiteren gibt es Klasseninvarianten, die alle Instanzen zur Laufzeit des Programms erfüllen müssen.

Für den Einsatz in Java existieren verschiedene Ansätze. *Jass* [BFMW01] ist ein Präcompiler, der es erlaubt Java-Programme nach dem DbC-Prinzip mit Spezifikationen in Form von Assertions, die zur Laufzeit ausgewertet werden, zu erweitern. Die Spezifikationen, die als speziell formatierte Kommentare in den Quelltext geschrieben werden, werden durch den Präcompiler nach Java übersetzt. Zusätzlich zu Assertions (Vorbedingungen, Nachbedingungen und Invarianten) unterstützt Jass *Refinement checks*, die Rahmenbedingungen für Vererbungen festlegen, und *Trace assertions*. Trace assertions ermöglichen dynamisches Verhalten zu überwachen, wie es u. a. mit Temporaler Logik möglich ist. Darunter fallen z. B. die Aufrufreihenfolge von Methoden und die Anzahl von Aufrufen in einer bestimmten Zeitspanne. Hierzu wird eine Notation verwendet, die ähnlich der Prozessalgebra *Communicating Sequential Processes* (CSP) [Hoa85] ist. Die hierbei überwachbaren Ereignisse sind Beginn und Ende von Methodenaufrufen.

Ein zweiter Ansatz ist jContractor [KHB99], der das DbC-Konzept mithilfe einer Java-Bibliothek in Java integriert. Die Spezifikation wird bei diesem Ansatz als Java-Methoden mit speziellen Namenskonventionen formuliert. Zusätzlich zu Assertions werden Quantifikatoren und Operatoren der Prädikatenlogik wie *Forall*, *Exists*, *suchThat* und *implies* in Assertions unterstützt.

Des Weiteren existieren verhaltensbasierte Schnittstellenbeschreibungssprachen, wie die Java Modeling Language (JML) [CKLP06] und Spec# [BLS05]. Diese beschreiben Vor- und Nachbedingungen von Methodenaufrufen. Die JML ist eine solche verhaltensbasierte Schnittstellenbeschreibungssprache für Java, die es erlaubt eine Teilmenge der JML zur Laufzeit zu überprüfen. In Assertions wird die Notation von Java verwendet, die durch verschiedene Spezifikationskonstrukte wie Quantifikatoren erweitert wurde. Sie werden als Annotationen in Kommentaren dem Java-Quelltext hinzugefügt. Zusätzlich können FSMs spezifiziert werden, auf deren Zustand in den Assertions zurückgegriffen werden kann. Für die objektorientierte Programmiersprache C# existiert eine Erweiterung Spec# [BLS05], die das DbC-Konzept für C# umsetzt. Diese ist vom Umfang der unterstützten Sprachmittel sehr ähnlich zu JML. Vorbedingungen und Nachbedingungen werden in ausführbaren Inlined-Code übersetzt.

SCHLUSSFOLGERUNG Alle diese DbC-Konzepte sind zielsprachengebunden und bieten eine unterschiedliche Ausdrucksstärke. Die Syntax der Spezifikationssprachen ist unterschiedlich und an die zu unterstützenden Programmiersprachen angelehnt. Sie erlauben die Spezifikation und Überprüfung von Regeln, die Objekte und Methoden eines Programms während der Laufzeit einhalten müssen.

[5] Eiffel-Website: http://www.eiffel.com/

Eine übergeordnete Überwachung der Kommunikation zwischen Komponenten, in denen der zeitliche Ablauf eine Rolle spielt, wie sie in dem in dieser Arbeit vorgestellten MBSecMon-Ansatz erreicht werden soll, lässt sich mit diesem auf Assertions und Invarianten basierten Ansatz nicht umsetzen. Die Erweiterung des DbC-Konzepts in Jass mit der Prozessalgebra CSP führt dazu, dass auch Abfolgen von Ereignissen bzw. parallele Abläufe überwacht werden können. Im MBSecMon-Prozess würde diese Sprache als Spezifikationssprache gelten, da ein CSP zur Ausführung in einen Automaten übersetzt würde. Sie würden somit auf eine Zwischensprache abgebildet werden.

4.3.5 *UMLsec und SecureUML*

Neben den im Design by Contract (DbC) eingesetzten Sprachen um Programmcode mit Assertions zu annotieren, existieren modellbasierte Ansätze, die insbesondere die *Unified Modeling Language* (UML) erweitern.

UMLsec [Jür02, Jür05] ist eine Erweiterung der UML, die sich zum Ziel gesetzt hat die Entwicklung sicherer Systeme zu unterstützen. UMLsec annotiert UML-Modelle mit sicherheitsrelevanten, nicht-funktionalen Zusatzinformationen. Es handelt sich um eine leichtgewichtige Erweiterung der UML, die zusätzliche Eigenschaften (Tags) und Constraints definiert, wobei UML-Elemente mittels Stereotypen markiert werden. Als Spezifikationssprache wird die *Object Constraint Language* (OCL) eingesetzt. Diese Annotationen dienen in erster Linie zur Verifikation der Systemspezifikation, die in UML spezifiziert ist.

Zur Laufzeitverifikation des implementierten Systems werden diese UMLsec-Sicherheitsanforderungen in verschiedene Zwischensprachen übersetzt, aus denen anschließend Monitore generiert werden. So werden Sicherheitsanforderungen, die in UMLsec spezifiziert wurden, in LTL übersetzt [BJ10], um sie durch den Ansatz von [BLS06] in Monitore zu überführen. Diese basieren auf der dreiwertigen Semantik der LTL von [BLS06] und werden dann, wie in Abschnitt 4.3.1 vorgestellt, über verschiedene nichtdeterministische Automaten in eine Finite State Machine (FSM) transformiert. Auf Basis dieser FSM werden Monitore generiert, die die in UMLsec spezifizierten Anforderungen zur Laufzeit überprüfen.

Ein weiterer Ansatz ist die Übersetzung der in UMLsec spezifizierten Sicherheitsanforderungen in die im DbC (Abschnitt 4.3.4) eingesetzte Sprache *Java Modeling Language* (JML). JML wird in [LJ09] verwendet, um die in UMLsec auf Basis von UML-Diagrammen spezifizierten Sicherheitsanforderungen in den zu überwachenden Code als Assertions einzubetten. Die Übersetzung von UMLsec zu JML wurde händisch durchgeführt, mit dem Ziel den Code gegen die UMLsec-Spezifikation zu verifizieren.

Neben vielen weiteren Profilen erweitert UMLsec die UML um die Definition von *Role-based Access Control* (RBAC). Bei RBAC werden Zugriffe auf Operationen und Dateien eingeschränkt, indem Benutzern des Systems Rollen mit speziellen Rechten zugewiesen werden. Für die Laufzeitverifikation des in UMLsec definierten Profils zur Spezifikation von RBAC-Eigenschaften werden diese Eigenschaften in die Zielsprache Java bzw. AspectJ übersetzt [MJH+10]. Die AspectJ-Variante hat gegenüber der Übersetzung in objektorientierten Java-Code den Vorteil, dass

die Eigenschaften vom eigentlichen Programmcode getrennt sind und die RBAC-Eigenschaften zur Laufzeit durchgesetzt werden können. Hierbei wird das *Java Authentication and Authorization Service*-Rahmenwerk (JAAS) eingesetzt.

Im Gegensatz zu UMLsec fokussiert sich SecureUML [LBD02, BDL06] hauptsächlich auf die Spezifikation von RBAC. Zusätzlich zu RBAC werden Autorisierungsbedingungen (*engl. authorization constraint*) eingeführt, die als Vorbedingung für den Zugriff auf eine Operation dienen. SecureUML ist im Gegensatz zu UMLsec eine schwergewichtige Erweiterung der UML. SecureUML unterstützt die Generierung von Java-Code für RBAC-Eigenschaften auf Basis von *Enterprise JavaBeans* (EJB). Jedoch werden die ebenfalls in OCL spezifizierten Eigenschaften nicht vor der Generierung des Codes auf dem UML-Modell selbst verifiziert.

SCHLUSSFOLGERUNG Auch in diesem Bereich werden verschiedene Zwischensprachen zur Abbildung der zu überwachenden Eigenschaften eingesetzt. Die Kombination der UMLsec-Eigenschaften mit dem Ansatz der Monitorgenerierung von [BJ10] aus einer dreiwertigen LTL, der in Abschnitt 4.3.1 vorgestellt wurde, ergibt sich ein ähnlicher Monitorgenerierungsprozess, wie der hier vorgestellte MBSecMon-Prozess. Die Generierung der Monitore geht über verschiedenste Automaten als Zwischensprachen und mündet in einer FSM. Bei der Übersetzung ergibt sich eine Zustandsexplosion, die durch Optimierungstechniken auf der generierten FSMs abgemildert werden muss. Das Ziel der Monitor-Petrinetze ist eine einzige auf die Laufzeitüberwachung zugeschnittene Zwischensprache zur Beschreibung von Ereignissequenzen zu spezifizieren, die im Gegensatz zu FSMs auch nebenläufige Abläufe kompakt darstellen kann.

4.3.6 *Aspektorientierte Programmierung*

Aspektorientierte Programmierung (AOP), die in Abschnitt 3.4 vorgestellt wurde, eignet sich zur Instrumentierung von Programmcode und zum Einweben von Assertions (*engl. Advices*) an definierten Stellen (*engl. Pointcuts*) in den auszuführenden Code. Somit lassen sich mit dieser Technik wie bei den „Design by Contract"-Ansätzen in Abschnitt 4.3.4 Vor- und Nachbedingungen, sowie Invarianten definieren, die zur Laufzeit des Programms überwacht werden. Hierbei wird zur Spezifikation die Syntax des entsprechenden AOP-Rahmenwerks verwendet.

Zusätzlich zu Assertions unterstützen Implementierungen der AOP wie AspectJ auch sogenanntes *Controllflow-based Croscutting*, das die Einbindung von zuvor eingetretenen Pointcuts in die Bedingungen eines Pointcuts erlaubt. Dieses Konzept kann auch zur Spezifikation von Laufzeitmonitoren verwendet werden, indem die Reihenfolge, in der Pointcuts erreicht werden sollen, durch Bedingungen spezifiziert werden und somit Fehler im Ablauf erkannt werden. Dies ist jedoch nur sehr eingeschränkt möglich, da nur positive Verläufe modelliert werden können und die Spezifikation unter Verwendung der Pointcuts sehr unübersichtlich wird. Für komplexere Ablaufspezifikationen werden in Ansätzen, die auf AOP basieren, formale Spezifikationssprachen verwendet, die den Ablauf der durch die Pointcuts erzeugten Ereignisse beschreiben. Ein Beispiel hierfür ist das im folgenden Abschnitt beschriebene Rahmenwerk MOP.

Im MBSecMon-Prozess selbst sind die AOP-Sprachen eine Zielsprache [Lan13] und werden u. a. zur Anbindung der Monitore an Java-Programme eingesetzt. Die Spezifikation der Ereignisabfolgen findet großteils in der MPN-Sprache statt.

4.3.7 *Monitor-oriented Programming*

Monitor-oriented Programming (MOP) [MJG+12] ist ein Rahmenwerk zur automatischen Generierung von Laufzeitmonitoren. Es hat dasselbe Ziel wie der in dieser Arbeit vorgestellte MBSecMon-Prozess, unterscheidet sich jedoch darüber hinaus beträchtlich in der Umsetzung.

MOP stellt abstrakt die grundlegenden Mechanismen zur Spezifikation und Generierung von Monitoren zur Verfügung. Es ist damit unabhängig von der Zielsprache, in der die Monitore generiert werden. MOP bietet ein Logik-Repository, in dem vorgefertigte Logik-Plugins eine grundlegende Unterstützung von Spezifikationssprachen zur Verfügung stellen.

Dieses abstrakte Rahmenwerk muss erweitert werden, um verschiedene Zielsprachen zu unterstützen. Hierbei müssen für jede Zielsprache zwei verschiedenen Modultypen bereitgestellt werden. Zum einen ist dies ein Spezifikationsprozessor, der die grundlegende Unterstützung für die Zielsprache implementiert, und zum anderen muss für jede Spezifikationssprache eine Implementierung des durch das Rahmenwerk bereitgestellten abstrakten Logik-Plugins erstellt werden. Hierdurch ist das MOP-Rahmenwerk sehr flexibel und kann auf die vorgestellte Weise erweitert werden.

Es existieren zwei Instanzen des MOP-Rahmenwerks:

- JavaMOP ist eine Instanz, zur Generierung von Laufzeitmonitoren in der Zielsprache Java. Die generierten Monitore eignen sich somit für die Überwachung von Java-Programmen. JavaMOP unterstützt durch den Formalismus des Parametrischen Trace Slicings die Zuordnung von Ereignissen zu Instanzen der Java-Klassen des überwachten Programms.

- BusMOP ist eine im Funktionsumfang eingeschränkte Instanz, die für die Generierung von Hardwaremonitoren entworfen wurde. Sie unterstützt die Zielsprache VHDL und die generierten Monitore überwachen den Datenverkehr auf einem PCI-Bus. Im Gegensatz zu JavaMOP wird keine Parametrisierung unterstützt.

Im Folgenden wird genauer auf JavaMOP eingegangen. Die Spezifikation dieser Signaturen findet in einer textuellen Syntax statt, die an den JavaMOP-Compiler übergeben werden. Diese Spezifikationen bestehen hierbei grundlegend aus zwei Teilen:

- Einer Definition der Ereignisse, basierend auf der AspectJ-Syntax und

- einer formalen Beschreibung der erlaubten Ereignissequenzen.

Der Compiler generiert aus den Signaturen daraufhin Monitore, die in das zu überwachende Java-Programm eingewoben werden. Hierbei kann das MOP-Rahmenwerk über Logikplugins um weitere Spezifikationssprachen erweitert werden.

Spezifikationssprache	Eigenschaften		
	Erfüllung	Verletzung	Zwischensprache
Finite State Machine (FSM)	-	✓	FSM
Extended Regular Expressions (ERE)	✓	✓	FSM
Context Free Grammar (CFG)	✓	✓	LookAheadLR(1)-Parser
Linear Temporal Logic (LTL)	-	✓	FSM
String Rewriting System (SRS)	✓	✓	-
Past Time Linear Temporal Logic (PTLTL)	✓	✓	FSM + Tabelle
PTLL with Calls and Returns (PTCaRet)	✓	✓	-

Tabelle 4.1: Durch MOP unterstütze Spezifikationssprachen

LOGIKPLUGINS FÜR JAVAMOP JavaMOP unterstützt eine Vielzahl an Spezifikationssprachen. Tabelle 4.1 zeigt eine Übersicht über die von JavaMOP unterstützten Formalismen. Sie zeigt welche Aussagen sich mit diesen Sprachen formulieren lassen und welche als Zwischensprache für die Codegenerierung genutzt werden.

- **Erfüllung** bedeutet, dass der Monitor auf die Erfüllung der Spezifikation reagieren kann.
- **Verletzung** bedeutet, dass der Monitor Verletzungen der Spezifikation erkennt und daraufhin eine Aktion auslösen kann.

Die Ausgabe der Logik-Plugins ist in allen Fällen Pseudocode, der anschließend in die Zielsprache übersetzt wird. Zu beachten ist, dass als grundlegende Zwischensprache die auch als Spezifikationssprache verwendeten FSMs genutzt werden, auf die ERE und Temporale Logiken abgebildet werden. SRS und PTTLCaRet werden im Gegensatz dazu direkt in die Pseudocoderepräsentation übersetzt. Bei der Entwicklung des MOP-Rahmenwerks stellte die Generierung effizienter Laufzeitmonitore aus CFG-Spezifikationen [MJCR08] den Hauptfokus dar. Diese werden in einen LALR(1)-Parser übersetzt. Grund für die CFG-Spezifikation ist hierbei die höhere Ausdrucksmächtigkeit im Gegensatz zu regulären Ausdrücken und Temporalen Logiken, die sich auf einfache FSMs reduzieren lassen.

Hierbei dienen die verschiedenen Spezifikationssprachen nur als Formalismus zur Beschreibung von Ereignissequenzen. Die Aspekte, die die Ereignisse beschreiben, können zusätzlich durch Bedingungen eingeschränkt werden und Daten auf lokale Variablen schreiben. Globale Variablen, die zwischen verschiedenen Signaturen zum Austausch von Daten genutzt werden, sind durch die Verwendung von AspectJ nicht realisierbar.

Beispiel *Signatur eines JavaMOP-Laufzeitmonitors*

Die in JavaMOP spezifizierte Monitorspezifikation in Auflistung 4.1 überwacht, dass aufeinanderfolgende *InitReq*-Nachrichten eine streng monoton ansteigende Sequenznummer besitzen.

Die Syntax von JavaMOP ist stark an die von AspectJ angelehnt. Im Kopf der Signatur werden der Signaturname – *Sequencenumbers* – und die beteiligten

Auflistung 4.1: Signatur eines JavaMOP-Laufzeitmonitors

```
import example.domain.*;
//Kopf
Sequencenumbers(Vehicle v, RSU r) {
 //Variablen
 private int seq_old = -1;
 //Ereignisse
 event initreq before(Vehicle v, RSU r, int id, int seq) :
  this(v)
  && target(r)
  && args(id,seq)
  && call(* example.domain.RSU.initReq(int,int))
  && condition(seq > seq_old) {
  seq_old = seq;
 }
 event falseinitreq before(Vehicle v, RSU r, int id, int seq) :
  this(v)
  && target(r)
  && args(id,seq)
  && call(* example.domain.RSU.initReq(int,int))
  && condition(seq >= seq_old) {
  seq_old = seq;
 }
 //Spezifikation
 cfg: S -> initreq initreq*
 fsm: initreq initreq*

 @match{
  System.out.println("Spezifikation eingehalten");
 }
 @fail{
  System.out.println("Spezifikation nicht eingehalten");
 }
}
```

Klassen – *Vehicle* und *RSU* – angegeben. Darauf folgt beliebiger Programmcode in der Zielsprache, der direkt in den generierten Monitor übernommen wird. In diesem Fall ist es die Definition einer Variablen *seq_old*, die die vorherige Sequenznummer zwischenspeichert. Ereignisse werden ähnlich zu den Pointcuts in AspectJ formuliert und können somit mittels Bedingungen eingeschränkt werden. Im Beispiel werden zwei Ereignisse definiert. Zum einen das *initreq*-Ereignis, das beim Aufruf der Methode *initReq* ausgelöst wird, wenn die aktuelle Sequenznummer *seq* größer als die vorherige ist. Zum anderen ist ein Ereignis *falseinitreq* definiert, das beim selben Methodenaufruf erzeugt wird, mit dem Unterschied, dass die aktuelle Sequenznummer nicht größer als die vorherige ist. Im Gegensatz zu AspectJ ist keine separate Definition eines Advices notwendig, sondern es kann in der Ereignisdefinition direkt beliebiger Java-Code angegeben werden. Im Beispiel wird jedes Mal wenn eines der bei-

den Ereignisse aufgerufen wird die aktuelle Sequenznummer auf die Variable *seq_old* gespeichert.

Auf die Definition der Ereignisse folgt die Spezifikation des Monitors. Hierbei darf nur eine der beiden Spezifikationen, die im Beispiel gezeigt sind, auf einmal verwendet werden. Sowohl die Kontextfreie Grammatik (CFG) als auch die Finite State Machine (FSM) beschreiben, dass beliebig oft hintereinander das *initreq*-Ereignis auftreten muss. Wenn sich die Sequenznummer nicht erhöht, wird das Ereignis *falseinitreq* ausgelöst und die Monitorspezifikation schlägt fehl. Die Schlüsselwörter `@match` und `@fail` erlauben auf eine Erfüllung und auf eine Verletzung der Signatur zu reagieren. In den entsprechenden Bereichen wird beliebiger Code in der Zielsprache eingefügt, der bei der Erfüllung bzw. Verletzung der Signatur ausgeführt wird.

PARAMETRISCHES MONITORING Zur Abbildung von Kommunikationssträngen verwendet MOP *Parametrisches Monitoring* (*engl. parametric monitoring*) [RC12], in dem Ereignisse parametrisiert sind und somit zur Laufzeit z. B. Instanzen des überwachten Systems zugeordnet werden können.

Definition 18 (Kommunikationsstrang). *Ein Kommunikationsstrang stellt eine Menge miteinander kommunizierender Einheiten dar. Verschiedene Kommunikationsstränge unterscheiden sich in ihren dazugehörigen kommunizierenden Einheiten. Eine Einheit kann jedoch in mehreren Kommunikationssträngen vorkommen.*

Dieses Verfahren basiert auf *Trace Matching* [AAC+05], das es ermöglicht einen Trace bezüglich der Einhaltung einer formalen Spezifikation zu überprüfen. Trace-Matching beachtet jedoch nicht die Zuordnung von Ereignissen an einen bestimmten Kommunikationsstrang, weshalb in MOP Trace-Matching auf Parametrisches Monitoring erweitert wird. Hierbei übernimmt die das Ereignis auslösende Komponente die Aufgabe, die Zuordnung des Kommunikationsstranges dem Ereignis mitzugeben. Durch diese Zuordnung kann in MOP zur Laufzeit für jeden Kommunikationsstrang ein einzelner Monitor initialisiert werden.

SCHLUSSFOLGERUNG Das MOP-Rahmenwerk unterstützt eine Vielzahl verschiedener Spezifikationssprachen und bietet durch Logikplugins die Möglichkeit aus diesen Monitoringcode zu generieren. Die Zielsetzung des Rahmenwerks ist ähnlich zu der des MBSecMon-Rahmenwerks. Jedoch sind die Spezifikationssprachen stark zielsprachengebunden, da die auf aspektorientierter Programmierung basierende Spezifikation der Ereignisse an die Zielsprache angelehnt ist. Des Weiteren ist die Spezifikation durch die Trennung der Spezifikation von Ereignissen, die auch Bedingungen enthalten können, und die Spezifikation der Ereignisabfolgen in einer zweiten formalisierten Sprache nicht vollständig formalisiert. Im MOP-Rahmenwerk werden hauptsächlich FSMs mit ihren Vor- und Nachteilen als Spezifikations- bzw. Zwischensprache eingesetzt.

4.3.8 *Schlussfolgerung*

Um die Vielfalt der Eingabesprachen und deren direkte Übersetzung in Monitore zu reduzieren, ist es das Ziel dieser Arbeit eine universelle Zwischensprache für die Monitorgenerierung formal zu spezifizieren, auf die Quellspezifikationen abgebildet werden können. Im Verlauf dieser Arbeit wird gezeigt werden, dass die neu eingeführten Monitor-Petrinetze dazu geeignet sind, als Zwischensprache für die Generierung von Monitoren aus verschiedensten Spezifikationssprachen zu fungieren. Diese Zwischensprache vereinfacht die Generierung von Monitoren für beliebige Zielsprachen und -plattformen.

4.4 LAUFZEITÜBERWACHUNG IN AUTOSAR

Der AUTOSAR-Standard [KF09], der in Abschnitt 2.1 vorgestellt wurde, sieht mehrere integrierte Überwachungstechniken vor. Diese werden vom Betriebssystem (*engl. Operating System*) (OS) und vom Watchdogmanager (WdM) zur Verfügung gestellt.

In der Spezifikation des Betriebssystems [AUT11a] werden drei verschiedene Techniken beschrieben. Alle beziehen sich auf die Überwachung von Zeitbedingungen zur Laufzeit. Die *Execution time protection* stellt sicher, dass ein vorher definiertes Zeitbudget für einzelne Tasks und *Category 2 Interrupt Service Routines* (C2ISR) zur Laufzeit nicht überschritten wird. Diese Zeitbudgets werden als statische obere Grenzen definiert. Die *Locking time protection* (LTP) überwacht die Zeit, die eine Ressource von einem Task oder einer C2ISR gesperrt wird. Sie beinhaltet alle Unterbrechungszeiten (*engl. suspension time*) der OS-Interrupts und alle Interrupts des Systems. Die LTP verhindert Prioritätsinvertierungen und dient zur Wiederherstellung eines funktionsfähigen Systemzustands nach potenziellen Verklemmungen (*engl. deadlock*). Die *Inter-arrival time protection*, überwacht die vergangene Zeit zwischen erfolgreichen Aufrufen von Tasks und der Ankunft von C2ISRs.

Die gerade vorgestellten in der Spezifikation des Betriebssystems definierten Überwachungsmechanismen werden auf OS-Level konfiguriert und beschreiben zeitliche Zusammenhänge zwischen Tasks. Sie eignen sich nicht, um Kontrollfluss- oder Datenflussanalysen zu spezifizieren. Im Gegensatz dazu erlaubt das in [PPPM13] vorgestellte in den AUTOSAR-Prozess integrierte MBSecMon-Rahmenwerk die Spezifikation von Kontroll- und Datenflussanalysen zur Laufzeit. Hierbei werden nicht Tasks, sondern auch die Systemsicht des Modells mit einbezogen.

Auch die AUTOSAR-Spezifikation des Watchdogmanagers [AUT11b] bietet drei Mechanismen an, die komplementär zu denen des AUTOSAR-Betriebssystems sind. Grundsätzlich basieren die Mechanismen auf Überwachungsstellen (*engl. checkpoints*), die, wenn sie erreicht werden, dies an den Watchdog-Manager (WdM) melden. Hierzu müssen zu überwachende Funktionen mit Aufrufen an den WdM instrumentiert werden. Sie ermöglichen die Überwachung der Reihenfolge der Transition der Überwachungsstellen und deren zeitlicher Zusammenhänge. Bei der *Alive supervision* überprüft der WdM in periodischen Abständen, ob die Über-

wachungsstellen, der überwachten Komponente, in festgelegten Zeitschranken erreicht wurden. Hierdurch ist es möglich Tasks zu erkennen, die zu häufig oder zu selten laufen. Die *Deadline supervision* erlaubt dagegen die Überwachung aperiodischer oder episodischer Ereignisse, die spezielle Bedingungen an die zeitlichen Abläufe zwischen den Überwachungsstellen stellen. Es wird die Ausführungszeit ganzer Blöcke überwacht, ob sich diese in den vorgegebenen Grenzen befindet. Der WdM bietet als dritten Mechanismus das *Logical monitoring* das sich auf die Entdeckung von Kontrollflussfehlern fokussiert. Hierbei wird überwacht, ob eine oder mehrere erwartete Programminstruktionen entweder in der falschen Reihenfolge oder überhaupt nicht ausgeführt werden. Dieser Mechanismus basiert auf der Arbeit von Oh et al. [OSM02]. Eine zu überwachende Funktion wird hierzu als Graph dargestellt. Zunächst wird die Funktion in Basisblöcke (*engl. basic blocks*) zerlegt, indem diese an allen (bedingten) Verzweigungen getrennt wird. Diese Basisblöcke werden dann in einem Graphen als Knoten repräsentiert und die erlaubten Zweige werden als Kanten zwischen diesen Blöcken dargestellt. Wenn zur Laufzeit ein neuer Block betreten wird, wird anhand des Graphen überprüft, ob es sich um einen erlaubten Zweig handelt.

Bis auf die Kontrollflussüberwachung eignen sich die Überwachungsmechanismen, die AUTOSAR momentan bietet, nur zur Überwachung zeitlicher Bedingungen auf einer relativ niedrigen Abstraktionsebene. Keine der hier vorgestellten Methoden wird auf der Ebene des AUTOSAR-Systemmodells konfiguriert, das dem Entwickler eine intuitive und integrierte Sicht auf das System gibt. Außer den in AUTOSAR vorgestellten Mechanismen wurde das Thema der Monitorgenerierung für AUTOSAR in der Forschung wenig behandelt.

Eine Ausnahme bilden die Arbeiten von Cotard et al. [CFB12a, CFB+12b]. In diesen Publikationen beschreiben die Autoren einen Ansatz zur Überwachung der Synchronisierung von Tasks auf Mehrkern-Hardwareplattformen. Als Grundlage werden Beziehungen zwischen Tasks als endlicher Automat (FSM) modelliert. Diese Spezifikation wird dann in Lineare Temporale Logik (LTL) übersetzt, aus der Moore-Automaten und schließlich C-Code generiert werden. Der Hauptfokus liegt dabei auf der Synchronisierung und nicht auf der Überwachung von Kontroll- oder Datenfluss.

Im Gegensatz dazu werden im MBSecMon-Prozess die zu überwachenden Signaturen als Szenarien in den erweiterten Live Sequence Charts modelliert. Hierbei liegt der Fokus auf Kommunikation zwischen den einzelnen Komponenten und nicht auf den einzelnen Komponenten selbst. Auch hier wird die Spezifikation in eine Zwischensprache – Monitor-Petrinetze – übersetzt. Die Aufteilung auf verschiedene Steuergeräte bzw. Komponenten findet erst vor der Generierung des AUTOSAR-konformen C-Codes satt.

4.5 Bewertung der existierenden Ansätze

In diesem Abschnitt wird nach der Betrachtung der existierenden Ansätze ein Vergleich zu den in Abschnitt 3.5 aufgestellten Anforderungen an eine Zwischensprache durchgeführt. Dabei wird bei Ansätzen, die nicht direkt das Ziel der Überwachung haben, ihr Potenzial bewertet.

Anforderung	Ansatz							
	DbC [BFMW01]	ES [GR98, LP99]	TL (gesamt/einzel) [KM09]	FSM (standard/komm. HFSM) [MJG+12]/[WWH08]	MOP [MJG+12]	EDL [MS05]	PN [FGP09]	CPN (CPA) [Kum95]
Ausdrucksstärke	+	+	+/+	+/+	o	+	+	+
AP1 Formale Spezifikation	-	+	+/+	+/+	+	-	+	+
AP2 Determ. Semantik	+	o	+/+	o/o	+	+	-	-
AP3 Keine Zustandsr.-Expl.	+	+	-/o	-/+	o	+	+	+
AP4 Effizient ausfürbar	+	+	o/o	+/o	o	o	+	o
AP5 Parallele Kommun.	-	+	-/-	-/-	+	+	o	o
AP6 Verschränkte Überw.	+	+	o/o	o/o	+	o	o	o
AP7 Einfache Generierung	+	-	-/-	+/-	o	-	+	-
AP8 Kompakte Repräsent.	o	+	o/o	-/o	o	o	o	o
AP9 Zeitaspekte	o	+	+/+	-/o	o	o	-	+
AP10 Aufteilbarkeit	-	-	-/+	-/+	-	-	-	+
AP11 Bedingte Überwachung	o	+	-/-	-/o	o	o	o	o
AP12 Gegenmaßnahmen	+	o	-/-	o/o	+	+	o	o
AP13 Plattformunabh.	-	-	+/+	+/+	-	o	+	o
AP14 Erl. u. verb. Verhalten	-	-	+/+	-/+	+	-	-	-
AP15 Deontische Bedingung	-	o	+/+	-/-	o	+	+	+
AP16 Ereignisklassen	-	+	-/-	o/-	-	-	-	+
AP17 Lok./glob. Variablen	-	+	-/-	+/o	-	-	-	+

+ erfüllt; o teilweise erfüllt; - nicht erfüllt

Tabelle 4.2: Bewertung existierender Ansätze mit den Anforderungen an eine Zwischensprache

Tabelle 4.2 stellt die unterschiedlichen möglichen Zwischensprachen den Anforderungen gegenüber. Die erste Spalte enthält die Anforderungen an eine Zwischensprache, die in Abschnitt 3.5 vorgestellt wurden. Für die verschiedenen Formalismen wurde jeweils ein Ansatz, der im vorherigen Teil des Kapitels vorgestellt wurde, als Repräsentant ausgewählt.

Die in den *Design by Contract*-Ansätzen (DbC) verwendeten Sprachen sind im Allgemeinen an Zielsprachen in ihrer konkreten Syntax angelehnt, um die Modellierung zu vereinfachen. Die Semantik der verwendeten Sprachen ist für die lokale Überwachung von Parametern mittels Assertions eingeschränkt und nur wenige Ansätze bieten eine Sprache an, die die Überwachung von Sequenzen erlaubt (CSP). Hierbei ist jedoch deren Anwendung durch die Ziele des DbC-Konzepts sehr eingeschränkt.

Die einfache Syntax und Semantik der Sprachen für *Expertensysteme* bietet Möglichkeiten viele verschiedene Spezifikationssprachen in ihnen zu überwachen, jedoch führt deren Auswertung zu einem relativ hohen Overhead zur Speicherung der Faktenbasis und Wechselwirkungen zwischen den einzelnen Regeln treten

leicht ein. Die Expertensysteme werden direkt von den IDS ausgeführt, was sie eher zu einer Zielsprache als eine Zwischensprache für verschiedenste Zielplattformen macht.

Temporale Logiken (TL) werden häufig zur Modellierung von einzuhaltenden Ereignissequenzen verwendet. Unter anderem finden sie ebenfalls in DbC-Ansätzen Einsatz. Zur Verifikation von Modellen werden Modelchecker genutzt, die einen großen Bedarf an Hardwareressourcen haben. Zur Laufzeitanalyse findet häufig eine Transformation in Automaten wie FSMs oder Büchi-Automaten statt, die schließlich in Code umgesetzt werden. TLs sind somit eher als Modellierungssprache und nicht als Zwischensprache zu sehen, da es bei der Übersetzung ausdrucksstarker Sprachen in TL zu Einschränkungen und zur Explosion der Größe der temporalen Formel kommt. Bei der Verwendung von nicht-deterministischen Automaten als Zwischensprache kann die Größe der Signatur im Gegensatz zu einer FSM zwar deutlich reduziert werden, jedoch benötigt dessen Ausführung eine deutliche höhere Laufzeit [SEJK08].

Finite State Machines (FSM) sind insbesondere durch ihre fehlenden parallelen Konzepte für komplexe Signaturen als Zwischensprache der Monitorgenerierung weniger geeignet. Die Ausführung einer FSM ist zwar effizient möglich, jedoch benötigen FSMs zur Abbildung paralleler Spezifikationen viel Speicherplatz. Hierarchische FSMs haben den Vorteil parallele Spezifikationen kompakt abbilden zu können, führen jedoch durch ihre relativ komplexe Semantik zu einem Mehraufwand bei der Codegenerierung und einer erhöhten Laufzeit.

Monitor-oriented Programming (MOP) ist ein Rahmenwerk, das verschiedenste Eingabeformalismen wie Kontextfreie Grammatiken, Erweiterte Reguläre Ausdrücke (ERE) und Temporale Logiken, die auf eine FSM übersetzt werden, und FSM selbst zur Beschreibung von Ereignissequenzen unterstützt. Hierbei basiert die Modellierungssprache jedoch auf aspektorientierter Programmierung und ist damit stark zielsprachenbezogen. Des Weiteren werden bei der Ausführung von Monitoren häufig Monitorinstanzen erstellt und wieder abgebaut. Durch die Objektorientierung des Ansatzes ist der Einsatz auf eingebetteten Systemen nur schwer umzusetzen. Die von MOP unterstützen Sprachen haben alle ihre Vor- und Nachteile, wie in Abschnitt 4.3.7 beschrieben wurde.

Wie in Tabelle 4.2 zu sehen ist, sind auf *Petrinetzen* basierende Sprachen am besten für den Einsatz als Zwischensprache für die modellbasierte Generierung von Laufzeitmonitoren geeignet. In vielen der betrachteten Ansätze wurde die Petrinetzsyntax um Konzepte für das Monitoring erweitert. In diesem Bereich sind besonders spezielle Start- und Endplätze hervorzuheben. Die Variante der Gefärbten Petrinetze (CPN) [Kum95] ist zwar sehr mächtig, jedoch durch die Vielzahl an komplexen Konzepten sehr aufwendig in Code für eine Zielplattform zu übersetzen. Des Weiteren eignen sich Petrinetze wie in [Fah07] gezeigt, um LTL-Formeln in eine ausführbare Form zu überführen.

In dieser Arbeit wird aus diesen Gründen eine zielsprachen- und zielplattformunabhängige Zwischensprache auf Basis der Petrinetze entwickelt, die an die speziellen Anforderungen einer Zwischensprache für die Monitorgenerierung angepasst wurde. Ein wichtiger Punkt hierbei ist, dass die Sprache so einfach wie möglich sein sollte, jedoch wichtige domänenspezifische Konzepte für die Spezi-

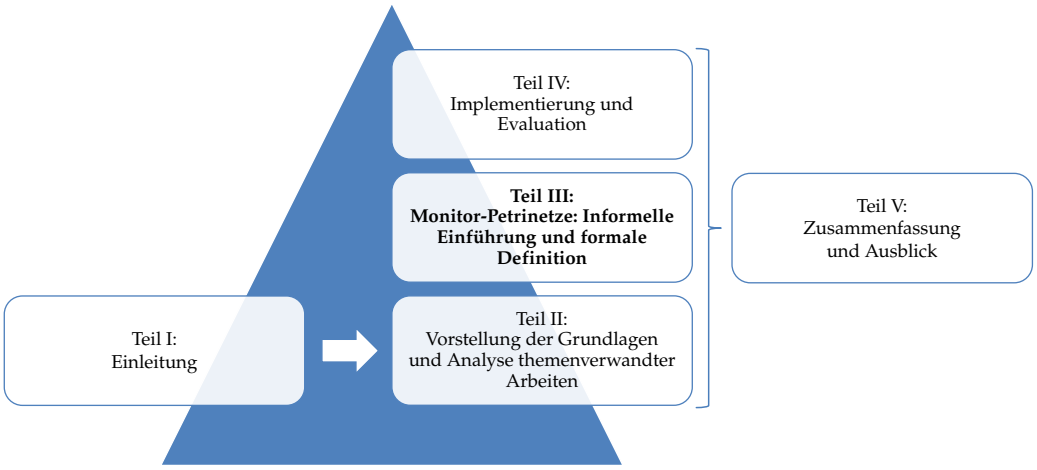
fikationen und Ausführung von Monitoren von sich aus unterstützen sollte. Dabei wird besonders auf die in Abschnitt 3.5 aufgeführten Anforderungen eingegangen.

Zusammenfassend sind dies:

- Eine ausreichende Ausdrucksstärke zur Überwachung asynchroner Kommunikationsstränge mit beliebig verschachtelten nebenläufigen Kommunikationsprozessen.
- Konzepte zur Zwischenspeicherung und Verwendung von Datenwerten in den Signaturen.
- Eine möglichst kleine Menge an Sprachkonstrukten in der Zwischensprache zur einfachen Generierung von Monitoren.
- Eine kompakte Repräsentation der Quellspezifikation und somit ein niedriger Speicherverbrauch.
- Ein möglichst geringer Laufzeitoverhead zum Einsatz auf eingebetteten Systemen.
- Eine hohe Skalierbarkeit bei Abbildung von komplexen Spezifikationen in die Zwischensprache durch Zerlegung in wiederverwendbare Teilspezifikationen, die sich gegenseitig referenzieren können.

Teil III

MONITOR-PETRINETZE: INFORMELLE EINFÜHRUNG UND FORMALE DEFINITION



5

MONITOR-PETRINETZE

Dieses Kapitel behandelt die Formalisierung der Monitor-Petrinetze (MPNs), die als neue Zwischensprache zur Generierung effizienter Laufzeitmonitore entwickelt wurde. Eine erste Formalisierung der MPN-Sprache wurde in [PPPS10] veröffentlicht.

In diesem Abschnitt wird zunächst ein Bedrohungsmodell (*engl. threat model*) aufgestellt, das das Zusammenspiel zwischen den durch die MPNs beschriebenen Signaturen darstellt. Vor der Formalisierung wird in Abschnitt 5.1 und 5.4 jeweils anhand eines Beispiels die konkrete Syntax und die Semantik der darauf folgenden Formalisierung der MPNs vorgestellt. In Abschnitt 5.2 werden die in Beispiel gezeigten grundlegenden Eigenschaften der MPN-Sprache noch einmal zusammengefasst. Die Formalisierung findet in zwei Schritten statt. Zunächst wird in Abschnitt 5.3 eine grundlegende Version der MPNs definiert und danach diese in Abschnitt 5.5 um ein Konzept zur verschränkten Überwachung einzelner Signaturen erweitert.

Ein Bedrohungsmodell ist in dieser Arbeit folgendermaßen definiert.

Definition 19 (Bedrohungsmodell). *Ein Bedrohungsmodell (engl. threat model) beschreibt eine Menge von Sicherheitsaspekten (Signaturen) und ihr Zusammenspiel untereinander. In der Gesamtheit zeigt das Modell, welche Angriffe durch den Monitor erkannt werden können.*

Abbildung 5.1 zeigt, wie eine Menge von Monitor-Petrinetzen die Sprache aller möglichen Kommunikationssequenzen (CES) eines überwachten Systems zu einer Menge von akzeptierten Ereignissequenzen (AES) einschränkt. Jede Abweichung

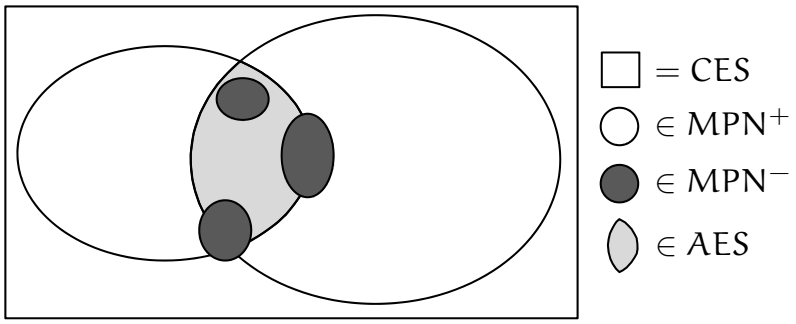


Abbildung 5.1: Bedrohungsmodell der MPN-Sprache

einer betrachteten Ereignissequenz (*engl. Trace*) von den AES wird durch wenigstens ein aktives MPN signalisiert, das entweder ein erwartetes Kommunikationsverhalten (Netze die zu MPN^+ gehören) oder einen bekannten Angriff (Netze die zu MPN^- gehören) spezifiziert. Wie in Abbildung 5.1 und Definition 20 zu sehen ist, besteht die Untermenge AES von CES aus der Schnittmenge der durch die Netze in MPN^+ erlaubten (akzeptierten) Ereignissequenzen mit Ausschluss aller Sequenzen, die durch die Netze der Menge MPN^- erkannt (akzeptiert) werden.

Definition 20 (Akzeptierte Ereignissequenzen). *Die von einem Monitor akzeptierten Ereignissequenzen (AES) sind folgendermaßen definiert. Das zu überwachende System besitzt die Ereignissequenzen CES.*

$$\mathsf{CES} = \textit{Menge aller möglichen Kommunikationssequenzen eines betrachteten verteilten Systems}$$

Die MPNs werden in positive und negative Signaturen unterteilt.

$$\mathsf{MPN} = \mathsf{MPN}^+ \uplus \mathsf{MPN}^-$$

wobei $\uplus$ die disjunkte Vereinigung beschreibt. Die Semantik $\mathcal{S}$ eines MPNs ist:

$$\mathcal{S} : \mathsf{MPN} \to \mathcal{P}(\mathsf{CES})$$

Die durch ein MPN $\mathfrak{mpn}$ akzeptierten Ereignissequenzen (AES) sind:

$$\mathcal{S}[\![\mathfrak{mpn}]\!] \subseteq \mathsf{CES}$$

Die akzeptierten Ereignissequenzen einer endlichen Menge von positiven und negativen MPNs sind:

$$\mathcal{S}^* : \mathcal{P}(\mathsf{MPN}) \times \mathcal{P}(\mathsf{MPN}) \to \mathcal{P}(\mathsf{CES})$$

$$\begin{aligned}\mathsf{AES} &= \mathcal{S}^*[\![\mathsf{MPN}^+, \mathsf{MPN}^-]\!] \\ &= \bigcap_{\mathfrak{mpn} \in \mathsf{MPN}^+} \mathcal{S}[\![\mathfrak{mpn}]\!] \setminus \bigcup_{\mathfrak{mpn} \in \mathsf{MPN}^-} \mathcal{S}[\![\mathfrak{mpn}]\!]\end{aligned}$$

mit $\mathsf{MPN}^+ \in \mathsf{MPN}$ *und* $\mathsf{MPN}^- \in \mathsf{MPN}$

Befindet sich eine Ereignissequenz nicht in der Menge der akzeptierten Ereignissequenzen, wurde ein Fehler bzw. Angriff erkannt.

Im Folgenden wird das Beispiel basierend auf Abschnitt 1.5 eingeführt, das zur Illustration der Formalisierung der Monitor-Petrinetze herangezogen wird.

5.1 Beispiel für eine Monitorspezifikation

In Abbildung 5.2 ist eine Monitorspezifikation (Signatur) abgebildet, die eine einfache „Denial of Service“-Attacke (DoS) erkennen soll. Dieses Beispiel wird im

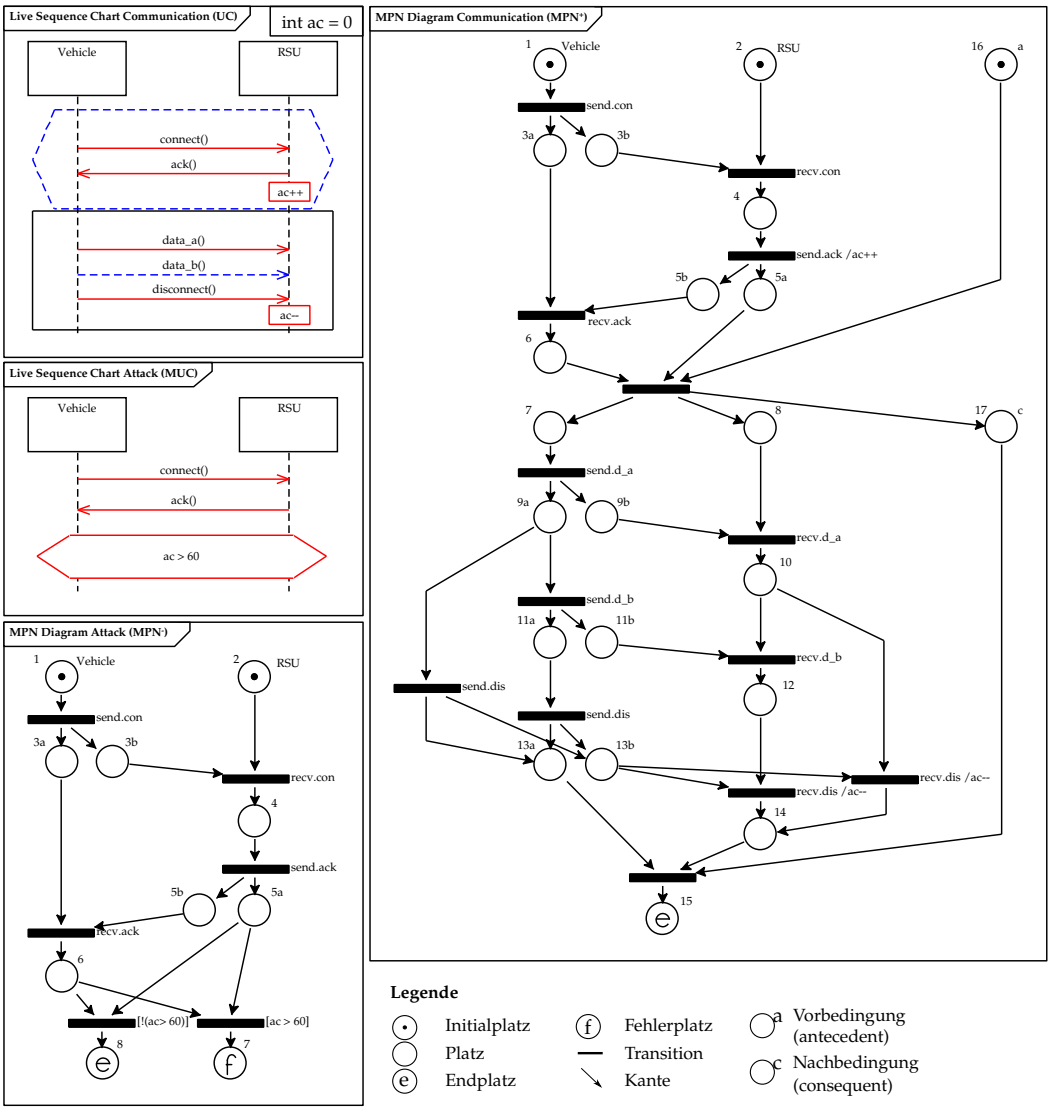


Abbildung 5.2: Erkennung einer einfachen DoS-Attacke

Folgenden, nachdem die Syntax und Semantik der MPNs beispielhaft vorgestellt wurde, zur Erläuterung der Formalisierung herangezogen.

eLSCs Auf der linken Seite von Abbildung 5.2 sind die Signaturen als erweitertes Live Sequence Charts (eLSC) dargestellt. Bei den eLSCs handelt es sich um einen Usecase *Communication*, der die gewünschte Kommunikation beschreibt, und einen dazugehörigen Misusecase *Attack*, der überprüft, ob der Angriffsfall eingetreten ist. Der Usecase beschreibt, wie sich ein Fahrzeug (Vehicle) mit der Mautbrücke (RSU) verbindet und danach Daten austauscht. Sobald die *connect*-Nachricht von der Mautbrücke mit dem Versenden einer *ack*-Nachricht angenommen wurde, wird ein globaler Zähler (ac), der die aktuell aktiven Verbindungen zählt, inkrementiert. Nach dieser Kommunikation, die in einem Prechart modelliert ist, muss das Fahrzeug eine Nachricht (*data_a*), die seine Abrechnungsda-

ten enthalten, an die Mautbrücke senden. Handelt es sich beim Mautbrückenbetreiber nicht um den eigenen Provider (Roamingbetrieb), müssen noch weitere Zahlungsinformationen (*data_b*) an die Mautbrücke geschickt werden. Zum Abschließen der Transaktion sendet das Fahrzeug eine *disconnect*-Nachricht an die Mautbrücke. Sobald die *disconnect*-Nachricht empfangen wurde, wird der Zähler für die aktiven Verbindungen wieder dekrementiert.

Dieses eLSC wird für jede neue Verbindung von außen (ein neues Fahrzeug) initialisiert. Hierdurch ist es möglich, die aktiven Verbindungen nachzuverfolgen.

Der Misusecase greift auf dieselbe globale Variable `ac`, die auch der Usecase verwendet, zu. Er hat im Gegensatz zu dem Usecase keine Vorbedingung. Nachdem sich ein neues Fahrzeug mit der Mautbrücke verbunden hat, prüft dieser, ob der Zähler für die aktiven Verbindungen größer als 60 ist. Wenn dies der Fall ist, sind mehr Verbindungen vorhanden als durch die räumlichen Gegebenheiten möglich ist und es handelt sich um eine DoS-Attacke.

MPNs Auf der rechten Seite von Abbildung 5.2 sind diese beiden Signaturen als MPN dargestellt. Das MPN *Communication* zeigt die Übersetzung des Usecase-eLSCs in ein MPN aus der Menge MPN^+. Für jede Lebenslinie im eLSC wird ein Initialplatz erstellt. Eine asynchrone Nachricht, wie die *connect*-Nachricht, wird in zwei Transitionen übersetzt. Eine für das Sende- und eine für das Empfangsereignis. Diese beiden Transitionen werden über einen Synchronisationsplatz so verbunden, dass das Sende- vor dem Empfangsereignis auftreten muss. Die übrigen beiden Nachfolgeplätze der Transitionen dienen der Sicherstellung des zeitlichen Ablaufs auf der Lebenslinie. Für die *ack*-Nachricht wird das gleiche Muster in gespiegelter Form angehängt.

Die Annotationen an den Transitionen des MPNs haben folgendes Format:

```
Ereignis[Bedingung]/Aktion
```

Alle Teile dieser Annotation sind optional und können nach Bedarf weggelassen werden. Transitionen ohne annotiertes Ereignis werden im Folgenden als *Epsilontransition* bezeichnet. Die Zuweisung (Statement) `ac++` kann als zusätzliche nachfolgende Transition modelliert werden oder wie hier geschehen als Aktion nach dem Empfang des Sendeereignisses annotiert werden. Der Übergang zwischen Pre- und Mainchart wird durch eine Epsilontransition zur Synchronisation, in der alle offenen Plätze zusammengeführt werden, dargestellt. Dahinter wird für jede Lebenslinie ein Platz angelegt, an dem wie schon im Prechart für das Mainchart fortgefahren wird.

Nachdem die Nachricht *data_a* und die optionale Nachricht *data_b* übersetzt wurden, wird auch die *disconnect*-Nachricht übersetzt und es werden zusätzlich „Übersprungstransitionen“ für die optionale Nachricht angelegt, sodass diese übersprungen werden kann. Auch hier wurde die Aktion `/ac--` an die Transition des vorherigen Ereignisses annotiert. Als Abschluss des MPNs wird, da es sich um einen Usecase handelt, auf einen Endplatz synchronisiert.

Zusätzlich zu den Plätzen, die aus der direkten Übersetzung der eLSC-Elemente entstehen, enthält das MPN zwei zusätzliche Plätze, die die Charts der eLSCs darstellen. Um nicht alle Plätze der Vorbedingung (aus Prechart entstanden) markieren zu müssen, wird der Platz, der den vorausgehenden (antecedent) Teil der

Signatur symbolisiert, als solcher markiert. Hierdurch ist eine Unterscheidung zwischen vorausgehenden Teil und nachfolgenden (consequent) Teil der Signatur bei der Auswertung des Monitorergebnisses möglich.

Das MPN *Attack* zeigt die aus dem Misusecase-eLSC entstandene MPN-Repräsentation aus der Menge MPN^{-}. Dieses eLSC besitzt kein Prechart, sondern nur ein Mainchart, das hier nicht explizit modelliert ist. Da ein Misusecase nur dann fehlschlägt, wenn ein Fehlerplatz erreicht wurde, ist es nicht notwendig, zusätzliche Plätze im MPN für die entsprechenden Charts im eLSC zu generieren. Zusätzlich zu den bekannten Übersetzungsschritten wird hier die heiße Bedingung `ac>60`, die sich über beide Instanzen erstreckt, übersetzt. An diesem Punkt findet eine Synchronisation auf einen Fehlerplatz statt, da bei erfüllter Bedingung der Misusecase entdeckt wurde. Da die heiße Bedingung sofort erfüllt sein muss, wird zusätzlich eine zweite Transition mit der invertierten Bedingung angelegt, die auf einen Endplatz führt.

AUSFÜHRUNG DES MONITORS Zur Vorbereitung auf die Formalisierung der Semantik in Abschnitt 5.3 werden hier drei Durchläufe des Monitors betrachtet. Der Erste zeigt einen positiven Durchlauf durch den Usecase *Comunication* mit einem Fahrzeug, das mit der Mautbrücke kommuniziert. Der Zweite demonstriert das Verhalten des Monitors, wenn ein Ereignis nicht der im Usecase modellierten Sequenz entspricht. Der dritte zeigt, wie der Misusecase *Attack* bei Durchfahrt mehrerer Fahrzeuge erkannt wird.

Tabelle 5.1 beschreibt die Markierung und ihre Änderung bei Stimulation mit Ereignissen des MPNs. Die erste Spalte stellt die Nummer des aktuellen *Makroschrittes* und die zweite Spalte die gerade zu verarbeitenden Ereignisse dar. Die Plätze des MPNs werden durch ihre Nummern, aus Abbildung 5.2 gekennzeichnet. In den beiden rechten Spalten befindet sich der Wert der globalen Variablen `ac` und der aktuelle Status des MPNs. Hierbei wird zwischen *laufend* (running), *beendet* (terminated) und *fehlgeschlagen* (failed) unterschieden. Die Werte in den Spalten unter den Platznummern stellen die Token, die mit einer Instanznummer der Signatur markiert sind, dar. Diese Instanznummern werden im Folgenden Eingabegenerationen genannt und dienen der Unterscheidung von Ereignissen, die verschiedenen Kommunikationssträngen (z. B. Fahrzeug/Mautbrücken-Kombinationen) zugeordnet sind. In den ersten beiden Beispielen wird hierbei nur die Kommunikation zwischen einem Fahrzeug und einer Mautbrücke betrachtet. Die Eingabegeneration ist somit 1.

Zeile 1 zeigt den Initialzustand des MPNs, in dem nur die Initialplätze mit einem Token der Eingabegeneration 1 belegt sind. Der Wert der Variablen `ac` ist auf 0 initialisiert und der Monitor ist im Status *laufend* (running). Darauf folgend wird das Ereignis *s.con* verarbeitet. Hierbei schaltet die Transition, die mit dem entsprechenden Ereignis annotiert ist, da ein Token der entsprechenden Eingabegeneration im Platz ihres Vorbereichs (1) liegt. Dies bedeutet, dass das Token in Platz 1 im Vorbereich der Transition gelöscht und in den Plätzen im Nachbereich der Transition (3a, 3b) je ein Token der Eingabegeneration erzeugt wird. Hieraus ergibt sich die in Makroschritt 2 angegebene Markierung des MPNs und das MPN ist für diese Eingabegeneration im Status *laufend* (running). Genauso

Nr.	Ereignis	Plätze																							
		vorausgehender Teil								*nachfolgender Teil*												*Chart*			
		1	2	3a	3b	4	5a	5b	6	7	8	9a	9b	10	11a	11b	12	13a	13b	14	15	16	17	*ac*	*Status*
1		1	1																			1		0	r
2	(1) s.con		1	1	1																	1		0	r
3	(1) r.con			1		1																1		0	r
4	(1) s.ack			1			1	1														1		1	r
5.1	(1) r.ack						1		1													1		1	r
5.2	-									1	1												1	1	r
6	(1) s.d_a										1	1	1										1	1	r
7	(1) r.d_a											1		1									1	1	r
8	(1) s.dis													1				1	1				1	1	r
9.1	(1) r.dis																	1		1			1	1	r
9.2	-																				1			1	t

Tabelle 5.1: Beispieldurchlauf durch den Usecase

wird mit den beiden folgenden Ereignissen verfahren, was in der Markierung in Makroschritt 3 resultiert. In Makroschritt 4 wird, nachdem die Transition geschaltet hat, die annotierte Aktion `\ac++` ausgeführt und die globale Variable damit auf 1 inkrementiert. Makroschritt 5 in dem das Ereignis *r.ack* verarbeitet wird, zeigt, wie sich ein Makroschritt in mehrere *Mikroschritte* aufteilt. Nachdem die entsprechende Transition im MPN geschaltet hat, sind die Plätze 5a, 5b, 6 und 16 belegt. Da an der nachfolgenden Transition kein Ereignis annotiert ist, handelt es sich um eine *Epsilontransition*, die schaltet, sobald die Plätze in ihrem Vorbereich belegt sind. Folgend auf die Verarbeitung eines von außen übergebenen Ereignisses, wird innerhalb desselben Makroschrittes so lange versucht Epsilontransitionen zu schalten, bis das MPN nicht mehr schaltet. Dies führt zur Markierung in Schritt 5.2, in der der Platz der die Vorbedingung markiert (16), nicht mehr belegt ist. Die Auswertung der Markierung des MPNs wird erst, nachdem das MPN nicht mehr schaltet, durchgeführt. Dieser Durchlauf wird nun basierend auf den folgenden Ereignissen weiter ausgeführt. In Schritt 9 findet nach der Verarbeitung des Ereignisses *r.dis* ein Mikroschritt statt, der dazu führt, dass der Endplatz (15) belegt wird. Dies bedeutet, dass die Signatur erfolgreich *beendet* (terminated) wurde. Alle Token der aktiven Eingabegeneration werden im Folgenden abgeräumt und das MPN neu initialisiert.

Tabelle 5.2 zeigt, wie sich die MPN-Semantik beim Auftreten von nicht im MPN modellierten Ereignissequenzen verhält. Schritte 1 bis 4 sind identisch zu der vorher betrachteten Ereignissequenz. In Schritt 5.1a wird ein nicht erwartetes (nicht akzeptiertes) Ereignis (*r.con*) dem Monitor übergeben, dass keine Transition feuern lässt. Da Platz 16, der als Vorbedingung markiert ist, zu diesem Zeitpunkt noch mit einem Token belegt ist und somit Platz 17 der den nachfolgenden Teil repräsentiert kein Token enthält, bedeutet dies, dass das MPN durch die Auswertung *beendet* (*terminated*) wird. Die Vorbedingung drückt einen Bereich in der Signatur aus, der dazu dient Zustände zu erkennen, in denen der nachfolgende (consequent) Teil der Signatur überwacht werden muss. Schritt 5.1b zeigt den al-

Nr.	Ereignis	Plätze																							
		vorausgehender Teil								*nachfolgender Teil*													*Chart*		
		1	2	3a	3b	4	5a	5b	6	7	8	9a	9b	10	11a	11b	12	13a	13b	14	15	16	17	*ac*	*Status*
1		1	1																			1		0	r
2	(1) s.con		1	1	1																	1		0	r
3	(1) r.con			1		1																1		0	r
4	(1) s.ack			1			1	1														1		1	r
5.1a	(1) r.con			1			1	1														1		1	t
5.1b	(1) r.ack						1		1													1		1	r
5.2	-									1	1												1	1	r
6	(1) s.d_a										1	1	1										1	1	r
7a	(1) r.d_b										1	1	1										1	1	f
7b	(1) r.d_a											1		1									1	1	r
8	(1) s.dis													1				1	1				1	1	r
9.1	(1) r.dis																	1		1			1	1	r
9.2	-																				1			1	e

Tabelle 5.2: Beispieldurchlauf durch den Usecase (Fehlschlagen)

ternativen Ablauf mit dem akzeptierten Ereignis. Schritt 7.1a stellt ein Ereignis dar, das wiederum nicht zu dem vom MPN beschriebenen Ablauf passt. In der Zwischenzeit ist das MPN im nachfolgenden Teil, in dem Platz 17 belegt ist, angekommen. Im Auswertungsschritt der MPN-Semantik wird hier ein Fehlerfall (*failed*) erkannt, da die Signatur in diesem Bereich des MPNs verpflichtend ist. Auch hier wird, wie bei Erreichen eines Terminalplatzes, das MPN auf die Initialmarkierung zurückversetzt. Der übrige Ablauf entspricht dem in Tabelle 5.1.

Im Gegensatz zu positiven MPNs (MPN^{+}) besitzen negative MPNs (MPN^{-}), die aus einem Misusecase entstanden sind, keine Unterscheidung zwischen Vorbedingung und Nachbereich. Damit der Misusecase eintritt, muss ein Fehlerplatz erreicht werden. Jeder Abbruch durch ein nicht schaltendes MPN führt hier zu einem Beenden der Überwachung (terminated).

Nach der beispielhaften Einführung der MPN-Semantik für einen Usecase wird im Folgenden der Misusecase *Attack* aus Abbildung 5.2 betrachtet. Mehrere Fahrzeuge haben sich in diesem Durchlauf (Tabelle 5.3) schon mit der Mautbrücke verbunden und auch teilweise wieder abgemeldet. Insgesamt sind zu Beginn dieses Ausschnitts noch 59 Fahrzeuge mit der Mautbrücke verbunden. Gezeigt werden in diesem Beispiel nur die Eingabegenerationen 80 und 81, wobei sich während dessen keine Fahrzeuge an- oder abmelden. Vor diesem Ausschnitt hat sich das Fahrzeug, dem die Eingabegeneration 80 zugewiesen wurde, schon angemeldet (*connect*-Nachricht wurde verschickt und empfangen) und ein weiteres Fahrzeug hat sich noch nicht angemeldet.

Im zweiten Schritt sendet das neue Fahrzeug ein *s.con*-Ereignis und wird der Eingabegeneration 81 zugewiesen. Das *ack*-Ereignis in Schritt 3 führt für die Kommunikation der Generation 80 dazu, dass der Zähler ac um eins auf 60 inkrementiert wird. Nach dem Senden der *r.ack*-Nachricht folgen zwei Epsilontransitionen

Nr.	Ereignis	Plätze											
		1	2	3a	3b	4	5a	5b	6	7	8	*ac*	Status
1		81	81	80		80						59	r(80,81)
2	(81) s.con		81	80, 81	81	80						59	r(80,81)
3	(80) s.ack		81	80, 81	81		80	80				60	r(80,81)
4.1	(80) r.ack		81	81	81		80		80			60	r(80,81)
4.2	-		81	81	81						80	60	t(80),r(81)
6	(81) r.con			81		81						61	r(81)
7	(81) s.ack			81			81	81				61	r(81)
8.1	(81) r.ack						81		81			61	r(81)
8.2	-									81		61	f(81)

Tabelle 5.3: Beispieldurchlauf durch den Misusecase

mit sich ausschließenden Bedingungen. Da der Zähler ac auf 60 steht, schaltet die linke Transition und das MPN wird beendet (terminated). Die Token der Generation 80 werden abgeräumt. Wenn nun Eingabegeneration 81 diese Stelle erreicht ist der Zähler in Schritt 6 schon auf 61 aktive Verbindungen erhöht worden und die rechte Transition schaltet. Hierdurch wird ein Token auf dem Fehlerplatz (7) erzeugt. Der durch den Misusecase modellierte Angriff wurde hiermit erkannt.

Wie vorher in Abbildung 5.1 gezeigt schränkt somit das gerade gezeigte negative MPN (aus MPN^{-}) die durch das positive MPN (aus MPN^{+}) erlaubten Sequenzen ein.

5.2 Zusammenfassung der Eigenschaften der MPN-Sprache

Die grundlegenden Eigenschaften der Monitor-Petrinetz-Sprache, die im vorherigen Abschnitt am Beispiel gezeigt wurden, werden in diesem Abschnitt noch einmal zusammengefasst und besprochen.

Anders als die in Abschnitt 3.2 vorgestellten Petrinetze sind die MPNs deterministisch. Dies wird unter anderem erreicht, indem im Gegensatz zu Petrinetzen in MPNs keine Konflikte existieren. Transitionen konkurrieren nicht um Token auf gemeinsamen Plätzen in ihrem Vorbereich, sondern können durch die Makroschrittsemantik, die die Aktivierung der Transitionen auf Basis des Ausgangszustandes evaluiert, gleichzeitig schalten. Durch das Aufräumen des Netzes beim Erreichen der Terminalplätze und bei Nichtschalten des Netzes kann kein Deadlock eintreten. Jeder Platz nimmt nur ein Token einer Generation auf. Plätze im Nachbereich, die schon mit einem Token der entsprechenden Generation belegt sind, verhindern nicht das Schalten der Transition und die Token werden zusammengeführt. Im Gegensatz zu Petrinetzen müssen in MPNs Transitionen, die aktiviert sind, schalten. Hierbei schalten alle Transitionen, die schalten können, innerhalb eines Makroschrittes.

Zur Unterstützung der Laufzeitüberwachung besitzen MPNs in ihrer Syntax Konstrukte, die eine implizite Auswertung des Monitorzustandes ermöglichen. Hierdurch können Vorbedingungen und Nachbereiche markiert werden und ver-

schiedene Endplätze erlauben die gezielte Erkennung positiver und negativer Enden eines MPNs. Des Weiteren können MPNs positive und negative Signaturen beschreiben, was sich auf die Auswertung der Markierung auswirkt. Spezielle Initialplätze, die Token nach festgelegten Regeln erzeugen, erlauben die Überwachung mehrerer Kommunikationsstränge in einem Netz.

Um eine Überwachung einer Signatur, die als MPN beschrieben wurde, zu ermöglichen, muss immer ein Pfad durch das MPN zu einem Terminalplatz vorhanden sein. Dies gilt insbesondere für die Modellierung negativer Signaturen, da nur auf diese Weise verbotene Ereignissequenzen erkannt werden können. Jedoch müssen dabei nur Pfade, die sich nicht implizit aus der MPN-Semantik ergeben, modelliert werden. Token dürfen, wenn ein Terminalplatz erreicht wird, noch im MPN vorhanden sein und müssen nicht vollständig durch den Endplatz abgeräumt werden. Dies ermöglicht eine kompakte Modellierung und damit eine kompakte Repräsentation im Speicher.

5.3 Grundlegende Version

Zur besseren Übersicht wird in diesem Abschnitt zunächst eine grundlegende Version der Monitor-Petrinetze eingeführt. Diese beinhaltet alle zur MPN-Sprache gehörenden Elemente und eine Semantik, die die Unterscheidung zwischen verschiedenen Kommunikationssträngen beinhaltet.

5.3.1 *Syntax der MPNs*

In diesem Abschnitt wird die abstrakte Syntax der Monitor-Petrinetzsprache, die auf Place/Transition-Netzen basiert, formal definiert. Die MPN-Sprache beinhaltet das Konzept der Eingabegenerationen, das in Abschnitt 5.1 exemplarisch zur Repräsentation von Kommunikationssträngen zwischen Mautbrücke und Fahrzeug eingesetzt wurde.

Beispiel *Eingabegenerationen*

Abbildung 5.3 zeigt, wie diese Eingabegenerationen mit der Menge der MPNs einen Raum an möglichen Markierungen aufspannt, der je nach Anzahl der aktiv überwachten Kommunikationsstränge anwächst. Hierbei wird für jede neue

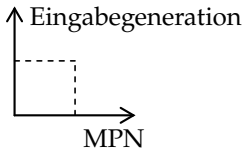


Abbildung 5.3: Eingabegenerationen in der MPN-Definition

Eingabegeneration für alle MPNs im Monitor eine neue MPN-Instanz (Eingabegeneration) erzeugt, mit der der neue Kommunikationsstrang überwacht wird.

Definition 21 (MPN-Instanz). *Eine MPN-Instanz ist ein MPN mit der Markierung einer bestimmten Eingabegeneration.*

Eine Eingabegeneration wird in der folgenden Formalisierung mit dem Oberbegriff Generation benannt. Diese Generation wird in der späteren Formalisierung erweitert.

Definition 22 (Syntax). *Ein Netz* $mpn \in$ *MPN ist eine erweiterte Variante der Place/Transition-Netze über einem gegebenen Alphabet von möglichen Eingabeereignissen (engl. input events)* $I = \{i_1, \ldots, i_{|I|}, i_\epsilon\}$ *(wobei* i_ϵ *ein Eingabeereignis mit einem Epsilonereignis ist, das spontane Transitionen feuern lässt) und einem festgelegten Set von Ausgangsereignissen (engl. output events)* $O = \{\texttt{running}, \texttt{terminated}, \texttt{failed}\}$ *mit*

$mpn = (S, T, F, E, G, I_{MPN}, m, p, a)$ *wobei*

- $S = S_i \cup S_n \cup S_t = \{s_1, s_2, \ldots, s_{|S|}\}$ *ist eine endliche Menge von Plätzen, die die Vereinigung von den disjunkten nicht leeren Mengen der*
  - *Initialplätze* S_i*, die Markierungen beinhalten und den initialen Zustand des Netzes repräsentieren.*
  - *normalen Plätze* S_n*.*
  - *Terminalplätze* S_t*, die das Ende eines MPN-Ausführungsprozesses signalisieren.*
- $S_t = S_e \cup S_f$ *sind die Terminalplätze, die reguläre End- oder Fehlerplätze sein können.*
- $S_c \subseteq S; S_a \subseteq S; S_c \cap S_a = \emptyset$ *sind Mengen von Plätzen, die entweder den vorausgehenden (antecedent) oder den nachfolgenden (consequent) Teil der Signatur symbolisieren. Sie werden zusätzlich zu den End- und Fehlerplätzen für die Unterscheidung von positiven und negativen Enden eines MPNs, wenn keine Transition durch das Eingabeereignis gefeuert wird, verwendet.*
- $T = \{t_1, t_2, \ldots, t_{|T|}\} \neq \emptyset$ *ist eine endliche Menge aller Transitionen.*
- $F \subseteq (S \setminus S_t \times T) \cup (T \times S)$ *ist die Flussrelation, die Plätze mit Transitionen verbindet.*
- $G \subseteq \mathbb{N}$ *ist eine Menge von natürlichen Zahlen, die zur Unterscheidung zwischen verschiedenen Generationen von Markierungen im Netz verwendet wird.*
- $E : G \to E$ *ist die Familie aller möglichen Umgebungen eines MPNs, die potenziell generationsspezifische Bindungen von Variablen zu Werten während der Ausführung eines MPNs speichert. Eine genauere Definition findet sich in Abschnitt 6.3.*
- $I_{MPN} \subseteq I$ *ist eine Menge von Eingabeereignissen, die von dem MPN verarbeitet werden und aus Ereignissen und einer Teilumgebung bestehen. Elemente von* $I \setminus I_{MPN}$ *werden einfach während der Ausführung des MPNs ignoriert.*

$m \in M : (S \times G) \to \{1, 0\}$ *ist die Funktion, die die Markierung des Netzes definiert. Die Funktion unterscheidet zwischen verschiedenen Eingabegenerationen (*$g \in G$*). Ein Platz kann je Generation* g *nur mit einem Token markiert sein.*

$p : T \times I_{MPN} \times E \times G \to \{\mathtt{true}, \mathtt{false}\}$ *ist ein Prädikat, das jeder Transition für eine gegebene Eingabe (engl. Input), eine Umgebung von Variablenbindungen und einer Generation einen logischen Wert zuweist.*

$a : T \times I_{MPN} \times E \times G \to E$ *sind die Aktionen, die an den Transitionen annotiert sind, die eine spezifische Eingabe (engl. Input), eine Generation und eine Umgebung als Kontext übergeben bekommen und eine neue Umgebung mit modifizierten Variablenbindungen als Ergebnis erzeugen.*

Vorbereich von $t \in T$ *:* $\bullet t = \{s \in S | (s, t) \in F\}$ *ist die Menge aller Plätze, die mittels einer Flussrelation von einem Platz zu der gegebenen Transition verbunden sind.*

Nachbereich von $t \in T$*:* $t\bullet = \{s \in S | (t, s) \in F\}$ *ist die Menge aller Plätze, die mittels einer Flussrelation von der gegebenen Transition zu einem Platz verbunden sind.*

Im Folgenden wird auf Elemente eines MPNs mit der Notation `mpn.e` oder in allen Fällen in denen nur ein MPN verarbeitet wird direkt mit `e` zugegriffen. Aus Gründen der Übersichtlichkeit und Verständlichkeit wird in den Definitionen mpn nicht explizit jeder Prozedur als Parameter übergeben. Außerdem werden die statischen Elemente eines MPNs, wie Plätze und Transitionen, direkt adressiert.

Die MPN-Sprache unterscheidet zwischen Signaturen, die erlaubtes und verbotenes Verhalten beschreiben.

Definition 23 (Positive und negative MPNs). *Ein MPN beschreibt ein positives (Usecase) oder ein negatives (Misusecase) Muster von Kommunikations-/Eingabeereignissen.*

$$MPN = MPN^{+} \uplus MPN^{-} \tag{5.1}$$

5.3.2 *Semantik der MPNs*

Zur Überwachung (Monitoring) von Ereignissequenzen wurde eine Semantik, die auf Mikro- und Makroschritten basiert, entwickelt. Das Schalten (Feuern) einer einzelnen Transition wird Mikroschritt genannt. Die Verarbeitung eines einzelnen Eingabeereignisses, das alle aktivierten Transitionen des Netzes nebenläufig schaltet und die neue Markierung auswertet, wird Makroschritt genannt.

Wenn eine neue Instanz einer Kommunikationsereignisteilsequenz (*engl. trace*) überwacht werden soll, werden Token mit einer neuen Generationsnummer auf den Initialplätzen des MPNs erzeugt und eine neue Umgebung für diese Generation wird initialisiert.

Definition 24 (Initiale Markierung). *Die initiale Markierung eines* mpn *mit* $m = m_0$ *ist folgendermaßen definiert. In allen initialen Plätzen wird ein Token erzeugt.*

$$m_0(s, g) = \begin{cases} 1 & \text{for } s \in S_i \\ 0 & \text{else.} \end{cases} ; g \in G \tag{5.2}$$

In der initialen Umgebung E_0 *werden alle Variablen, die in der MPN-Spezifikation benötigt werden, mit ihrem Standardwert initialisiert (Formale Definition folgt in Abschnitt 6.3).*

Während ein Ereignis verarbeitet wird, schalten alle aktivierten Transitionen.

Beispiel *Aktivierte Transitionen*

Abbildung 5.4 zeigt ein Beispiel in konkreter Syntax für eine aktivierte Transition. Beide Plätze in Vorbereich der Transition sind mit einem Token der Generation 1 belegt. Wenn nun das Prädikat an der Transition zu wahr ausgewertet wird, schaltet die Transition. Die in der folgenden Definition verwendeten Variablen sind an den einzelnen Elementen annotiert.

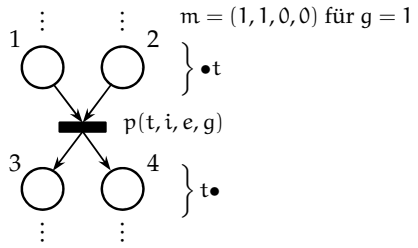


Abbildung 5.4: Beispiel einer aktivierten Transition

Definition 25 (Aktivierungsbedingung). *Eine Transition* t *eines* mpn *ist aktiviert, wenn das Prädikat* p *der Transition zu wahr ausgewertet wird und alle Plätze im Vorbereich der Transition mit einem Token der gerade verarbeiteten Generation* g *markiert sind.*

$$\texttt{enabled}(t : T, i : I_{MPN}, e : E, m : M, g : G) \rightarrow \texttt{Bool}$$
$$\texttt{enabled}(t, i, e, m, g) :\Leftrightarrow p(t, i, e, g) \wedge \forall s \in \bullet t : m(s, g) = 1$$

Eine schaltende Transition kann eine Aktion haben, die Teile der Umgebung manipuliert. Diese Änderungen müssen mit dem vorherigen Zustand der Umgebung zusammengeführt werden.

Definition 26 (Zusammenführung). *Änderungen an der Umgebung werden mit folgender Prozedur verarbeitet:*

$$\begin{aligned}&\mathtt{merge}(e : E, e_1 : E, e_2 : E) \to E\\ &e' = \mathtt{merge}(e, e_1, e_2)\end{aligned}$$

wobei $e' \in E$ das zusammengeführte Ergebnis der Änderungen e_1 und e_2 der Variablenbindungsumgebung e ist. Die Umgebungen e_1 und e_2 entsprechen den Auswirkungen zweier verschiedener Transitionen, die nebenläufig schalten. Konflikte bei Variablenzuweisungen von nebenläufig schaltenden Transitionen sollten nicht auftreten und können entweder durch Verwendung von Transitionsprioritäten gelöst werden oder als Spezifikationsfehler, der den Überwachungsprozess abbricht, behandelt werden.

Das Schalten von Transitionen ändert die Markierung des Netzes. Bei der Aktualisierung werden nur der übergebene Platz und das Token der übergebenen Generation verändert. Alle anderen Markierungen bleiben hierbei unverändert.

Definition 27 (Aktualisierung der Markierung). *Änderungen der Markierung werden folgendermaßen verarbeitet:*

$\mathtt{update}(m : M, s' : S, b : \mathtt{Bool}, g' : G, \textbf{\textit{out}}\ m' : M) :$
$\quad \forall s \in S, \forall g \in G : m'(s, g) =$
$\quad\quad \textit{if}\ s = s' \wedge g = g'\ \textit{then}\ b\ \textit{else}\ m(s, g)$

Nach der Definition der Aktualisierung der Markierung des Netzes wird im Folgenden die Verarbeitung eines Ereignisses definiert.

Definition 28 (Verarbeitung eines Ereignisses). *Alle Transitionen, die aktiviert sind (ausgewertet auf der übergebenen Umgebung e und der Markierung m), schalten, die Aktionen werden ausgeführt und es wird für die verarbeitete Generation gespeichert, ob eine Transition im Netz gefeuert hat.*

$\mathtt{processEvent}(\textbf{\textit{inout}}\ m : M, i : I_{MPN}, g : G, \textbf{\textit{inout}}\ e : E, \textbf{\textit{inout}}\ \textit{fired} : \mathtt{Bool}) :$
$\quad$ *// Kopiere Markierung und Umgebung*[1]
$\quad m' = m;\ e' = e;$
$\quad \textit{foreach}\ t \in T, \textit{with}\ \mathtt{enabled}(t, i, e, m, g)$
$\quad\quad$ *// führe Aktion aus*
$\quad\quad e'' = a(t, i, e, g);\ e' = \mathtt{merge}(e, e', e'');$
$\quad\quad \mathtt{fired} = \mathtt{true};$
$\quad\quad$ *// verarbeite Vorbereich*
$\quad\quad \textit{foreach}\ s \in {\bullet t}$
$\quad\quad\quad \mathtt{update}(m', s, 0, m', g);$
$\quad \textit{foreach}\ t \in T\ \textit{with}\ \mathtt{enabled}(t, i, e, m, g)$
$\quad\quad$ *// verarbeite Nachbereich*

[1] In der Implementierung werden nur die Änderungen der Markierung gespeichert.

```
    foreach s ∈ t•
      update(m′, s, 1, m′, g);
  m = m′; e = e′;
```

Beispiel *Determinismus des Schaltens mehrerer Transitionen*

Die in Definition 28 vorgestellte Verarbeitung eines Ereignisses stellt ein deterministisches Ergebnis sicher. Abbildung 5.5 zeigt eine Situation, die bei Betrachtung jeder Transition einzeln, wie es in Petrinetzen üblich ist, zu einem nichtdeterministischen Ergebnis führen würde. Es wird dem Netz das Ereignis *a* übergeben. Wenn wie auf der linken Seite zuerst die rechte Transition schal-

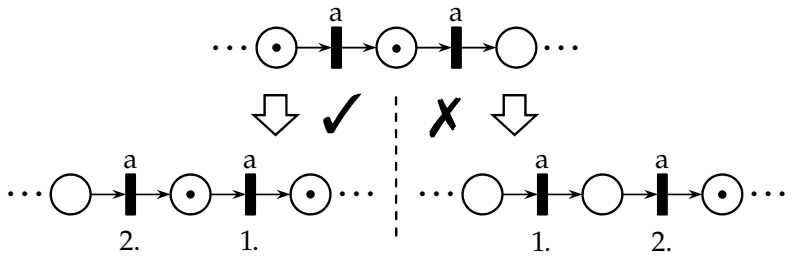


Abbildung 5.5: Problem bei gleichzeitig schaltenden Transitionen

tet, wird auf dem mittleren Platz ein Token gelöscht und auf dem rechten eines erstellt. Das Schalten der linken Transition im selben Schritt würde dann zum dargestellten Ergebnis mit zwei Token im Netz führen. Ist die Bearbeitungsreihenfolge umgekehrt, führt dies zu der auf der rechten Seite dargestellten Markierung mit nur einem Token Netz.

Durch die Erstellung einer Kopie des Netzes zur Auswertung der zu schaltenden Transitionen sowie die Aufteilung der Verarbeitung des Vorbereichs aller schaltender Transitionen (Löschen der Token) und anschließende Bearbeitung des Nachbereichs (Erstellen neuer Token), wird in einem MPN immer das linke Ergebnis erreicht.

Nachdem ein Ereignis von einem MPN verarbeitet wurde, muss die neue Markierung ausgewertet werden, um das Ergebnis des aktuellen Überwachungsschrittes zu bestimmen.

Definition 29 (Auswertung des Überwachungsergebnisses). *Diese Funktion evaluiert das Ergebnis der Ausführung des MPNs für eine Generation. Das Ergebnis ($\in O$) hängt von der Markierung der Terminalplätze ab. Falls das MPN nicht geschaltet hat, wird anhand der Belegung der als vorausgehender (antecedent) oder als nachfolgender (consequent) Teil der Signatur markierten Plätze und ob das MPN einen Use- oder Misusecase darstellt das Ergebnis bestimmt.*

```
evaluateMonitoringResult(m : M, g : G, fired : Bool) → O :
  if fired then
    // eine Transition hat geschaltet
    if ∃s ∈ S_f : m(s, g) = 1 then failed
```

```
    else if ∃s ∈ S_e : m(s, g) = 1 then terminated
    else running
  else
    // keine Transition hat geschaltet
    if mpn ∈ MPN⁻ then terminated
    else
      // Unterscheidung zwischen vorausgehendem (antecedent) und nachfolgendem
         (consequent) Teil
      if ∃s ∈ S_c|m(s, g) = 1 then failed;
      else terminated;
```

In dieser Auswertung besitzt das Fehlschlagen (*failed*) einer Signatur Priorität über das Beenden (*terminated*). Wenn in einem Fehlerplatz und in einem Endplatz am Ende der Ereignisverarbeitung ein Token liegt, gilt die Signatur als fehlgeschlagen.

Das Zurücksetzen zur initialen Markierung für eine zugeordnete Eingabegeneration ermöglicht das Beenden eines Überwachungsversuchs. Dabei wird nur der Teil der Markierung zurückgesetzt, der der übergebenen Generation entspricht.

Definition 30 (Initialisieren einer Markierung). *Die Initialisierung eines Teils der Markierung auf die initiale Markierung m_0 wird mit folgender Prozedur durchgeführt:*

```
init(m : M, g′ : G, out m′ : M) :
  ∀s ∈ S, ∀g ∈ G : m′(s, g) = if g = g′ then m_0(s, g) else m(s, g);
```

Zur Unterstützung der Synchronisation verschiedener paralleler Kontrollflüsse in einem MPN wird eine Transition ohne annotiertes Ereignis benötigt, das die Abwesenheit eines spezifischen Ereignisses symbolisiert – das *Epsilonereignis* Dieses Ereignis kann einem MPN als reguläres Ereignis mit einem zusätzlichen Kontext, der in Bedingungen und Aktionen genutzt werden kann, übergeben werden.

Definition 31 (Nachverarbeitung). *In einer zweiten Phase, nachdem das reguläre Ereignis verarbeitet wurde, schalten Epsilontransitionen, wenn ihr Kontext zu wahr ausgewertet wird, bis keine Epsilontransition mehr aktiviert ist. Wenn in einem MPN mindestens ein Token einer Generation auf einem Terminalplatz liegt oder es nicht geschaltet hat, wird die Markierung m ausgewertet und danach der Teil der Markierung, die zur entsprechenden Generation gehört wieder mit der initialen Markierung m_0 initialisiert. Die Variable* mresult *gibt den aktuellen Status des MPNs zurück.*

```
postprocessing(inout m : M, g : G, inout e : E, fired : Bool, out mresult : O) :
  epsfired = true;
  // so lange das Netz schaltet
  while (epsfired)
    processEvent(m, i_ε, g, e, epsfired);
    if epsfired then
```

```
      // setze wahr wenn Epsilontransition geschaltet hat
      fired = true;
  mresult = evaluateMonitoringResult(m, g, fired);
  if mresult ≠ running then
    // entferne Generation aus den aktiven Generationen
    G = G \ {g};
```

Letztendlich werden die vorher beschriebenen Definitionen zu einem Makroschritt zusammengefasst.

Definition 32 (Makro-Schritt-Ausführung). *Die Verarbeitung eines Eingabeereignisses und die Evaluation des Ergebnisses wird Makroschritt genannt. Nur wenn die übergebene Eingabe* i *in der Menge der akzeptierten Eingaben* I_{MPN} *enthalten ist, wird sie verarbeitet. Falls die Generation des übergebenen Ereignisses nicht schon in der Menge der aktiven Ereignisse* G *ist, wird die Markierung für die übergebene Generation* g *initialisiert. Danach wird das Ereignis verarbeitet. Die Prozedur* macro *ist definiert als:*

```
macro(inout mpn : MPN, i : I, g : ℕ, out mresult : O) :
  mpn.e = merge(mpn.e, i.e, i.e); // Die von außen übergebene Teilumgebung wird mit der
       des MPNs zusammengeführt.
  // Ist i ein akzeptiertes Eingabeereignis?
  if i ∈ I_MPN then
    if !g ∈ mpn.G then
      // aktiviere neue Generation
      mpn.G = mpn.G ∪ {g}
      init(mpn.m, g, mpn.m);
    if i == i_ε then // Epsilonereignis von außen
      fired = true;
    else
      fired = false;
    mresult = running;
    processEvent(mpn.m, i, g, mpn.e, fired);
    postprocessing(mpn.m, g, mpn.e, fired, mresult);
```

Wenn ein Epsilonereignis von außen an macro übergeben wird, transportiert dieses in der Regel einen Wert, wie zum Beispiel die regelmäßige Aktualisierung einer Variablen. Da es sich bei solchen Ereignissen nicht um klassische Ereignisse, die durch Versenden von Nachrichten entstehen, handelt, führen diese auch nicht zum Abbruch des MPNs, wenn keine Transition schaltet. Dies wird durch das Setzen von fired auf *true* erreicht. Hierdurch wird es möglich Variablen in der Umgebung eines MPNs zu aktualisieren, auf die im Nachbearbeitungsschritt dann in Bedingungen an Transitionen zugegriffen werden kann. MPNs können auch bei diesen Ereignissen als terminated oder failed ausgewertet werden, falls ein End- oder Fehlerplatz erreicht wird.

Um Parameter (Umgebungen), die durch Eingaben dem Monitor übergeben werden, auch für spätere Verarbeitungsschritte zur Verfügung zu stellen, werden diese am Anfang des Makroschrittes der Umgebung des MPNs hinzugefügt.

Hierdurch können von außen übergebene Parameterwerte im Folgenden von den MPN-Instanzen zur Auswertung verwendet werden. Die Diskussion der Gültigkeitsbereiche verschiedener Variablenarten findet in Abschnitt 6.3 statt.

Die eingeführte relevante Menge von Eingaben I_{MPN} beschränkt den in der Signatur zu modellierenden Anteil von möglichen Ereignissen. Dies ermöglicht bei einer Spezifikationssprache, wie z. B. den LSCs von Harel [HK02], die Semantik abzubilden, dass nur Nachrichten (Ereignisse), die in der Spezifikation modelliert sind, vom Monitor überwacht werden. Andererseits lässt sich auch die Semantik der eLSCs abbilden, in der alle Nachrichten, die auftreten können, auch modelliert werden müssen, falls es nicht anders durch ein Ignore-Fragment definiert ist. Dies kann erreicht werden, indem die Menge I_{MPN} alle Eingaben, die in den Signaturen des Gesamtmonitors vorkommen, beinhaltet. Durch diese Konfigurationsmöglichkeit wird eine kompakte Darstellung der Quellsignaturen als MPN erreicht, die es erlaubt kompakte Monitore zu spezifizieren und zu generieren, da die Einschränkung der zu modellierenden Ereignisse und damit die zu modellierenden Transitionen und Plätze reduziert werden können.

Ein Monitor besteht aus mehreren Signaturen, die zusammen die akzeptierten Ereignissequenzen des Monitors festlegen. Eine Ereignissequenz befindet sich in der Semantik eines MPNs, wenn die in Definition 33 beschriebene Signatur zu „true“ auswertet.

Definition 33 (Akzeptierte Ereignissequenzen eines MPNs). *Eine Ereignissequenz* $ces \in CES$ *ist eine akzeptierte Ereignissequenz (AES) des MPNs* $mpn \in MPN$ *wenn:*

$ces \in \mathcal{S}[\![mpn]\!] :\Leftrightarrow process(ces : CES, mpn : mpn, g : \mathbb{N}) \rightarrow Bool$*:*

```
while (ces ≠ ∅)
  ces = (i, ces');
  ces = ces';
   macro(mpn, i, g, mresult);
    if (mresult == failure)
     return false;
  return true;
```

wobei g *eine neue vorher nicht verwendete Generation darstellt.*

Wenn alle Instanzen der Signaturen eines Monitors sich im Zustand *running* oder *terminated* befinden, gelten sie als nicht fehlgeschlagen und die bisherige Ereignissequenz gehört zur Menge AES. Das heißt alle Präfixes eines MPNs sind, solange sie nicht durch eine andere Signatur des Monitors zu *failure* ausgewertet werden, in der Menge AES.

5.4 Beispiel für eine verschränkt ausgeführte Signatur

Im vorherigen Abschnitt wurde die grundlegende Syntax und Semantik der Monitor-Petrinetze definiert. Mit dieser ist es möglich, einfache Signaturen, die eine durchgängige, vollständige Sequenz von Ereignissen beschreiben, abzubilden. Jedoch werden diese Signaturen erst nachdem sich das MPN beendet hat bzw. fehlgeschlagen ist neu überwacht. Dadurch ist es möglich, dass Ereignissequenzen,

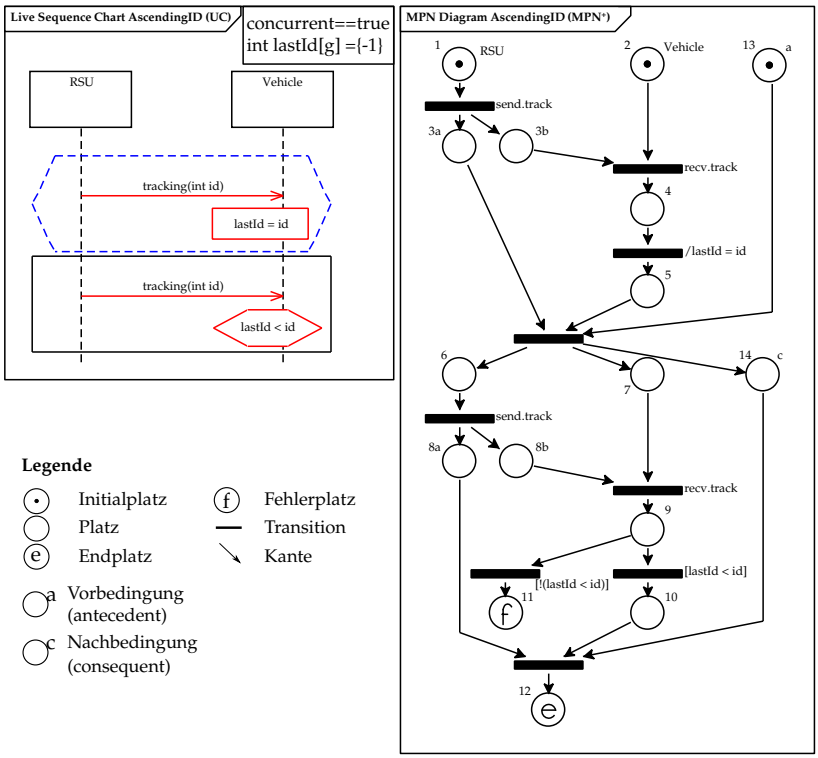


Abbildung 5.6: Beispiel eines verschränkt ausgeführten MPNs

die in einer Vorbedingung einer Signatur beschrieben sind, nicht erkannt werden, da der Monitor gerade einen anderen Teil der Signatur überwacht.

Beispiel *Einschränkung durch nicht verschränkte Überwachung*

Gegeben ist eine Ereignissequenz *e1*, *e1*, *e2*, ... Die Signatur, die auf die Teilsequenz *e1*, *e2* prüft, würde den zweiten Teil der Sequenz nicht erkennen, da erst, nachdem das zweite Ereignis *e1* aufgetreten und verarbeitet wurde, die Signatur neu initialisiert würde.

Eine solche verschränkte bzw. nebenläufige (*engl. concurrent*) Überwachung einer Signatur wird in der vorliegenden Formalisierung der MPNs nicht unterstützt.

Im folgenden Beispiel wird diese verschränkte Überwachung zur Überwachung der Konsistenz eines streng monoton ansteigenden Zählers genutzt. Die Signatur in Abbildung 5.6 ist als erweitertes Live Sequence Chart (eLSC) und als MPN modelliert und beschreibt einen Usecase. Die Signatur wird verschränkt ausgeführt. Dies bedeutet, wenn die Nachricht *tracking* mit dem Parameter id erkannt wird, wird der Wert des Parameters auf lastId zwischengespeichert. Zusätzlich wird eine neue Instanz (Subgeneration) des Musters erzeugt. Tritt diese Nachricht ein zweites Mal auf, wird zunächst die erste Instanz abgearbeitet und der neue Wert des Parameters mit dem zwischengespeicherten Wert verglichen. Danach wird diese Subgeneration ausgewertet und beendet. Darauf folgend startet die Abarbeitung der zweiten Subgeneration mit demselben Ereignis. Der neue Wert wird wieder auf lastId zwischengespeichert und eine neue Subgeneration erzeugt.

Nr.	Ereignis	Plätze																		
		vorausgehender Teil								*nachfolgender Teil*							*Chart*			
		id	1	2	3a	3b	4	5	6	7	8a	8b	9	10	11	12	13	14	*lastId*	Status
1		0	1	1													1		0	r(1)
2	(1) s.track	1	2	1,2	1	1											1,2		0	r(1,2)
3.	(1) r.track	1	2	2	1		1										1,2		0	r(1,2)
3.1	-	1	2	2	1			1									1,2		1	r(1,2)
3.2	-	1	2	2					1	1							2	1	1	r(1,2)
4	(1) s.track	2	3	2,3	2	2				1	1	1					2,3	1	1	r(1,2,3)
5	(1) r.track	2	3	3	2		2				1		1				2,3	1	1	r(1,2,3)
5.1	-	2	3	3	2			2			1			1			2,3		2	r(1,2,3)
5.2	-	2	3	3					2	2						1	3	2	2	t(1), r(2,3)
6	(1) s.track	1	4	3,4	3	3				2	2	2					3,4	2	2	r(2,3,4)
5	(1) r.track	1	4	4	3		3				2		2				3,4	2	2	r(2,3,4)
5.1	-	1	4	4	3			3			2				2		3,4		2	r(2,3,4)
5.2	-	1	4	4					3	3					2		4	3	2	f(2), r(3,4)

Tabelle 5.4: Beispieldurchlauf einer verschränkt überwachten Signatur

Tabelle 5.4 zeigt einen Beispieldurchlauf durch diese Signatur. In diesem Durchlauf werden nur Ereignisse der Eingabegeneration 1 vom Monitor verarbeitet. In der Tabelle sind im Gegensatz zum vorherigen Beispiel nur die Subgenerationen der Eingabegeneration 1 abgebildet. Bei verschränkt ausgeführten MPNs wird immer eine neue noch nicht geschaltete (passive) Instanz bereitgehalten. Schritt 2 zeigt die Markierung nach der Verarbeitung des ersten Ereignisses *s.track* mit dem Parameterwert 1. Da das Token der Subgeneration 1 aus einem der Initialplätze gelöscht wurde, wird eine neue Instanz (Subgeneration 2) angelegt. In Schritt 3 schaltet die Transition mit dem Ereignis *r.track* für die erste Subgeneration. Die gerade erzeugte Subgeneration 2 schaltet jedoch nicht, da sie das Ereignis *s.track* erwartet, und wird, da sie die passive Subgeneration ist, nicht terminiert. Anschließend schalten in den Mikroschritten für die aktive Subgeneration alle aktivierten Epsilontransitionen. Hierbei wird für Subgeneration 1 der Wert des Parameters id auf lastId gespeichert.

Beim Auftreten der nächsten *tracking*-Nachricht werden die Ereignisse zunächst von Subgeneration 1 verarbeitet. Nach dem Schalten der unteren Transition mit dem Ereignis *r.track*, die in der Markierung im Schritt 5 resultiert, wird der letzte gespeicherte Wert auf der Variablen lastId mit dem aktuellen Wert des Parameters id verglichen. Da sich die ID der Nachricht zu diesem Zeitpunkt von 1 auf 2 erhöht hat, *terminiert* die Subgeneration in einem Endplatz (12). Die später erzeugte passive Subgeneration wird, wie in Schritt 2 und 3 für Subgeneration 1 gezeigt wurde, verarbeitet. Dabei wird wie vorher eine neue passive Subgeneration (3) erzeugt. Wie gerade gezeigt, findet erst am Ende des Makroschrittes die Auswertung des Ergebnisses des MPNs statt.

Die nächste *tracking*-Nachricht hat mit 1 eine ID, die niedriger als die vorherige ist. Dies führt im nächsten Schritt, in dem das Ereignis *r.track* verarbeitet wurde, dazu, dass die Transition mit der negativen Bedingung zu wahr ausgewertet wird

und bei der Auswertung des Ergebnisses ein Token im Fehlerplatz (11) liegt. Es wurde der gesuchte Fehler gefunden und der Monitor endet mit dem Zustand *failure*.

Das gerade vorgestellte Subgenerationskonzept ist ein internes Konzept der Ausführung von MPN-Signaturen, das nicht direkt von außen beeinflusst wird. Die Subgenerationen, die mit Eingabegenerationen verknüpft sind, haben eine eigene Reihenfolge der Abarbeitung und werden MPN-intern verwaltet und ausgewertet. Eine Abbildung des Subgenerationskonzeptes auf eine Menge von Eingabegenerationen wäre grundsätzlich möglich, würde jedoch bedeuten, dass die Verwaltung und Abarbeitung von außen gesteuert werden müsste. Subgenerationen repräsentieren eine interne Verwaltung mehrerer verschränkt ausgeführter Überwachungsversuche einer Signatur und haben nichts mit der Unterscheidung verschiedener zu überwachender Kommunikationsstränge zu tun.

5.5 Erweiterte Version

Das gerade im vorherigen Abschnitt am Beispiel eingeführte Konzept der Subgenerationen wird im Folgenden genauer betrachtet und formalisiert.

Nach der Formalisierung der grundlegenden MPN-Syntax und Semantik mit der Unterscheidung zwischen verschiedenen Kommunikationssträngen (Eingabegenerationen) wird hier ein Konzept zur verschränkten Überwachung von Instanzen einer Signatur (Subgenerationen) für einen Kommunikationsstrang eingeführt. Hierfür wird die Syntax und Semantik aus Abschnitt 5.3 redefiniert, wobei nur die Unterschiede zur vorherigen Definition beschrieben werden. Das neue Konzept der Subgenerationen kann wie die bisher eingeführten Generationen für jedes einzelne MPN konfiguriert werden. Durch das Weglassen der Teile der Definition, die spezifisch für Generationen oder Subgenerationen sind, indem ihre Größe auf eins limitiert wird, kann die Definition für die entsprechende Teilsprache ohne Generationen bzw. Subgenerationen der MPNs abgeleitet werden.

Zur Abbildung solcher verschränkt überwachter Muster in die Monitor-Petrinetzsprache wird im Folgenden die MPN-Sprache um das Konzept der Subgenerationen erweitert.

5.5.1 *Syntax der MPNs*

Die Erweiterung der Syntax der MPNs bezieht sich besonders auf die Einführung einer neuen Dimension von Instanzen der MPN-Signaturen. Diese Instanzen werden durch Subgenerationen repräsentiert, die verschränkte Überwachungsversuche einer Signatur zur Laufzeit abbilden.

Beispiel *Subgenerationen*

Abbildung 5.7 zeigt die neu hinzugekommenen Subgenerationen, die einzelnen Eingabegenerationen zugeordnet werden. Hierbei ist zu beachten, dass das Konzept der Subgenerationen optional ist und sich die Anzahl der vorhandenen Subgenerationen je nach MPN und Eingabegeneration unterscheiden kann. Nur MPNs, die verschränkt überwacht werden, besitzen Subgenerationen.

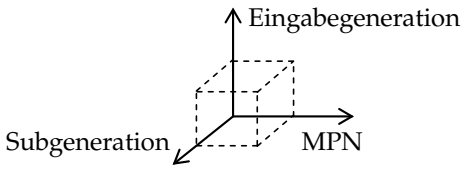


Abbildung 5.7: Subgenerationen in der MPN-Definition

Die folgende Definition erweitert die grundlegende Formalisierung damit um das Konzept der Subgenerationen. Diese Subgenerationen werden Eingabegenerationen zugeordnet. Paare bestehend aus Eingabe- und zugehöriger Subgeneration bilden im Folgenden gemeinsam eine Generation.

Definition 34 (Erweiterte Syntax). *Basierend auf Definition 22, ist ein* $mpn \in MPN$ *eine erweiterte Variante eines Place/Transition-Netzes mit* $mpn = (S, T, F, E, G, I_{MPN}, m, p, a)$ *wobei*

- $G \subseteq G_i \times G_s; G_i \subseteq \mathbb{N}; G_s \subseteq \mathbb{N}$ *ist die Menge der Generationen, die zwischen verschiedenen Eingangsgenerationen* (G_i) *und Subgenerationen* (G_s) *der Token unterscheidet. Diese wird durch Tupel natürlicher Zahlen definiert.*
- $E_G : G \to E$ *ist die Familie aller möglicher Umgebungen eines MPNs, das potenziell (sub-)generationsspezifische Bindungen von Variablen zu Werten während der Ausführung eines MPNs speichert.*
- $m \in M : (S \times G) \to \{1, 0\}$ *ist die Funktion, die die Markierung des Netzes definiert. Die Funktion unterscheidet zwischen verschiedenen Kombinationen von Eingabegenerationen und Subgenerationen* (G). *Ein Platz kann je Generation* g *nur mit einem Token markiert sein.*
- $p : T \times I \times E \times G \to \{\mathtt{true}, \mathtt{false}\}$ *ist ein Prädikat, das jeder Transition für eine gegebene Eingabe (engl. Input), eine Umgebung von Variablenbindungen und einer Generation einen logischen Wert zuweist.*
- $a : T \times I \times E \times G \to E$ *sind die Aktionen, die an den Transitionen annotiert sind, die eine spezifische Eingabe (engl. Input), eine Generation und eine Umgebung als Kontext übergeben bekommen und eine neue Umgebung mit modifizierten Variablenbindungen als Ergebnis erzeugen.*

5.5.2 *Semantik der MPNs*

Im Folgenden werden notwendige Änderungen an der Ausführungssemantik der in Abschnitt 5.3.2 eingeführten MPNs durch senkrechte Balken an der linken Seite der Definition gekennzeichnet. Für die folgenden Definitionen ist es nötig, die höchste/kleinste aktive Subgeneration für eine bestimmte Eingabegeneration zu ermitteln. Die Paare (Generationen) mit Eingabe- und Subgeneration müssen da-

zu erst nach der aktuell verarbeiteten Eingabegeneration gefiltert werden und dann aus dieser Menge die höchste/niedrigste vorhandene Subgeneration ermittelt werden.

Definition 35 (Subgenerationen zu einer Eingabegeneration). *Die existierenden Subgenerationen zu einer Eingabegeneration g_i werden durch die Funktion $\mathrm{subGen}(g_i : G_i) \to \mathcal{P}(G_s)$ bestimmt. Sie ist folgendermaßen definiert.*

$$\mathrm{subGen}(g_{cur}) = \{g_s \in G_s \mid (g_{cur}, g_s) \in G\} \tag{5.3}$$

Mithilfe dieser Definition kann die höchste Subgeneration für eine Eingabegeneration ermittelt werden.

Definition 36 (Höchste Subgeneration). *Die Funktion $\mathrm{maxSub}(g_{cur} : G_i) \to \mathbb{N}$ bestimmt das Maximum der schon vorhandenen Subgeneration einer übergebenen Eingabegeneration g_{cur} in den Paaren der vorhandenen Generationen (G).*

$$\mathrm{maxSub}(g_{cur}) = g_s \in G_{is} \mid \nexists\, (g'_s \in G_{is}) > g_s \tag{5.4}$$

mit der Menge der existierenden Subgenerationen aus der Menge der Paare mit der aktuellen Eingabegeneration g_{cur}

$$G_{is} = \mathrm{subGen}(g_{cur}) \tag{5.5}$$

Auf ähnliche Weise lässt sich auch die niedrigste Subgeneration für eine Menge an Subgenerationen bestimmen.

Definition 37 (Niedrigste Subgeneration). *Die niedrigste Subgeneration in einer Menge an Subgenerationen (G_{sub}) wird durch die Funktion $\mathrm{minSub}(G_{sub} : \mathcal{P}(G_s)) \to \mathbb{N}$ bestimmt. Sie ist folgendermaßen definiert.*

$$\mathrm{minSub}(G_{sub}) = g_s \in G_{sub} | \nexists (g'_s \in G_{sub}) > g_s \tag{5.6}$$

Die Definition der initialen Markierung (Def. 24) ändert sich durch die Einführung der Subgenerationen nur geringfügig. Es wird die um die Subgeneration erweiterte Definition der Generation verwendet. Auch die Definitionen für die Aktivierungsbedingung (Def. 25), die Aktualisierung der Markierung (Def. 27) und die Initialisierung der Markierung (Def. 30) werden bis auf die Neudefinition der Generation übernommen.

Durch die Einführung der Subgenerationen, die die verschränkte Überwachung eines MPNs (Signatur) ermöglichen, muss die Verarbeitung eines Ereignisses erweitert werden. Dabei werden alle aktiven Instanzen der Eingabegeneration des Ereignisses (Subgenerationen) eines MPNs in der Reihenfolge ihrer Erzeugung abgearbeitet. Instanzen mit einer höheren Subgenerationsnummer können dabei

auf Werte der Umgebung zugreifen, die durch Aktionen der vorherigen Instanzen verändert wurden. Hierdurch wird eine aufeinander aufbauende Verarbeitung von Werten unterstützt.

Beispiel *Subgenerationen zu Laufzeit des Monitors*

Ein Beispiel für eine solche Überwachung ist die in Abschnitt 5.4 gezeigte Signatur, die einen Wert, der mit der Eingabe übergeben wird, auf monotonen Anstieg überprüft. Abbildung 5.8 zeigt die Existenz von Subgenerationen zu einem Zeitpunkt der Monitorausführung, in dem die Überwachung der verschiedenen Eingabegeneration unterschiedlich weit fortgeschritten ist. Die Sub-

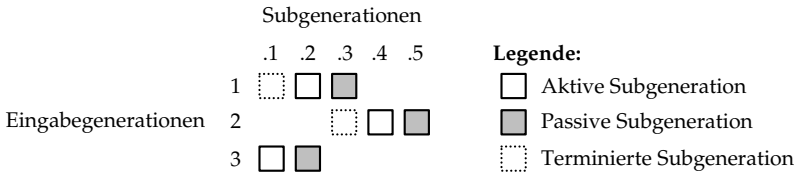


Abbildung 5.8: Aktive und passive Subgenerationen

generationen sind jeweils einer Eingabegeneration zugeordnet und haben eine Reihenfolge, die auf ihrer Erzeugungsreihenfolge basiert. Die Signaturinstanzen werden bei der Ausführung immer von der kleinsten (links) zur höchsten Subgeneration (rechts) mit dem aktuell verarbeiteten Ereignis stimuliert. Dabei ist die letzte Subgeneration immer die passive, die noch nicht geschaltet hat und auf das erste Ereignis, das in der Signatur modelliert ist, wartet. Terminierte Signaturinstanzen werden aus der Menge der Subgenerationen entfernt.

Definition 38 (Verarbeitung eines Ereignisses). *Alle Transitionen, die aktiviert sind, schalten und es wird für jede Generation gespeichert, ob irgendeine Transition im Netz gefeuert hat. Die Subgenerationen werden in der Reihenfolge ihrer Erzeugung verarbeitet, um die Änderungen der Umgebung auf die nächste Subgeneration zu übertragen. Dies ermöglicht verschränkte Überwachungsversuche einer Signatur.*

```
processEvent(inout m : M, i : I_MPN, g_i : G_i, inout e : E, out , newSubgen : Bool,
    inout fired : Bool[]) :
 newSubgen = false;
 // kopiere Markierung und Umgebung
 m' = m; e' = e;
 subGens = subGen(g_i); // hole Subgenerationen von g_i
 while (|subGens| > 0)
  g_s = minSub(subGens); // kleinste Subgeneration
  subGens = subGens \ g_s; // entferne kleinste Subgeneration
  g = (g_i, g_s);
  foreach t ∈ T with enabled(t, i, e, m, g)
   // führe Aktion der Transition aus
   e'' = a(t, i, e, g); e' = merge(e, e', e'');
   fired[g] = true;
   // verarbeite Vorbereich
   foreach s ∈ •t
```

```
    update(m′, s, 0, m′, g);
    if (s ∈ S_i ∧ g_s = maxSub(g_i)) then
     newSubgen = true
  foreach t ∈ T with enabled(t, i, e, m, g)
   // verarbeite Nachbereich
   foreach s ∈ t•
    update(m′, s, 1, m′, g);
  m = m′; e = e′; // nach jeder Subgeneration
 // erzeuge falls nötig eine neue Subgeneration
 if (newSubgen ∧ ∃s ∈ S_i| m(s, (g_i, maxSub(g_i))) = 0) then
  G = G ∪{(g_i, maxSub(g_i) + 1)};
  foreach s ∈ S_i // erzeuge Token der neuen Subgeneration
   update(m′, s, 1, m′, (g_i, maxSub(g_i)));
  m = m′; e = e′;
```

Wenn die höchste Subgeneration (passive Subgeneration) geschaltet hat, bedeutet dies, dass das Ereignis zum Anfang der Signatur passt und es wird eine neue Subgeneration erzeugt. Diese nimmt nun die Stelle der passiven Subgeneration ein. Schaltet die passive Subgeneration nicht, führt dies nicht zum Beenden und zur Neuinitialisierung, da sich diese noch im Initialzustand befindet.

Nachdem das MPN gemäß seines Zustandes und der Eingabe geschaltet hat, wird der aktuelle Zustand ausgewertet. Hierbei wird das Ergebnis für jede Subgeneration einzeln ausgewertet und der Zustand des Monitors zurückgegeben.

Definition 39 (Auswertung des Überwachungsergebnisses). *Diese Funktion evaluiert das Ergebnis der Ausführung des MPNs für eine Generation. Das Ergebnis (∈ O) hängt von der Markierung der Terminalplätze ab. Falls das MPN nicht geschaltet hat, wird anhand der Belegung der als vorausgehender (antecedent) oder als nachfolgender (consequent) Teil der Signatur markierten Plätze und ob das MPN einen Use- oder Misusecase darstellt das Ergebnis bestimmt.*

```
evaluateMonitoringResult(m : M, g : G, fired : Bool[]) → O :
 if fired[g] then
  if ∃s ∈ S_f : m(s, g) = 1 then failed
  else if ∃s ∈ S_e : m(s, g) = 1 then terminated
  else running
 else
  if mpn ∈ MPN⁻ then terminated
  else
   if ∃s ∈ S_c|m(s, g) = 1 then failed;
   else terminated;
```

Die vorgestellte Auswertung findet im Rahmen des folgenden Nachverarbeitungsschrittes statt. Auch diese Definition wird auf die Verarbeitung von Subgenerationen erweitert. Hierbei werden die Subgenerationen im Gegensatz zu den Eingabegenerationen ebenfalls von der kleinsten zur größten Subgeneration aus-

gewertet. Die eventuell bei der Verarbeitung des Eingabeereignisses neu erzeugte Subgeneration wird hierbei noch nicht verarbeitet.

Definition 40 (Nachverarbeitung). *In diesem Schritt schalten zunächst alle Epsilontransitionen, die aktiviert sind. Dies wird so lange fortgesetzt, bis keine Änderungen im Netz mehr auftreten. Wenn eine Generation eines MPNs zumindest ein Token in einem Terminalplatz hat, oder wenn das Netz im Makroschritt nicht gefeuert hat, wird die Markierung m ausgewertet und danach der Teil der Markierung, der zur aktuell verarbeiteten Generation gehört, auf die initiale Markierung m_0 zurückgesetzt.*

```
postprocessing(inout m : M, g_i : G_i, inout e : E, fired : Bool[], newSubgen : Bool,
    out mresult : O[]) :
 epsfired = true;
 subGens = subGen(g_i); // hole Subgenerationen von g_i
 // ignoriere neue Subgeneration
 if (newSubgen) subGens = subGens \ maxSub(subGen(g_i));
 while (|subGens| > 0)
  g_s = minSub(subGens); // kleinste Subgeneration
  subGens = subGens \ g_s; // entferne kleinste Subgeneration
  g = (g_i, g_s);
  while (epsfired) // so lange das Netz geschaltet hat
   processEvent(m, i_e, g, e, newSubgen, epsfired); // newSubgen wird hier ignoriert
   if (epsfired)
    fired[g] = true;
 subGens = subGen(g_i); // hole Subgenerationen von g_i
 // ignoriere neue Subgeneration
 if (newSubgen) subGens = subGens \ maxSub(subGen(g_i));
 while (|subGens| > 0)
  g_s = minSub(subGens); // kleinste Subgeneration
  subGens = subGens \ g_s; // entferne kleinste Subgeneration
  g = (g_i, g_s);
  mresult[g] = evaluateMonitoringResult(m, g, fired);
  if mresult[g] ≠ running then
   G = G \ {g}
```

In der Ausführung eines Makroschrittes wird zur Initialisierung eine Generation der entsprechenden Eingabegeneration und der Subgeneration 1 erzeugt. Zur Entscheidung, welche Subgenerationen ausgewertet werden müssen, wird der Parameter $newSubgen$ eingeführt.

Definition 41 (Ausführung eines Makroschrittes). *Für jeden übergebenen Eingang i schalten für die dazugehörige Generation alle aktivierten Transitionen. Die Prozedur $macro$ ist definiert als:*

```
macro(inout mpn : MPN, i : I, g_i : ℕ, out mresult : O[]) :
 mpn.e = merge(mpn.e, i.e, i.e);
 if i ∈ I_MPN then
  if ∄g : g = (g_i, g_s) ∈ mpn.G then
   mpn.G = mpn.G ∪ {(g_i, 1)};
```

```
  init(mpn.m, g, mpn.m);
 if i == i_ε then // Epsilonereignis von außen
  fired[] = {true}; // initialisiere Array
 else
  fired[] = {false}; // initialisiere Array
 mresult[] = {running}; // initialisiere Array
 processEvent(mpn.m, i, g_i, mpn.e, newSubgen, fired);
 postprocessing(mpn.m, g_i, i, mpn.e, fired, newSubgen, mresult);
```

Wie schon im Beispiel in Abschnitt 5.4 gezeigt wurde, kann für die Anzahl der benötigten Subgenerationen in vielen Fällen eine obere Grenze bestimmt werden.

Beispiel *Anzahl der möglichen Subgenerationen*

Die Anzahl der gleichzeitig aktiven Subgenerationen liegt in der Beispielsignatur bei drei Subgenerationen, da diese zwei Nachrichten verarbeitet und zusätzlich eine passive Subgeneration benötigt wird. In komplexeren Signaturen kann die automatische Bestimmung dieser Grenze komplizierter sein.

Für die Zuordnung einer Ereignissequenz zu den akzeptierten Ereignissequenzen der Signaturen als MPN wird Definition 33 um die Behandlung von Subgenerationen erweitert.

Definition 42 (Akzeptierte Ereignissequenzen eines MPNs). *Eine Ereignissequenz* $ces \in CES$ *ist eine akzeptierte Ereignissequenz (AES) des MPNs* $mpn \in MPN$ *wenn:*

```
ces ∈ S[[mpn]] :⇔ process(ces : CES, mpn : mpn, g_i : ℕ) → Bool:
 while (ces ≠ ∅)
  ces = (i, ces');
  ces = ces';
   macro(mpn, i, g_i, mresult);
    foreach g = (g_i, g_s) with g_s ∈ subGens(g_i)
     if (mresult[g] == failure)
      return false;
 return true;
```

wobei g_i *eine neue vorher nicht verwendete Eingabegeneration darstellt.*

Sobald eine Subgeneration mit *failed* ausgewertet wurde, gehört die übergebene ces nicht mehr zu den akzeptierten Ereignissequenzen. Somit ist ein Fehler erkannt worden.

5.6 MODELLIERUNGSRICHTLINIEN

Bei der Modellierung von MPNs sind zusätzlich zu den in der MPN-Syntax vorgegebenen Eigenschaften eines Netzes einige Richtlinien einzuhalten. Eine Missachtung führt zu fehlerhaften Signaturen oder zu Signaturen mit überflüssigen Elementen. Abbildung 5.9 illustriert einige der Richtlinien.

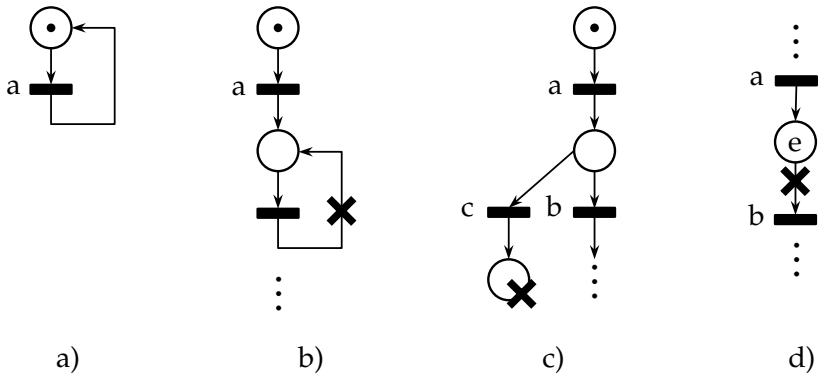


Abbildung 5.9: Beispiele für korrekte und fehlerhafte MPNs

- Ein MPN muss mindestens einen Initialplatz und eine Transition besitzen (Abb. 5.9 a).
- Epsilontransitionszyklen mit beliebiger Länge sind verboten, da sie zur Laufzeit des Monitors zu einem Lifedeadlock führen würden (Abb. 5.9 b).
- Alle Initialplätze müssen eine ausgehende Transition besitzen.
- Standardplätze müssen sowohl ein- als auch ausgehende Transitionen besitzen (Abb. 5.9 c).
- Terminalplätze müssen eine eingehende aber dürfen keine ausgehende Transition haben (Abb. 5.9 d).

Durch diese grundlegenden Richtlinien entstehen Netze, die endliche Abläufe beschreiben, die in einem Terminalplatz enden oder potenziell unendliche Abläufe, die durch Auswertung der Signaturen beim Nichtschalten eines Netzes enden können. MPN-Netze müssen nicht zwangsweise ein zusammenhängendes Netz bilden. Wenn es sich um unabhängige Teilnetze handelt, muss jedoch jedes von ihnen mindestens einen Initialplatz enthalten und eine Transition besitzen (Abb. 5.9 a). Terminalplätze sind nur dann notwendig, wenn die zu spezifizierenden Enden der Abläufe nicht schon durch die MPN-Semantik abgedeckt sind.

6

ERWEITERUNG DER MONITOR-PETRINETZE

Bisher wurden in Kapitel 5 einzelne, ganzheitlich modellierte Signaturen betrachtet. Diese Signaturen können eigenständig überwacht werden, da sie ihren Kontext (Vorbedingung), in dem sie gültig sind, selbst beschreiben. Ein Vorteil dieser vollständigen Art der Modellierung ist, dass Signaturen, die im Monitor überwacht werden sollen, einfach dem Monitor hinzugefügt und aus ihm entfernt werden können. Bei der Modellierung solcher Signaturen in einer Spezifikationssprache, wie zum Beispiel den in Abschnitt 3.1 vorgestellten erweiterten Live Sequence Charts, entstehen durch diese Vorbedingungen (Precharts) implizite Zusammenhänge zwischen den Signaturen. Das bedeutet, dass Muster, die in einer Signatur modelliert sind, eine Vorbedingung einer anderen Signatur sein können. Dies führt zu schwer wartbaren Monitorspezifikationen, da der Modellierer diese Zusammenhänge verstehen und überblicken muss.

Definition 43 (Teilsignatur). *Teilsignaturen sind Signaturen, die nur im Zusammenhang mit anderen Teilsignaturen eine vollständige Signatur im bisherigen Sinn bilden. Sie entstehen durch die Auslagerung gemeinsamer bzw. redundanter Teile, die in mehreren Signaturen vorkommen, und die Zusammenfassung identischer Vorbedingungen mehrerer Signaturen.*

Aus diesen Gründen bietet die MBSecMon-Spezifikationssprache eine übergeordnete Sprache, die MUC-Sprache aus Abschnitt 3.1.2, zur Strukturierung von Signaturen an. Diese erlaubt mehrfach in der Monitorspezifikation vorkommende Teilsignaturen auszulagern und Teilsignaturen, bedingt vom Zustand anderer Signaturen, zu überwachen.

Aus der Sicht der Monitor-Petrinetze als Zwischensprache und der Monitorgenerierung aus Signaturen in dieser Zwischensprache führen die bisher betrachteten einzelnen Signaturen zu einigen Nachteilen. Die aus diesen Signaturen generierten Monitore enthalten die Signaturen als einzelne Programmmodule, die mit Ereignissen stimuliert werden. Identische Teilsignaturen in verschiedenen Signaturen werden zur Laufzeit des Monitors mehrfach ausgewertet. Die Redundanz in den einzelnen Signaturen führt zusätzlich zu einer umfangreicheren Spezifikation und somit zu erhöhtem Speicherverbrauch auf dem Zielsystem. Das Ziel für den Einsatz der Monitore auf eingebetteten Systemen muss es deshalb sein, eine möglichst kompakte Spezifikation mit wenigen Redundanzen zu erreichen.

Zur Umsetzung dieser Ziele in der MPN-Sprache stehen zwei Möglichkeiten zur Verfügung:

- Die Zusammenführung aller in die MPN-Sprache übersetzten Teilsignaturen in eine Gesamtsignatur.
- Die Einführung eines Referenzsystems zur Beschreibung von Abhängigkeiten zwischen MPNs.

Die Zusammenführung aller Signaturen in ein Gesamtnetz ist für ausgelagerte Teilsignaturen generell möglich, würde jedoch die Komplexität der MPN-Semantik und des Generierungsprozesses erhöhen. Des Weiteren führt die Zusammenführung ausgelagerter Teilsignaturen in ein Gesamtnetz zu einer Vervielfältigung mehrfach inkludierter Teilnetze. Das Abräumen beendeter oder fehlgeschlagener Teilnetze in diesem Gesamtnetz führt zu einer erhöhten Anzahl von Transitionen im Netz, da nicht mehr wie in der MPN-Semantik definiert ganze Teilnetze verworfen werden können. Solche großen MPNs mit vielen Transitionen, die identische Ereignisse annotiert haben, führen bei der Auswertung der zu schaltenden Transitionen zu einem erhöhten Laufzeitoverhead.

Ein weiteres Ziel ist die Unterstützung rekursiver Aufrufe der Signaturen in der Zwischensprache. Dies kann durch ein Gesamtnetz in der MPN-Sprache nicht realisiert werden.

Aus diesen Gründen wird ein optionales Referenzsystem, das die Verknüpfung mehrerer MPNs untereinander unterstützt und keine Anpassung der MPN-Semantik erfordert, vorgestellt. Das in Abschnitt 6.1 am Beispiel einer Menge von Signaturen in der MBSecMon-Sprache vorgestellte Referenzsystem vermeidet Duplikationen von Teilnetzen in Signaturen und unterstützt Rekursion in MPNs, die durch das bisherige MPN-Konzept nicht ausdrückbar ist. Darauf folgend wird das vorgestellte Referenzsystem in Abschnitt 6.2 formalisiert.

Abschnitt 6.3 konkretisiert die in der MPN-Formalisierung eingeführte Umgebung zur Speicherung von Variablenwerten. Darauf folgend wird in Abschnitt 6.4 auf Basis dieses Variablenkonzepts auf die Modellierung von Zeitaspekten in MPNs eingegangen. Für die Ausführung von Gegenmaßnahmen werden im Referenzsystem ausführbare MPN-Signaturen eingesetzt, die in Abschnitt 6.5 betrachtet werden. Abschließend findet in Abschnitt 6.6 die Betrachtung der Verteilung komponentenübergreifend modellierter MPN-Signaturen auf verschiedene Teilsysteme statt.

6.1 Referenzsystem – Abhängigkeiten zwischen MPNs

Das in den bisherigen Kapiteln vorgestellte Konzept der Monitore basierend auf einzelnen MPNs führt dazu, dass jedes Eingabeereignis die Abarbeitung aller MPNs im Monitor auslöst, selbst wenn die Signatur, die das MPN beschreibt, in diesem Zustand des Systems nicht eintreten kann. Die vorgestellte Einschränkung des Makroschrittes, dass ein MPN nur auf die Ereignisse reagiert, die auch in seiner Beschreibung vorkommen, reduziert den Auswertungsaufwand. Es müssen jedoch für die verbleibenden Ereignisse alle Vorbedingungen ausgewertet werden. In einem solchen Fall würde die Vorbedingung des MPNs negativ ausgewertet und das MPN wieder zurückgesetzt, um das nächste Ereignis zu verarbeiten. Diese Art der Spezifikation führt zu einem großen Ausführungsaufwand des

Monitors. Des Weiteren müssen alle MPNs bzw. die Signaturen in der Spezifikationssprache ganzheitlich modelliert werden, indem jedes seinen vollständigen Kontext beschreibt, um Fehlerkennungen (False-Positives und False-Negatives) zu auszuschließen. Diese Art der Modellierung führt bei umfangreichen Systemen schnell zu einer unübersichtlichen Spezifikation, in der sich, insbesondere in Refaktorierungsschritten, leicht Modellierungsfehler einschleichen können. Das Anwachsen der MPN-Spezifikationen (Plätze und Transitionen) führt im resultierenden Monitor zu einem höheren Speicherverbrauch und durch die höhere Anzahl der Transitionen, die bei der Verarbeitung der Ereignisse geprüft werden müssen, zu einer höheren Laufzeit für die Ereignisverarbeitung.

Beispiel *Redundante Signaturen*

Zwei vollständig modellierte Einzelsignaturen, die ähnliche Teilabläufe beinhalten, können sich sehr ähneln. Die in Abbildung 6.1 auf der linken Seite gezeig-

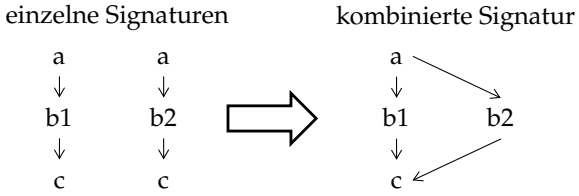


Abbildung 6.1: Abstraktes Beispiel für redundante Signaturen

ten Signaturen sind bis auf die Teilsignaturen *b1* und *b2* identisch. Wenn diese nun während der Laufzeit des Monitors abgearbeitet werden, führt dies dazu, dass Ereignisse in Teilsignatur *a* und *c* doppelt – für jede Signatur einzeln – vom Monitor verarbeitet werden. Änderungen an den Teilsignaturen *a* und *c* müssen mehrfach durchgeführt werden. Dies führt zu einer schlechteren Wartbarkeit der Monitorspezifikation. Die rechte Seite von Abbildung 6.1 zeigt, wie sich diese Nachteile umgehen lassen, indem ein Konzept zur Zusammenfassung identischer Teilsignaturen verwendet wird.

Zur Reduzierung dieses Modellierungs- und Ausführungsaufwandes, sowie des benötigten Speichers, wird in diesem Abschnitt ein Referenzsystem eingeführt, das die Kopplung von MPN-Beschreibungen untereinander ermöglicht. Ziel ist es, das Konzept möglichst unabhängig von der Spezifikationssprache (im Beispiel die MBSecMon-Sprache) zu halten und dabei alle notwendigen Konzepte zur Strukturierung von MPN-Teilsignaturen bereitzustellen.

Das Referenzsystem muss hierbei ausdrucksstark genug sein, um das Misuse-case-Konzept der MBSecMon-Sprache semantisch abzubilden. Jedoch ist es unabhängig von dem Konzept der Spezifikationssprache und zeigt allgemein ausführungsspezifische Zusammenhänge zwischen den MPNs.

Eine durch das Referenzsystem abzubildende Sprache ist die in Abschnitt 3.1.2 vorgestellte MUC-Sprache der MBSecMonSL. Diese setzt Use- (UC) und Misuse-cases (MUC) zueinander in Beziehung, sodass Teilsignaturen ausgelagert und wiederverwendet werden können.

Definition 44 (Basis-Anwendungsfall). *Als Basis-Anwendungsfall wird im Folgenden der Anwendungsfall (Use- oder Misusecase) verstanden, aus dem Teilsignaturen ausgelagert werden oder zu dem Teilsignaturen in Beziehung gesetzt werden. Bei der «extend»-, «threaten»- und «mitigate»-Beziehung ist dies der Anwendungsfall an der Pfeilspitze und bei der «include»-Beziehung der Anwendungsfall am Beginn des Pfeils.*

Folgende Beziehungen existieren in der MUC-Sprache:

EXTEND Die Signaturen werden in der parallel zum Basis-Anwendungsfall modellierten Signatur ausgeführt und stellen eine Alternative zum im Basis-Anwendungsfall modellierten Verhalten dar. Hierbei können auch Alternativen zu Teilsignaturen des Basis-Anwendungsfalls beschrieben werden.

THREATEN Diese Beziehung wird ähnlich wie die «extend»-Beziehung behandelt. Sie verknüpft negative Signaturen (Misusecases) mit einer Signatur im Basis-Anwendungsfall. Hierbei wird die negative Signatur parallel zum Basis-UC ausgeführt, stellt jedoch keine positive Alternative, sondern eine vollständige negative Signatur dar.

INCLUDE Die Teilsignatur wird an der entsprechenden Stelle in den Ablauf in der Signatur des Basis-Anwendungsfalls integriert. Hierbei werden beide Signaturen parallel ausgeführt und wie eine Signatur behandelt. Im Unterschied zu der «extend»-Beziehung ist die Teilsignatur keine Alternative zur Signatur im Basis-Anwendungsfall, sondern ein Teil der Signatur.

MITIGATE Sie verknüpft mögliche Gegenmaßnahmen, die nachfolgend (nicht parallel) ausgeführt werden, wenn der referenzierte Basis-Anwendungsfall (UC oder MUC) fehlschlägt. Dies bedeutet für einen UC, dass er fehlgeschlagen ist, und für einen MUC, dass er erkannt wurde.

Beispiel *MUC-Diagramm zur Strukturierung von Signaturen*

Abbildung 6.2 zeigt eine Strukturierung der Signaturen in der MUC-Sprache der MBSecMon-Spezifikationssprache. *CARDME Local* ist in diesem Beispiel die zentrale Signatur, die mit den anderen Signaturen in Beziehung steht. Dieser zentrale Usecase (UC) wird durch den UC *CARDME Parking* über eine «extend»-Beziehung erweitert. Ein Teil dieser Signatur ist über eine «include»-Beziehung in einen zweiten UC *Tracking* ausgelagert. Zusätzlich steht der Misusecase (MUC) *DoS Attack* durch die «threaten»-Beziehung in Verbindung mit dem zentralen UC. Wenn dieser MUC erkannt wird, wird der UC *Close Connection* ausgeführt, der über eine «mitigate»-Beziehung auf den MUC zeigt. Dieser UC *Close Connection* mildert den im MUC modellierten Angriffsfall ab.

Die eingeführten Use- und Misusecases werden durch Signaturen in Form von eLSCs beschrieben. Im folgenden Beispiel wird aus Gründen der Übersichtlichkeit nur eine Signatur pro Anwendungsfall verwendet. Im Allgemeinen können

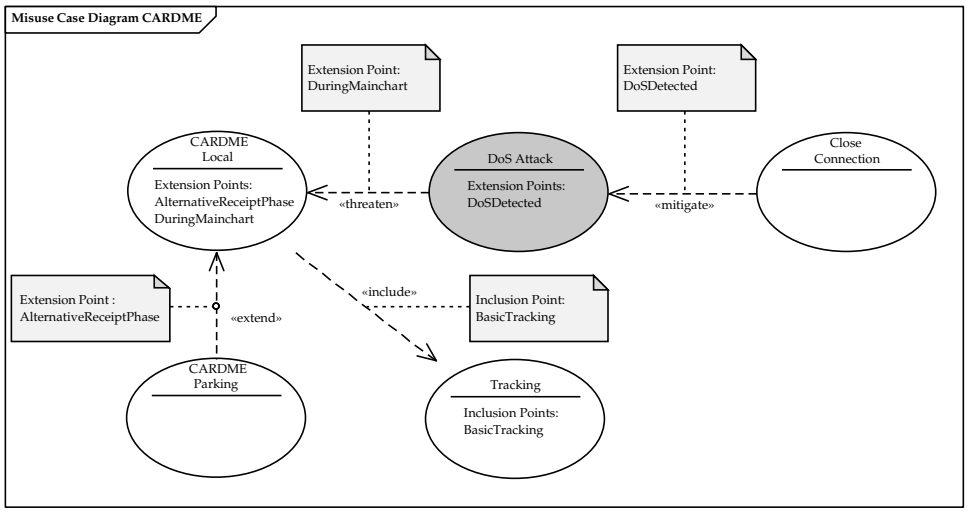


Abbildung 6.2: Beispiel für eine Misusecase-Strukturierung in der MBSecMon-Sprache

jedoch Use- und Misusecases mehrere Signaturen enthalten, in denen z. B. mehrere Alternativen beschrieben werden.

Beispiel *Signaturen des MUC-Diagramms*

Abbildung 6.3 zeigt die zugehörigen Signaturen als eLSC der Use- und Misusecases in Abbildung 6.2. Die Benennung basiert hierbei auf den im Misusecase-Diagramm verwendeten Namen, die um den Zusatz *BasicPath* erweitert wurden.

CARDME Local Diese Signatur zeigt den für dieses Beispiel vereinfachten Ablauf des CARDME-Protokolls. Als Vorbedingung werden zwei Initialisierungsnachrichten erwartet, in denen ein erster Kontakt zwischen Roadside-Unit (RSU) und dem Fahrzeug (Vehicle) mit dem Austausch der unterstützen Protokolle erfolgt. Handelt es sich bei dem gewählten Protokoll um das CARDME-Protokoll, beschreibt diese Signatur den danach folgenden Ablauf der Kommunikation. Hierbei werden zunächst die beiden Nachrichten der Präsentationsphase ausgetauscht. Darauf folgend wird eine Quittung (Receipt) von der RSU an das Fahrzeug gesendet und der Empfang vom Fahrzeug bestätigt. Solange sich das Fahrzeug im Bereich der RSU befindet, findet eine Nachverfolgung (Tracking) statt, die mit dem *ClosingRequest* beendet wird. Während der Monitor das Mainchart der Signatur überwacht, werden weitere Initialisierungsnachrichten, die in den Ignore-Fragmenten modelliert sind, für diese Signatur ignoriert.

CARDME Parking *CARDME Parking* erweitert die Basissignatur *CARDME Local* am Erweiterungsfragment (Subchart) der Quittungsphase. Im MUC-Diagramm (Abb. 6.2) wird diese Signatur über die «extend»-Beziehung eingebunden. Diese referenziert das im eLSC durch ein Notizelement markierte Subchart, zu dem es eine Alternative darstellt. Anstelle der *ReceiptRequest*- und *ReceiptResponse*-Nachrichten können auch die für das Parkraummanagement spezialisierten Nachrichten, die im LSC *Parking BasicPath* modelliert sind, auftreten.

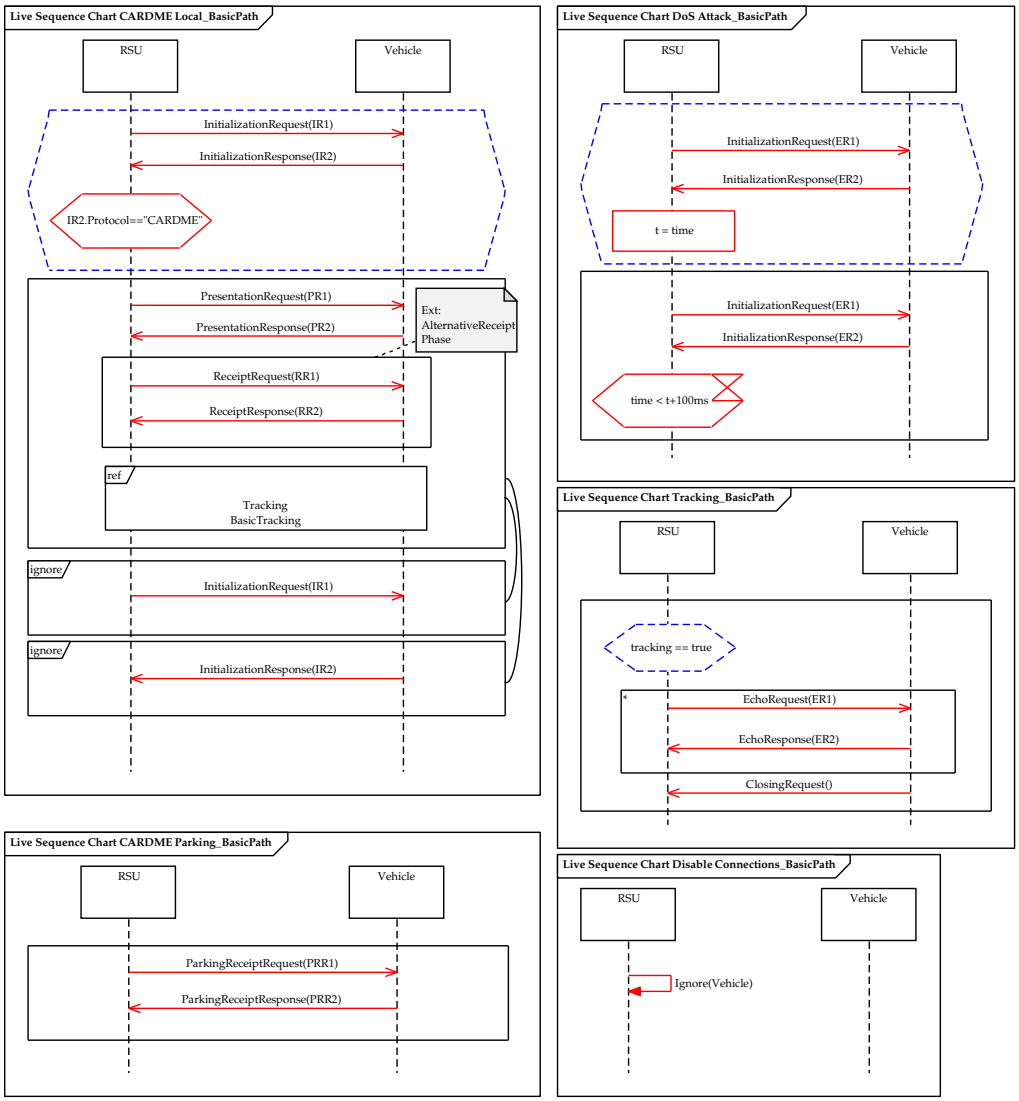


Abbildung 6.3: Signaturen des CARDME-Szenarios als eLSCs

Tracking Diese Signatur wird im Beispiel durch eine «include»-Beziehung in den UC *CARDME Local* integriert. Im eLSC *CARDME Local* ist dies als Referenzfragment modelliert, der auf einen Inklusionspunkt mit dem Namen *Basic Tracking* zeigt. Hierdurch wird die Teilsignatur an der Stelle des Referenzfragments erwartet. Die inkludierte Signatur selbst prüft zuerst, ob die Nachverfolgung von Fahrzeugen (Tracking) aktiviert ist und falls ja, werden beliebig viele Paare von Tracking-Nachrichten erwartet, die final durch ein *ClosingRequest* abgeschlossen werden.

DoS Attack Nachdem das Fahrzeug die Initialisierungsphase abgeschlossen hat, beginnt der Hauptteil (Mainchart) der eLSC-Signatur, in der die Kommunikationspartner Daten austauschen. In der Signatur *CARDME Local* werden ab diesen Zeitpunkt Initialisierungsnachrichten ignoriert. Das System selbst verarbeitet diese jedoch, wodurch eine Überlastung des Systems auftreten kann. Um einen darauf basierenden Angriff zu verhindern, wird mit der Signatur *DoS Attack* die Frequenz der eintreffenden Initialisierungsnachrichten in dieser

Zeit überwacht. Finden die Kommunikation häufiger als einmal in 100 ms statt, wird eine potenzielle Attacke erkannt.

Close Connection *Close Connection* ist im MUC-Diagramm mit dem Misusecase *DoS Attack* über eine «mitigate»-Beziehung verbunden und wird aktiv ausgeführt, falls der Misusecase entdeckt wird. Hierdurch werden die Auswirkungen des im Misusecases beschriebenen Fehlverhaltens oder Angriffs abgemildert. In diesem Fall besteht das eLSC nur aus einem synchronen Methodenaufruf *ignore* mit dem Identifikator des Fahrzeugs als Parameter.

Nach der Vorstellung der im Referenzsystem abzubildenden Beziehungen anhand der Beispielsignaturen in der MBSecMon-Sprache, wird nun vor der Einführung der formalen Definition gezeigt, wie diese Spezifikation in das MPN-Referenzsystem übertragen werden kann.

Beispiel *Übersetzung in die MPN-Sprache*

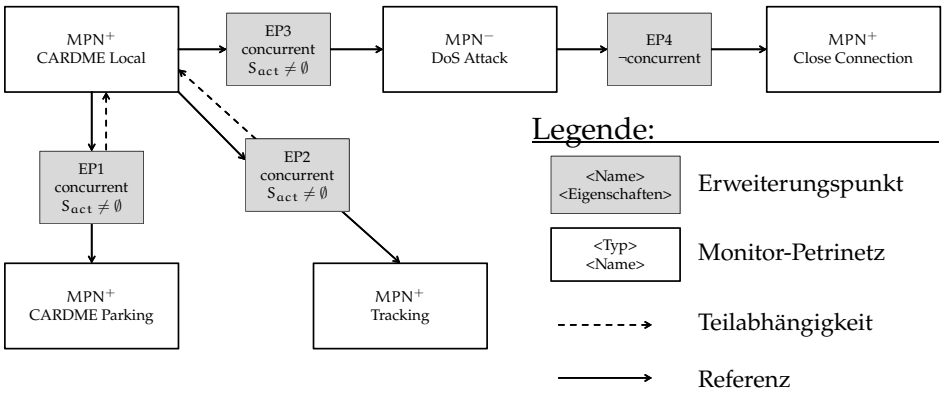


Abbildung 6.4: Zusammenhänge zwischen MPNs im Referenzsystem

Abbildung 6.4 zeigt wie die in Abbildung 6.2 gezeigte Strukturierung der Signaturen in der MBSecMonSL in das Referenzsystem der MPNs übersetzt wird. Hierbei findet die Unterscheidung zwischen den verschiedenen Beziehungen im MUC-Diagramm durch die Beschreibung ihres konkreten Verhaltens statt. Ein Erweiterungspunkt hat grundlegend folgende Eigenschaften:

- Eine Basis-Signatur, die über eine Referenz mit dem Erweiterungspunkt verbunden ist.
- Eine oder mehrere über Referenzen mit dem Erweiterungspunkt verbundene Teilsignaturen.
- Teilabhängigkeiten, die spezifizieren welche MPNs, bei einer Kaskadierung von MPNs über Erweiterungspunkte, bei der Auswertung des Erweiterungspunkts mit einbezogen werden müssen.
- Ein Attribut *concurrent*, das beschreibt, wie die referenzierten MPNs im Verhältnis zum Basis-MPN ausgeführt werden.
- Eine Menge von Plätzen im Basis-MPN (S_{act}), die Einfluss auf die Ausführung der referenzierten MPNs haben.

- Eine generationsspezifische boolesche Variable, die zum Blockieren eines Teilablaufs im Basis-MPN zur Laufzeit genutzt wird.

So wird die «threaten»- bzw. «extend»-Beziehung als Referenz mit einem nebenläufigen (*engl. concurrent*) Erweiterungspunkt modelliert. Die Unterscheidung dieser beiden Erweiterungspunkttypen kann durch das Ziel der Referenz bestimmt werden. Wie in Definition 23 festgelegt, zeigt MPN^{+} dabei, dass es sich um ein erlaubtes Verhalten und MPN^{-}, dass es sich um einen Fehlerfall handelt. Eine «include»-Beziehung wird durch einen Erweiterungspunkt, der ebenfalls nebenläufig ausgeführt wird, repräsentiert. Teile der referenzierenden MPN-Instanz können dabei während der Überwachung der referenzierten Teilsignatur angehalten werden. Hierbei verhindert eine Bedingung an Transitionen im MPN, dass der entsprechende Teil des Basis-MPNs weiter schalten kann. Bei allen erwähnten Beziehungen werden dem Erweiterungspunkt Aktivierungsplätze im Basis-MPN zugewiesen, die die Überwachung der referenzierten Teilsignaturen steuern. Abschließend wird eine «mitigate»-Beziehung als nicht nebenläufig modelliert, wobei die Menge an Aktivierungsplätzen S_{act} auch leer sein kann. Dies führt zu einer Ausführung des referenzierten MPNs nach Fehlschlagen (*failed*) des Basis-MPNs. Durch Teilabhängigkeiten werden bei den Erweiterungspunkten EP1 und EP2 angezeigt, dass es sich bei dem referenzierten MPN um eine Teilsignatur handelt, die zusammen mit *CARDME Local* ausgewertet werden muss.

In Abbildung 6.5 und 6.6 werden die im Beispiel gezeigten Signaturen als MPN dargestellt. Die zu den Quellsignaturen in Abbildung 6.3 zusätzliche Signatur *CARDME Local_Alt_EP1* in Abbildung 6.6 entsteht bei der Übersetzung in die MPN-Sprache durch die Auslagerung der mit einer Alternative (*Parking*) verbundenen Teilsignatur aus *CARDME Local*. In einem solchen Fall werden Teilsignaturen aus dem Basis-MPN in ein einzelnes MPN, das mit demselben Erweiterungspunkt, wie die anderen Alternativen, verbunden ist, extrahiert.

Die Aktivierung der MPN-Instanzen der referenzierten Teilsignaturen zur Laufzeit werden durch die Belegung der als Aktivierungsplätze markierten Plätze im Basis-MPN gesteuert. Abbildung 6.5 zeigt diese Platzmengen in Verbindung mit den zugehörigen Erweiterungspunkten. Hierbei kennzeichnen die zu den Erweiterungspunkten gehörende Menge der Aktivierungsplätze (S_{act}) Plätze, von denen einer mit einem Token belegt sein muss, damit der Erweiterungspunkt und damit die referenzierten MPNs aktiviert werden. Solche Punkte existieren im Beispiel für die Erweiterungspunkte EP1 bis EP3. Für EP4, der durch die Übersetzung einer «mitigate»-Beziehung entstanden ist, ist die Platzmenge S_{act} leer. Der Erweiterungspunkt EP4, der zusätzlich als nicht nebenläufig markiert ist, wird erst nach dem Fehlschlagen seines Basis-MPNs aktiviert. Eine nicht-leere Menge an Aktivierungsplätzen würde in diesem Fall als Vorbedingung für die Ausführung des referenzierten UC interpretiert werden. Hierdurch ist es möglich, Alternativen für Gegenmaßnahmen abhängig vom Zustand des Basis-MPNs zu spezifizieren. Die aus der «include»- und «extend»-Beziehung entstandenen Erweiterungspunkte EP1 und EP2 haben zusätzlich noch eine boolesche, generationsspezifische Variable *EP1.end*

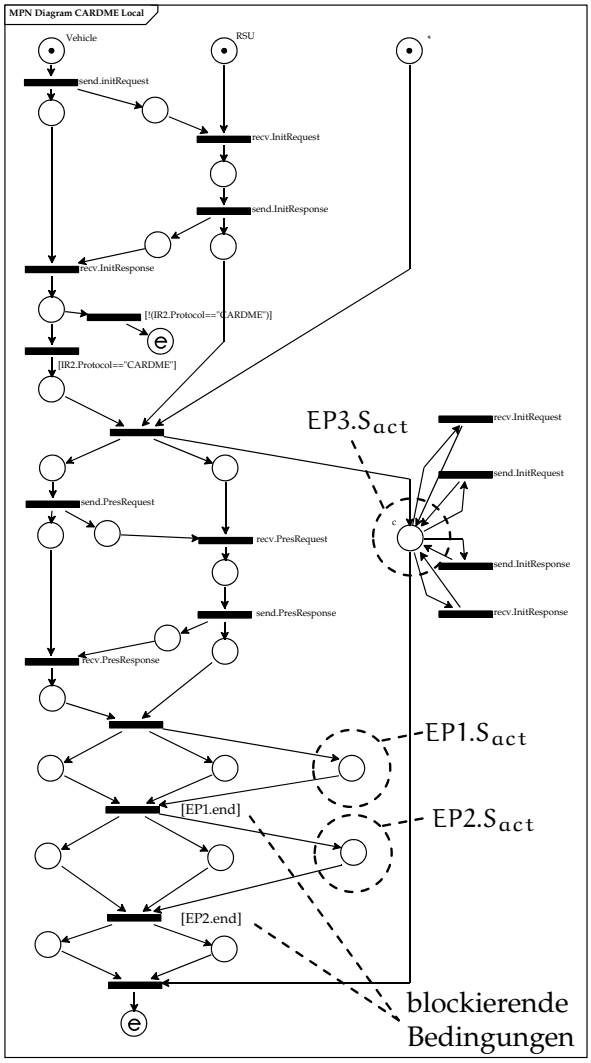


Abbildung 6.5: Signaturen des CARDME-Szenarios als MPN (1/2)

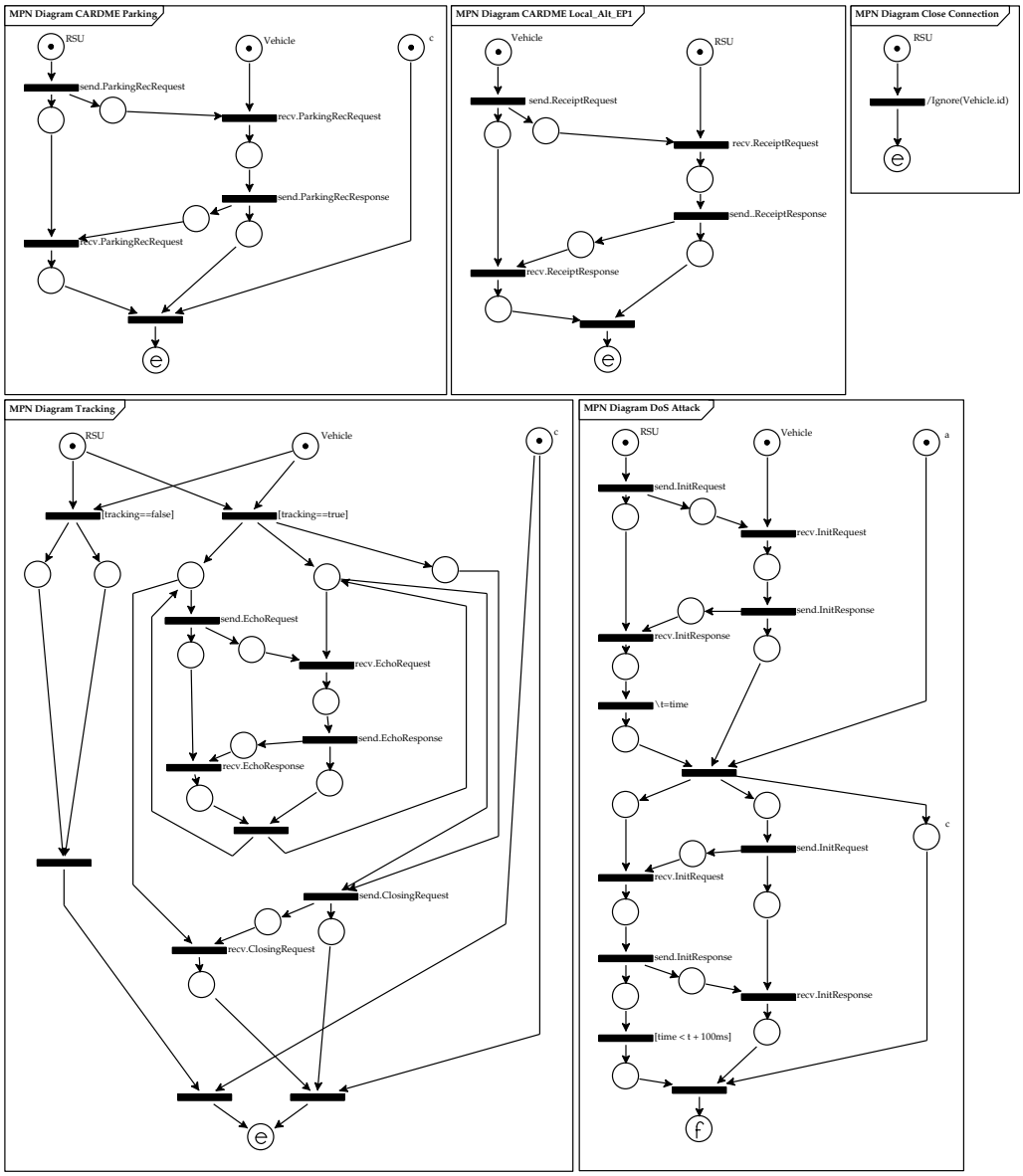


Abbildung 6.6: Signaturen des CARDME-Szenarios als MPN (2/2)

bzw. *EP2.end*, die im Basis-MPN *CARDME Local* zur zeitweisen Blockierung der Signatur eingesetzt werden.

In Tabelle 6.1 und 6.2 sind zwei Beispieldurchläufe unter Beachtung der durch das Referenzsystem vorgegebenen Abhängigkeiten für eine Eingabegeneration gezeigt. Hierbei sind Ereignisse in Form von Nachrichten zusammengefasst und die Belegung der MPNs nicht explizit dargestellt, um einen besseren Überblick über die neuen Konzepte des Referenzsystems zu geben. Die ersten beiden Spalten beschreiben die Nummer der zusammengefassten Schritte und die abgekürzten Namen der versendeten Nachrichten. Dahinter folgen die einzelnen MPNs mit Kennzeichnung, ob ihre Instanz freigeschaltet oder blockiert wird. Das Basis-MPN *CARDME Local* ist in der Tabelle nicht enthalten, da es durchgängig aktiviert ist. In den rechten vier Spalten sind die Erweite-

Nr.	Ereignis	MPN				Erweiterungspunkt			
		Parking/ Local_Alt_EP1	*Tracking*	*DoS*	*Close*	*EP1*	*EP2*	*EP3*	*EP4*
1.	IReq, IResp			act				$S_{act} = 1$	
2.	PReq, PResp	act				`false` $S_{act} = 1$			
3.	RReq, RResp	deact	act			`true` $S_{act} = 0$	`false` $S_{act} = 1$		
4.	...		deact	deact			`true` $S_{act} = 0$	$S_{act} = 0$	

Tabelle 6.1: Positiver Beispieldurchlauf durch die Spezifikation mit Referenzsystem

rungspunkte dargestellt und notiert, ob ihre Platzmengen belegt sind. Hierbei steht $S_{act} = 1$ dafür, dass mindestens ein Platz in der Menge mit einem Token belegt ist und $S_{act} = 0$, dass kein Platz mit einem Token belegt ist. Die Logikwerte `true` und `false` stehen für den Wert der erweiterungspunktspezifischen Variablen (*EP1.end* und *EP2.end*), die im MPN *CARDME Local* zum Blockieren und Freischalten von Transitionen genutzt werden.

Der positive Durchlauf in Tabelle 6.1 beginnt mit einer Folge von vier Ereignissen der Nachrichten *InitialisationRequest* (IReq) und *InitialisationResponse* (IResp). Nach der Verarbeitung des letzten Ereignisses (recv.InitResponse) wird der Platz im MPN *CARDME Local*, der sich in der Aktivierungsplatzmenge des Erweiterungspunktes 3 (EP3) befindet, belegt. Dies löst die Aktivierung des MPNs *DoS Attack* aus, das ab dem nächsten verarbeitenden Ereignis die Überwachung beginnt. Bis zu diesem Zeitpunkt musste nur das MPN *CARDME Local* die auftretenden Ereignisse verarbeiten. Nach der Verarbeitung der nächsten Ereignisse der beiden Nachrichten der Präsentationsphase in Schritt 2 ist ein Aktivierungsplatz des EP1 belegt. An diesem Punkt wird parallel zu der in das MPN ausgelagerten Alternative *CARDME Local_Alt_EP1* die zweite Alternative in der Belegphase ermöglicht, indem die beiden MPNs *CARDME Parking* und *CARDME Local_Alt_EP1* aktiviert werden. Die Variable *EP1.end* des EP1 wird auf `false` initialisiert und blockiert somit das Basis-MPN *CARDME Local* über diese Variable. Die Verarbeitung der nächsten beiden Nachrichten in Schritt 3, die dem ausgelagerten Teilablauf aus dem MPN *CARDME Local_Alt_EP1* entsprechen, führt dazu, dass der Aktivierungsplatz der Alternative (EP1) verlassen wird. Dies wird durch das Setzen der Variablen *EP1.end* des EP1 auf `true` erreicht und damit durch Übergabe eines Epsilonereignisses an das Basis-MPN der Aktivierungsplatz des EP2 belegt. Das MPN *CARDME Parking* wird infolgedessen beendet und das zusammen mit *CARDME Local* überwachte MPN *Tracking* aktiviert. *CARDME Local* wird im Folgenden, durch eine Bedingung *[EP2.end]* blockiert, bis die Signatur *Tracking* positiv erkannt wurde. Nachdem eine passende Abfolge von Ereignissen verarbeitet wurde, werden die aktivierten MPN-Instanzen *Tracking* und *DoS* wieder deaktiviert und das Basis-MPN *CARDME Local* beendet sich positiv (*terminated*).

Nr.	Ereignis	MPN				Erweiterungspunkt			
		Parking/ Local_Alt_EP1	*Tracking*	*DoS*	*Close*	*EP1*	*EP2*	*EP3*	*EP4*
1.	IReq, IResp			act				$S_{act} = 1$	
2.	PReq, PResp	act				`false` $S_{act} = 1$			
3.	PRReq, PRResp	deact	act			`true` $S_{act} = 0$	`false` $S_{act} = 1$		
4.	IReq, IResp		deact						
5.	IReq, IResp ($< 100\,ms$)				act		`true` $S_{act} = 0$	$S_{act} = 0$	

Tabelle 6.2: Negativer Beispieldurchlauf durch die Spezifikation mit Referenzsystem

Tabelle 6.2 zeigt einen Durchlauf, der zur Erkennung der DoS-Attacke führt. Schritt 1 und 2 sind dabei identisch zu denen im vorherigen Beispielablauf. In Schritt 3 werden im Gegensatz zum vorherigen Ablauf die alternativen Ereignisse der Nachrichten *ParkingReceiptRequest* (PRReq) und *ParkingReceiptResponse* (PRResp) des MPNs *CARDME Parking* dem Monitor übergeben. Nachdem diese durch das über EP1 referenzierte MPN (*CARDME Parking*) erkannt wurden, beenden sich diese (*terminated*) und seine Alternative *CARDME Local_Alt_EP1*. Darauf folgend wird das MPN *CARDME Local* durch das Freischalten der Transition mit der Bedingung *[EP1.end]* auf die Plätze hinter dem durch die alternative Sequenz abgedeckten Bereich geschaltet. Die MPN-Instanz von *CARDME Parking* wird durch das Verlassen des Aktivierungsplatzes von EP1 wieder deaktiviert. Durch die Belegung des Aktivierungsplatzes von EP2 wird eine Instanz von *Tracking* aktiviert. Folgend auf diesen Schritt werden in Schritt 4 und 5 die Ereignisse für zwei aufeinanderfolgende Initialisierungsnachrichten mit einem zeitlichen Abstand, der unter 100 ms liegt, gesendet. Hierdurch wird der die MPN-Instanz des negativen MPNs *DoS Attack* mit dem Ergebnis *failed* beendet. Das als nicht nebenläufig spezifizierte positive MPN *Close Connection* wird hierdurch aktiviert und mit einem Epsilonereignis angesprochen, damit die dort modellierte Gegenmaßnahme nicht bis zur nächsten Eingabe verzögert wird.

In diesem Beispiel wurde bisher nur eine Eingabegeneration betrachtet. In einem komplexen Monitor mit der Verwendung von Eingabe- und Subgenerationen müssen auch diese beachtet und die MPN-Instanzen, sowie ihre Freischaltung für einzelne Generationen (Eingabe- bzw. Subgenerationen) verwaltet werden.

Zur Abbildung eines solcher Beziehungen der Spezifikationssprache werden die folgenden Erweiterungen an der MPN-Sprache vorgenommen. Dabei muss die Möglichkeit geschaffen werden, MPN-Instanzen vom Zustand anderer MPN-Instanzen abhängig zu aktivieren und zu deaktivieren. Im deaktivierten Zustand verarbeiten diese keine Ereignisse. Zusätzlich müssen die Instanzen der referenzierten MPNs zurückgesetzt werden, falls sie vor ihrer Beendigung durch das Verlassen der Aktivierungsplätze im Basis-MPN deaktiviert werden. Beim parallelen

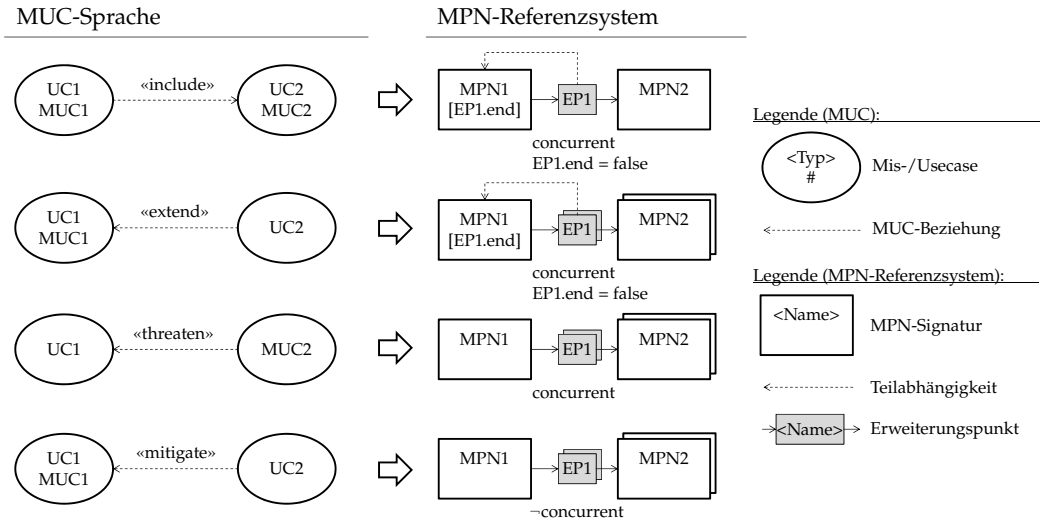


Abbildung 6.7: Übersetzung der MUC-Sprache in das Referenzsystem

MUC-Sprache	EP-Konfig.	Ref. MPNs	Ausführung	Domänen
«include»	concurrent	1	abhängig	MPN^+/MPN^- $\leftrightarrow MPN^+/MPN^-$
«extend»	concurrent	1..*	abhängig	MPN^+/MPN^- $\leftrightarrow MPN^+/MPN^-$
«threaten»	concurrent $\wedge dependentMPNs = \emptyset$	1..*	abhängig	$MPN^+ \leftrightarrow MPN^-$
«mitigate»	$\neg$concurrent	1..*	selbstständig	MPN^+/MPN^- $\leftrightarrow MPN^+$

Tabelle 6.3: Konfigurationsmöglichkeiten des Referenzsystems

Ausführen referenzierter MPNs, wobei das referenzierte MPN eine Alternative oder Teilsignatur darstellt, muss eine teilweise Blockierung und Freischaltung der aufrufenden Basis-MPN-Instanz möglich sein.

Tabelle 6.3 und Abbildung 6.7 zeigen anhand der vier vorgestellten Beziehungen der MUC-Sprache die entsprechenden Konfigurationen des Referenzsystems der MPN-Sprache. Eine «include»-Beziehung wird dabei auf einen nebenläufig (concurrent) ausgeführten Erweiterungspunkt (EP) abgebildet, der nur ein MPN referenziert. Im Gegensatz dazu werden «extend»- und «threaten»-Beziehungen zwar ebenfalls auf einen nebenläufig ausgeführten EP abgebildet, jedoch mit beliebig vielen referenzierten MPNs. Die eingezeichneten Teilabhängigkeiten bei «extend»- und «include»-Beziehungen können beliebig viele MPNs referenzieren (*dependentMPNs*), die zusammen mit den vom Erweiterungspunkt referenzierten MPNs behandelt werden müssen. Dabei handelt es sich um das Basis-MPN und andere MPNs vor diesem. Der aus der «threaten»-Beziehung entstandene EP referenziert immer MPNs aus der Menge MPN^-. Diese referenzierten MPNs sind dabei keine Teile oder Alternativen zu dem Basis-MPN, weshalb bei ihrer Auswer-

tung das Basis-MPN nicht mit einbezogen werden muss. Wird ein Erweiterungspunkt, der aus einer «threaten»-Beziehung entstanden ist, bei der Übersetzung erreicht, werden deshalb die in der Referenzierungskette vorhergehenden MPNs nicht mehr in die Menge der Teilabhängigkeiten (*dependetMPNs*) mit einbezogen. Zur Abbildung der «mitigate»-Beziehung wird ein nicht nebenläufig ausgeführter EP verwendet. Aktivierungsplätze können als Vorbedingung der Aktivierung genutzt werden, um Zustände des Basis-MPNs festzulegen, in denen das referenzierte MPN aktiviert wird.

Bei der Übersetzung einer «include»- bzw. «extend»-Beziehung enthält der Erweiterungspunkt, wie es im Beispiel (Abb. 6.5) für die Erweiterungspunkte EP1 und EP2 zu sehen ist, eine generationsspezifische Variable (repräsentiert in Abb. 6.7 durch *ep.end*). Diese kann im Basis-MPN an Transitionen eingesetzt werden, um einen Teilablauf der Signatur mithilfe einer Bedingung zeitweise zu blockieren, bis eines der referenzierten MPNs positiv beendet wurde. Bei der Übersetzung von Alternativen, die aus einer «extend»-Beziehung entstanden sind, muss die im Basis-MPN modellierte Teilsignatur aus diesem extrahiert werden, um eine einheitliche Verarbeitung der alternativen Abläufe zu ermöglichen.

Das in Kapitel 5 eingeführte Bedrohungsmodell (Abb. 5.1), das das Zusammenspiel der Signaturen beschreibt, ändert sich durch die Einführung des hier vorgestellten Referenzsystems nicht. Dies ist darauf zurückzuführen, dass Teilsignaturen, die zuvor als eigenständige Signaturen modelliert waren, nun durch die Referenzierung übergeordnet zusammengefasst werden. Durch die Referenzierung einzelner Teilsignaturen und die daraus entstehenden Gesamtsignaturen bleiben die akzeptierten Ereignissequenzen somit erhalten. Das Referenzsystem erlaubt zusätzlich die Wiederverwendung von Teilsignaturen, die von verschiedenen Basis-MPNs aus referenziert werden können.

6.2 Formalisierung des Referenzsystems

Im Folgenden wird die erweiterte Definition der Syntax der Monitor-Petrinetze aus Abschnitt 5.5 als Grundlage genommen und um die Konzepte des Referenzsystems erweitert. Hierbei bleiben alle syntaktischen Elemente der MPNs erhalten und es werden nur die zusätzlichen Elemente beschrieben.

6.2.1 *Syntax des Referenzsystems*

Zunächst wird eine zusätzliche Einordnung der MPN-Signaturen definiert, die den Initialzustand der Signaturen beschreibt. Die Einordnung unterscheidet zwischen Signaturen, die zu Beginn der Monitorausführung aktiviert bzw. deaktiviert sind.

Definition 45 (Aktivierte und deaktivierte MPNs). *Ein MPN kann zum Beginn der Überwachung aktiviert oder deaktiviert sein. Nur aktivierte MPNs reagieren während der Ausführung auf Eingabeereignisse.*

$$MPN = MPN^{act} \dot{\cup} MPN^{deact} \tag{6.1}$$

Im Allgemeinen sind MPNs, die durch ein anderes MPN referenziert werden, zu Beginn deaktiviert und nicht referenzierte MPNs aktiviert.

Beispiel *Aktivierte und deaktivierte MPNs*

Die (Teil-)Signaturen der Monitorspezifikation aus Abbildung 6.2 sind den Mengen MPN^{act} und MPN^{deact} folgendermaßen zugeordnet.

$$\begin{aligned}\mathrm{MPN}^{act} &= \{\text{CARDME Local}\}\\ \mathrm{MPN}^{deact} &= \{\text{DOS Attack, Tracking, CARDME Parking, Close Connection,}\\ &\quad \text{CARDME Local\_Alt\_EP1}\}\end{aligned}$$

Die extrahierte Erweiterung der *CARDME Local*-Signatur, die Teilsignatur *CARDME Local_Alt_EP1*, ist so spezifiziert, dass sie zu Anfang der Überwachung deaktiviert ist.

Die syntaktische Beschreibung der MPNs wird für die Einführung des Referenzsystems folgendermaßen erweitert. Hierbei werden identische Teile zu Definition 34 nicht wiederholt.

Definition 46 (Syntax). *Ein Netz* $\mathfrak{mpn} \in$ *MPN ist eine erweiterte Variante der Place/Transition-Netze über einem gegebenen Alphabet von möglichen Eingabeereignissen (engl. input events)* $I = \{i_1, \ldots, i_{|I|}, i_\epsilon\}$ *(wobei* i_ϵ *eine Eingabe mit einem Epsilonereignis ist, das spontane Transitionen feuern lässt) und einem festgelegten Set von Ausgangsereignissen (engl. output events)* $O = \{\mathtt{running}, \mathtt{terminated}, \mathtt{failed}, \mathtt{terminated\_temp}, \mathtt{failed\_temp}\}$ *mit*
$\mathfrak{mpn} = (S, T, F, E, G, I_{MPN}, EP, \mathfrak{m}, p, a)$ *wobei*

$ep \in EP = \mathcal{P}(S_{act}) \times \mathtt{Bool} \times \mathcal{P}(V \times G) \times \mathcal{P}(\mathrm{MPN}_{deact}) \times \mathcal{P}(\mathrm{MPN})$ *sind die Erweiterungspunkte eines MPNs, wobei die Potenzmenge der Menge der Aktivierungsplätze* S_{act} *entspricht. Der folgende boolesche Parameter entspricht der Eigenschaft* concurrent *und bestimmt, ob die referenzierten MPNs nebenläufig oder ersatzweise für das referenzierende MPN ausgeführt werden sollen. Des Weiteren besitzen Erweiterungspunkte eine Variable pro Generation, die Blockierung eines Teils des Basis-MPNs über eine Bedingung zu steuern. Die folgende Potenzmenge entspricht den durch den Erweiterungspunkt referenzierten MPNs. Die letzte Potenzmenge beschreibt, welche MPNs mit dieser Teilsignatur zusammenhängen. Diese Angabe der Teilabhängigkeiten wird genutzt, um die bei der Auswertung des Zustandes der MPN-Instanzen mit einzubeziehenden Teilsignaturen zu referenzieren.*

$\mathtt{extP} : \mathrm{MPN} \to \mathcal{P}(EP)$ *ist die Funktion, die eine Menge von Erweiterungspunkten, die dem übergebenden MPN zugeordnet sind, zurückliefert.*

$S_{act} \subseteq S$ *ist eine endliche Menge von Plätzen, die eine Aktivierung anderer MPNs auslösen, wenn ein Token in ihnen liegt.*

$\mathtt{epVar} : ep \times G \to \{\mathtt{true}, \mathtt{false}\}$ *ist eine Funktion, die zu einer Kombination eines Erweiterungspunktes und einer Generation den im Erweiterungspunkt gespeicherten Wahrheitswert zurückliefert. Die Funktion wird mit* `false` *initialisiert. Sie wird zum Blockieren von Teilabläufen im MPN genutzt.*

$\mathtt{dependentMPNs} : \mathsf{EP} \to \mathcal{P}(\mathsf{MPN})$ *ist die Funktion, die zu einem Erweiterungspunkt eine Menge von MPNs zurückliefert, die mit den referenzierten MPNs zusammenhängen.*

$\mathtt{refMPNs} : \mathsf{EP} \to \mathcal{P}(\mathsf{MPN}^{\mathtt{deact}})$ *ist die Funktion, die zu einen Erweiterungspunkt eine Menge von MPNs zurückliefert, die durch diesen referenziert werden.*

$\mathtt{act} \in \mathsf{Act} : \mathsf{MPN} \times \mathsf{G}_\mathtt{i} \to \{\mathtt{true}, \mathtt{false}\}$ *ist die Funktion, die abhängig von MPN und Generation zurückgibt, ob das MPN aktiviert ist oder nicht.*

$\mathtt{refGens} \in \mathsf{RefGens} : \mathsf{MPN} \times \mathsf{G} \times \mathsf{EP} \to \mathcal{P}(\mathsf{G}_\mathtt{i})$ *ist die Funktion, die für jedes MPN und einem dazugehörigen Erweiterungspunkt eine Generation auf weitere interne Eingabegenerationen abbildet. Sie liefert nach übergebener Generation die passende Eingabegeneration des referenzierten MPNs zurück. Wenn keine passende äquivalente Eingabegeneration zur übergebenen Kombination aus MPN, Generation und Erweiterungspunkt vorhanden ist, wird eine leere Menge zurückgegeben. Diese Funktion wird im Folgenden zur Verwaltung von neuen Instanzen, der durch Referenzen aktivierten MPNs, verwendet.*

$\mathtt{dependentGens} \in \mathsf{DependentGens} : \mathsf{MPN} \times \mathsf{G}_\mathtt{i} \times \mathsf{EP} \to \mathsf{G} \cup \{\bot\}$ *ist die Funktion, die für ein MPN, einen Erweiterungspunkt und eine Eingabegeneration die Generation aus dem übergebenden referenzierenden MPN zurückliefert, aus der die Eingabegeneration entstanden ist. Diese Funktion verwendet die* refGens*-Funktion zur Bestimmung der äquivalenten Generation. Wenn für die Kombination aus MPN, äquivalenter Eingabegeneration und Erweiterungspunkt keine Generation vorhanden ist, wird nicht definiert ($\bot$) zurückgegeben.*

Abbildung 6.5 zeigt am Beispiel des Erweiterungspunktes 1 (EP1) wie die Platzmenge S_{act} in einem Basis-MPN angeordnet sein kann. Im Allgemeinen kann die Menge S_{act} beliebig viele Plätze eines MPNs enthalten. Diese repräsentieren Markierungen des MPNs, in denen eine referenzierte Teilsignatur aktiviert werden soll. Wenn mindestens ein Platz aus S_{act} mit einem Token belegt ist, werden die referenzierten MPNs freigeschaltet. Liegt kein Token mehr auf einem der Plätze in S_{act}, werden alle referenzierten MPN-Instanzen, die der gerade verarbeiteten Generation zugeordnet sind, beendet.

Die Funktion *refMPNs* ordnet einem Erweiterungspunkt referenzierte MPNs zu. Im Gegensatz dazu ordnet die *dependentMPNs*-Funktion dem Erweiterungspunkt MPNs zu, die zu den referenzierten MPNs in Teilabhängigkeit stehen. In dem in Abbildung 6.4 gezeigten Beispiel sind diese Teilabhängigkeiten für *EP1* und *EP2* nur das Basis-MPN *CARDME Local* selbst, da es sich bei diesem um ein nicht referenziertes Wurzelelement handelt. Bei über EPs kaskadierten MPNs können sich Abhängigkeiten zu weiteren, davorliegenden MPNs in dieser Kette ergeben, die bei der Auswertung des Zustands des MPNs mit einbezogen werden müssen.

6.2.2 *Semantik des Referenzsystems*

Basierend auf der statischen Einteilung in aktivierte und deaktivierte MPNs in Definition 45, wird die Freischaltung der MPN-Instanzen zu Beginn der Monitorausführung initialisiert.

Definition 47 (Initialisierung der Freischaltung). *Die Freischaltung der MPN-Instanzen wird mithilfe der statischen Zuordnung der MPNs zu* MPN^{act} *und* MPN^{deact} *folgendermaßen initialisiert:*

$$act_0(mpn, g_i) = \begin{cases} true & \textit{for } mpn \in \mathrm{MPN}^{act} \\ false & else. \end{cases} ; g \in G \tag{6.2}$$

Während der Ausführung des Monitors müssen MPN-Instanzen, basierend auf der Markierung der referenzierenden Basis-MPNs, aktiviert bzw. deaktiviert werden.

Definition 48 (Aktualisierung der Freischaltung). *Änderungen der Freischaltung der MPNs werden folgendermaßen verarbeitet:*

$updateAct(mpn' : MPN, g_i' : G_i, b : Bool, \textbf{\textit{out}}\ act' : Act) :$
$\forall mpn \in MPN, \forall g_i \in G_i : act'(mpn, g_i) =$
$\quad \textit{if}\ mpn = mpn' \wedge g_i = g_i'\ \textit{then}\ b\ \textit{else}\ act(s, g_i)$

Hierbei bestimmt der Parameter b*, ob die zu aktualisierende Generation des MPNs aktiviert werden soll (*$true$*) oder deaktiviert werden soll (*$false$*).*

Im späteren Teil der Definition des Referenzsystems ist es notwendig, die Menge der vorhandenen Generationen nach der höchsten (als letztes erstellten) Generation zu filtern.

Definition 49 (Höchste Eingabegeneration). *Die Funktion* $maxGen(G_i : \mathcal{P}(\mathbb{N})) \rightarrow G_i$ *bestimmt die höchste vorhandene Generation eines übergebenen MPNs.*

$$maxGen(G_i) = g_i \in G_i \mid \nexists(g_i' \in G_i > g_i) \tag{6.3}$$

Im Laufe der Ausführung des Monitors werden durch Abhängigkeiten zwischen den einzelnen MPNs neue Eingabegenerationen, die zu einer übergebenen Eingabegeneration äquivalent sind, angelegt.

Beispiel *Äquivalente Eingabegenerationen*

Durch das Referenzsystem sollen auch verschränkt ausgeführte MPNs (mit Subgenerationen) unterstützt werden. Abbildung 6.8 zeigt ein Beispiel, in dem diese neuen äquivalenten Generationen benötigt werden.

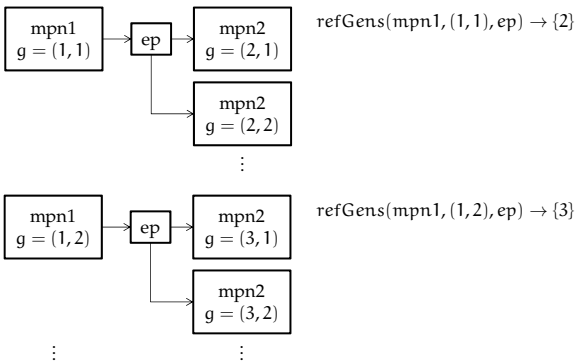


Abbildung 6.8: Äquivalente Eingabegenerationen

Wird ein referenziertes MPN von einer Basis-MPN-Instanz aktiviert, die verschränkt überwacht wird (Subgenerationen verwendet), muss eine Zuordnung der aktuellen Generation zu einer neuen internen Eingabegeneration stattfinden. Da das referenzierende Basis-MPN mehrere neue MPN-Instanzen für jede seiner Subgenerationen erzeugen kann, reicht es nicht aus die ursprüngliche Eingabegeneration zu verwenden. Es müssen neue der ursprünglichen Generation zugeordnete (äquivalente) Eingabegenerationen erzeugt werden.

In Abbildung 6.8 sind sowohl die Signaturen *mpn1* als auch *mpn2* verschränkt überwachte Signaturen. Für die Eingabegeneration 1 wurden im *mpn2* zwei äquivalente Eingabegenerationen (2 und 3) erzeugt, um die Unterscheidung zwischen von Generation 1.1 und 1.2 erzeugten verschränkten Überwachungsversuchen in *mpn2* zu ermöglichen. Über die Funktion $\mathtt{refGens}$ kann für jedes Basis-MPN und einen der dazugehörigen EPs zu der aktuellen Generation (1.1 bzw. 1.2) die entsprechende äquivalente Eingabegeneration des referenzierten MPNs (2 bzw. 3) bestimmt werden.

Mit der folgenden Funktion werden diese Generationszuordnungen aktualisiert.

Definition 50 (Aktualisierung der äquivalenten Generationen). *Änderungen an den von anderen Generationen abhängigen Eingabegenerationen werden folgendermaßen verarbeitet:*

$$
\begin{aligned}
&\mathtt{updateRefGens}(\mathtt{mpn}' : \mathtt{MPN}, \mathtt{g}' : \mathtt{G}, \mathtt{g}_{\mathtt{inew}} : \mathbb{N}, \mathtt{ep}' : \mathtt{EP}, \mathtt{add} : \mathtt{Bool},\\
&\quad \textbf{\textit{inout}}\ \mathtt{refGens} : \mathtt{RefGens}) :\\
&\forall \mathtt{mpn} \in \mathtt{MPN}, \forall \mathtt{ep} \in \mathtt{mpn.EP}, \forall \mathtt{g} \in \mathtt{mpn.G} : \mathtt{refGens}(\mathtt{mpn}, \mathtt{g}, \mathtt{ep}) =\\
&\quad \textit{if}\ \mathtt{mpn} = \mathtt{mpn}' \wedge \mathtt{ep} = \mathtt{ep}' \wedge \mathtt{g} = \mathtt{g}'\ \textit{then}\\
&\qquad \textit{if}\ \mathtt{add} = \mathtt{true}\ \textit{then}\ \mathtt{refGens}(\mathtt{mpn}', \mathtt{g}, \mathtt{ep}) \cup \{\mathtt{g}_{\mathtt{inew}}\}\ \textit{else}\\
&\qquad\quad \mathtt{refGens}(\mathtt{mpn}', \mathtt{g}, \mathtt{ep}) \setminus \{\mathtt{g}_{\mathtt{inew}}\}\\
&\quad \textit{else}\ \mathtt{refGens}(\mathtt{mpn}', \mathtt{g}, \mathtt{ep})
\end{aligned}
$$

Wenn eine MPN-Instanz, die im Referenzsystem eingebettet ist, beendet wird, kann dies Einfluss auf die von ihr referenzierten MPN-Instanzen haben.

Beispiel *Abstrakte Beschreibung der Abarbeitung von Referenzen*

Abbildung 6.9 zeigt, welche Auswirkungen das Beenden einer MPN-Instanz haben kann, die über das Referenzsystem in Beziehung zu anderen Instanzen steht. Da im Allgemeinen zur Laufzeit auch mehrere MPN-Instanzen von einer

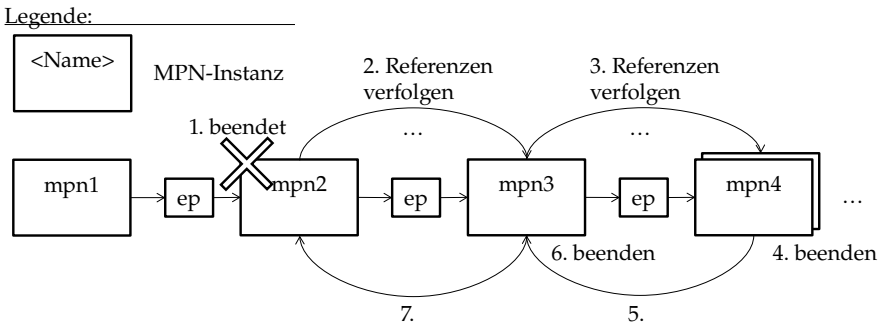


Abbildung 6.9: Verarbeitung von Referenzen

Basis-MPN-Instanz aus referenziert werden können, handelt es sich bei den Abhängigkeiten um eine Baumstruktur mit einer Instanz eines nicht referenzierten MPNs als Wurzel. Hierbei wird die gerade aktuelle Generation bzw. ihre äquivalenten Generationen (bei Verwendung von Subgenerationen) bearbeitet. Sobald die Instanz *mpn2* sich beendet (*failed / terminated*), sind alle von ihr abhängigen referenzierten Instanzen auch zu beenden, da ihr Gültigkeitskontext verlassen wird. Dies findet in einem Bottom-up-Verfahren statt. Zunächst werden alle Referenzen verfolgt, und sobald ein Blatt gefunden wurde, alle Instanzen rekursiv beendet (*terminated*).

Die beendeten nachfolgenden MPN-Instanzen werden durch folgende Prozedur zurückgesetzt.

Definition 51 (Beenden von referenzierten MPN-Instanzen). *Diese Prozedur beendet rekursiv im Bottom-up-Verfahren alle MPN-Instanzen, die von der übergebenen Generation abhängen.*

```
endingReferencedMPNs(srcMPN : MPN, ep : EP, g : G, inout mres : O[][]) :
 foreach refMPN ∈ refMPNs(ep)
  if (ep.concurrent) // wenn parallel ausgef. EP
   foreach g_iref ∈ refGens(srcMPN, g, ep)
    foreach g_ref ∈ {(g_ic, g_sc)|(g_ic, g_sc) ∈ refMPN.G ∧ g_ic = g_iref} // alle Subgenerationen
     if (mres[refMPN][g_ref] == running)
      foreach ep' ∈ refMPN.extP(refMPN) // Referenzen verfolgen
       endingReferencedMPNs(refMPN, ep', g_ref, mres);
      mres[refMPN][g_ref] = terminated; // Beenden
      refMPN.G = refMPN.G \ g_ref;
    updateRefGens(refMPN, g, g_iref, ep, false, srcMPN.refGens);
```

Beim rekursiven Beenden der referenzierten MPN-Instanzen werden Erweiterungspunkte ignoriert, die als nicht nebenläufig markiert wurden. Die Rekursion wird hier abgebrochen, da es sich um keine nebenläufig zu überwachende Signatur handelt. Diese Signaturen agieren, wenn sie aktiviert wurden, selbstständig und müssen, falls sie mehr als einen Makroschritt umfassen, eigenständig mittels annotierter Ereignisse, Bedingungen und Aktionen ablaufen. Des Weiteren werden nur MPN-Instanzen, die noch als laufend (*running*) markiert sind, beendet, da die Abarbeitung der MPN-Instanzen von den Blättern zur Wurzel stattfindet und damit nachfolgende Instanzen schon behandelt wurden.

Die folgenden beiden Definitionen beschreiben, wie ein MPN von außen deaktiviert und beendet bzw. aktiviert wird. In Definition 52 wird hierzu zunächst das übergebene MPN für die entsprechende Eingabegeneration deaktiviert. Darauf folgend werden alle MPN-Instanzen, die von dieser zu deaktivierenden MPN-Instanz abhängen, ebenfalls beendet. Hierzu wird die in Definition 51 beschriebene rekursive Prozedur aufgerufen. Je nach übergebenem Monitorzustand aus der Menge der Ausgangsereignisse wird das Ergebnis der zu deaktivierenden MPN-Instanz aktualisiert. Dabei bedeutet ein als neuer Monitorzustand übergebener Wert *null*, dass das Ergebnis nicht geändert werden muss.

Definition 52 (Deaktivieren und Beenden von referenzierten MPN-Instanzen). *Diese Prozedur deaktiviert und beendet referenzierte MPN-Instanzen und aktualisiert die Ausgangsereignisse dieser MPN-Instanzen.*

```
deactivateAndEndRefMPNs(inout mpn : MPN, g : ℕ × ℕ, newres : O ∪ {null},
  inout mres : O[][])
 updateAct(mpn, g.g_i, false, mpn.act); // Deaktivieren des Basis—MPNs
 foreach ep ∈ extP(mpn) // Beenden aller referenzierten Instanzen
  endingReferencedMPNs(mpn, ep, g, mres);
 if (newres ≠ null) // Setzen des neuen Zustands
  mres[mpn][g] = newres;
```

Mithilfe der folgenden Prozedur wird das Aktivieren einer referenzierten MPN-Instanz durchgeführt. Nach der Aktivierung der MPN-Instanz wird ihr ein Epsilonereignis übergeben, um das MPN in einen stabilen Zustand zu überführen und das Erreichen von Terminalplätzen zu erkennen. Dies wird u. a. für die Spezifikation sofort ausgeführter Gegenmaßnahmen benötigt. Daraufhin muss der Monitorzustand dieser MPN-Instanz aktualisiert werden.

Definition 53 (Aktivieren von referenzierten MPNs). *Diese Prozedur aktiviert referenzierte MPNs für die übergebene Generation und aktualisiert den aktuellen Monitorzustand.*

```
activateMPN(inout mpn : MPN, g : ℕ × ℕ, process : Bool, inout mres : O[][])
 g_inew = maxGen(mpn.G_i) + 1;
 g_new = (g_inew, 1);
 mpn.G = mpn.G ∪ {g_new}; // Erstellen einer neuen Generation
 init(mpn.m, g_new, mpn.m);
```

```
updateAct(mpn, g_inew, true, mpn.act); // Ausführung
updateRefGens(mpn, g, g_inew, true, mpn.refGens);
mresult = running;
if (process)
 fired[g] = true;
 mpn.postprocessing(mpn.m, g_inew, mpn.e, fired, false, mresult);
mres[mpn][g_new] = mresult;
```

Bei der Verwendung des Referenzsystems werden Signaturen in mehrere MPNs aufgeteilt. Die Auswertung der einzelnen MPN-Instanzen auf lokaler Basis reicht somit nicht mehr aus, da ein MPN, das nicht geschaltet hat, im Gegensatz zu ganzheitlich modellierten Signaturen nicht zwangsläufig auch beendet werden muss. Es ist sein gesamter Kontext (alle zusammenhängenden Teilsignaturen) zu betrachten. Hierzu wird im Folgenden Definition 39 für eine Unterscheidung zwischen erreichten Terminalplätzen und dem Nicht-Schalten eines MPNs angepasst. Die in Definition 46 neu hinzugekommenen Ausgangsereignisse *terminated_temp* und *failed_temp* werden verwendet, um deren endgültige Evaluation in einen zusätzlichen globalen Auswertungsschritt auszulagern.

Definition 54 (Auswertung des Überwachungsergebnisses mit Referenzsystem). *Diese Funktion evaluiert das Ergebnis der Ausführung des MPNs für eine Generation. Das Ergebnis (∈ O) hängt von der Markierung der Terminalplätze und dem Fakt, ob die MPN-Instanz geschaltet hat, ab. Falls das MPN nicht geschaltet hat, wird anhand der Belegung der als vorausgehender (antecedent) oder nachfolgender (consequent) Teil der Signatur markierten Plätze und ob das MPN einen Use- oder Misusecase darstellt, das Ergebnis bestimmt.*

```
evaluateMonitoringResult(m : M, g : G, fired : Bool[]) → O :
 if fired[g] then
  if ∃s ∈ S_f : m(s, g) = 1 then failed
  else if ∃s ∈ S_e : m(s, g) = 1 then terminated
  else running
 else
  if mpn ∈ MPN⁻ then terminated_temp
  else
   if ∃s ∈ S_c|m(s, g) = 1 then failed_temp;
   else terminated_temp;
```

Durch die Einführung des Referenzsystems kann die Auswertung der einzelnen MPN-Instanzen und insbesondere das Beenden dieser beim Nicht-Schalten nicht mehr lokal von der entsprechenden MPN-Instanz getroffen werden. Aus diesem Grund wird auch die in Kapitel 6 eingeführte Nachverarbeitung (Def. 40) modifiziert, sodass sie das Löschen der Monitorinstanzen den nachfolgenden Schritten überlässt.

Definition 55 (Nachverarbeitung ohne direkte Beendigung). *In diesem Schritt schalten zunächst alle Epsilontransitionen, die aktiviert sind. Dies wird so lange fortgesetzt, bis keine Änderungen im Netz mehr auftreten. Wenn eine Generation eines MPNs zumindest ein Token in einem Terminalplatz hat, oder wenn das Netz im Makroschritt nicht gefeuert hat, wird die Markierung m ausgewertet. In dieser Version der Prozedur wird danach der Teil der Markierung, der zur aktuell verarbeiteten Generation gehört, nicht auf die initiale Markierung m_0 zurückgesetzt, um eine globale Verarbeitung der MPN-Instanzen unter Einbeziehung des Referenzsystems zu ermöglichen.*

```
postprocessing(inout m : M, g_i : G_i, inout e : E, fired : Bool[], newSubgen : Bool,
    out mresult : O[]) :
 epsfired = true;
 if (newSubgen) maxSub(subGen(g_i)) − 1; // ignoriere neue Subgeneration
 subGens = subGen(g_i); // hole Subgenerationen von g_i
 while (|subGens| > 0)
  g_s = minSub(subGens); // kleinste Subgeneration
  subGens = subGens \ g_s; // entferne kleinste Subgeneration
  g = (g_i, g_s);
  while (epsfired) // so lange das Netz geschaltet hat
   processEvent(m, i_ε, g, e, newSubgen, epsfired); // newSubGen wird hier ignoriert
   if (epsfired)
    fired[g] = true;
 subGens = subGen(g_i); // hole Subgenerationen von g_i
 while (|subGens| > 0)
  g_s = minSub(subGens); // kleinste Subgeneration
  subGens = subGens \ g_s; // entferne kleinste Subgeneration
  g = (g_i, g_s);
  mresult[g] = evaluateMonitoringResult(m, g, fired);
  //if mresult[g] ≠ running then
  // G = G \ {g}
```

Die Erweiterung der Ausgangsereignisse und die Auslagerung des Zurücksetzens der MPN-Instanzen sind die beiden einzigen Änderungen an der ursprünglichen MPN-Semantik.

Das Referenzsystem soll eine gezielte Überwachung von Signaturen auf Basis des Überwachungszustandes (Markierung) anderer Signaturen unterstützen. Auch hierbei sollen so wenig wie möglich Änderungen an der ursprünglichen Definition der MPN-Sprache durchgeführt werden. Deshalb wird im Folgenden eine unabhängig von der Ereignisauswertung aufrufbare Prozedur definiert, die je nach Markierung der Instanzen des Basis-MPNs die referenzierten MPN-Instanzen freischaltet oder sperrt.

Definition 56 (Analyse des MPNs auf freizuschaltende und zu blockierende referenzierte MPNs). *Diese Prozedur analysiert die durch die Generation gegebene Markierung des übergebenen MPNs und schaltet basierend auf den vorhandenen Erweiterungspunkten die referenzierten MPN-Instanzen frei. Dies geschieht auf Basis der Aktivierungsplatzmenge S_{act}. Im Fall, dass ein referenziertes MPN deaktiviert wird (S_{act}*

des Basis-MPNs ist nicht belegt), wird es zurückgesetzt. Bei Erweiterungspunkten, die nicht nebenläufig sind, wird dieser Aktivierungsschritt ignoriert.

```
analyseDependencies(inout mpn : MPN, g : G, inout mres : O[][]) :
 foreach ep = extP(mpn)
  activateRefMPNs = false;
  // überprüfen, ob ein Aktivierungsplatz belegt und der EP nebenläufig ist
  if (∃s ∈ ep.S_act | m(s, g) = 1 ∧ ep.concurrent)
   activateRefMPNs = true;
  foreach mpn = refMPNs(ep)
   foreach g_iref ∈ refGens(mpn, g, ep)
    if (activateRefMPNs ∧ ¬act(mpn, g_iref))
     activateMPN(mpn, g, false, mres);
     updateEPVar(ep, g, false); // Zurücksetzen der EP−Variablen
    elseif (¬activateRefMPNs) // Generation aus dem aktuellen MPN entfernen
     endingReferencedMPNs(mpn, ep, g, null, mres);
```

Die Einführung des Referenzsystems führt dazu, dass entschieden werden muss, wann eine MPN-Instanz mit Subgenerationen im Gesamten fehlgeschlagen ist und wann nicht. Diese Entscheidung wurde in der Definition der MPNs bisher ausgespart und Subgenerationen als eigene Instanzen angesehen. Für die Auswertung im Referenzsystem muss nun ein Monitorzustand für die Eingabegeneration definiert werden.

Definition 57 (Monitorzustand für die Eingabegeneration). *Wenn eine der Subgenerationen fehlgeschlagen ist, bedeutet dies, dass auch die dazugehörige Eingabegeneration fehlgeschlagen ist.*

```
monitoringState(mpn : MPN, g_i : ℕ, mres : O[][]) → O :
 if |mpn.subGen(g_i)| > 1
  if ∃mres[mpn][g] == failed : g ∈ {{g_i} × mpn.subGen(g_i)} then
   failed
  else running
 else
  mres[mpn][(g_i, 1)];
```

Im Verlauf der Überwachung müssen immer wieder MPN-Instanzen, die nicht mehr gültig sind, beendet werden. Hierzu zählen auch alle durch die zu beendende MPN-Instanz referenzierten MPN-Instanzen.

Definition 58 (Beenden eines MPNs). *Diese Prozedur übernimmt die Aufgabe, beendete MPN-Instanzen aus den Netzen zu entfernen. Hierbei kann unterschieden werden, ob nur eine Subgeneration entfernt werden soll oder die gesamte Eingabegeneration.*

```
endMPN(srcMPN : MPN, g_i : ℕ, g_s : ℕ, newres : O, onlySub : Bool,
  inout mres : O[][]) :
 if (¬onlySub)
```

```
 foreach g ∈ {g_i × subGen(g_i)}
  deactivateAndEndRefMPNs(srcMPN, g, newres, mres);
  srcMPN.G = srcMPN.G \ {g};
else // nur Subgen löschen
 g = (g_i, g_s);
 deactivateAndEndRefMPNs(srcMPN, g, newres, mres);
 srcMPN.G = srcMPN.G \ {g}
```

Die folgende Funktion setzt die Variablen der Erweiterungspunkte, die in den Basis-MPNs verwendet werden, auf einen neuen Wert, um blockierte Teile des Basis-MPNs nach paralleler Überwachung referenzierter MPNs wieder freizuschalten.

Definition 59 (Setzen der Erweiterungspunktvariablen). *Mit dieser Funktion werden erweiterungspunkt- und generationsspezifische Variablen auf* true *bzw.* false *gesetzt.*

$$\begin{aligned}&\mathtt{updateEPVar}(ep' : EP, g' : \mathbb{N} \times \mathbb{N}, b : \mathtt{Bool}) :\\ &\forall mpn \in MPN, \forall ep \in mpn.EP, \forall g \in mpn.G : mpn.epVar(ep, g) =\\ &\quad \textit{if } ep = ep' \wedge g = g' \textit{ then } b \textit{ else } epVar(ep, g)\end{aligned}$$

Die folgende Prozedur wertet, basierend auf dem Zustand der Monitorinstanzen, die eingeführten Abhängigkeiten (Referenzen) aus. Durch die Anpassung von Definition 54 und die Auswertung des Ergebnisses der einzelnen Monitorinstanzen im Vorfeld, muss nicht mehr unterschieden werden zu welcher Teilklasse (MPN^+ oder MPN^-) die MPNs an den Enden der Referenz gehören. Basierend auf dem aktuellen Zustand (*running*, *terminated*, *failed*, *terminated_temp* und *failed_temp*) und der Konfiguration der Erweiterungspunkte werden die vom Referenzsystem herrührenden Änderungen an den MPN-Instanzen vorgenommen. Diese Änderungen beinhalten das Beenden eines MPNs und das Deaktivieren von MPN-Instanzen. Die Ergebnisse des in den MPNs inhärenten Konzepts der Auswertung eines Ergebnisses nach der Belegung der Plätze werden nur dann vom Referenzsystem überschrieben, wenn es durch referenzierte MPNs zu zusätzlichen Beendigungsbedingungen kommt. Bei Änderungen an MPN-Instanzen, die den Zustand der Überwachung betreffen, werden Änderungen an den Ergebnissen durchgeführt, um nach der Verarbeitung eine aktuelle und zum Monitorzustand konsistente Darstellung des Endzustandes der Ausgangsereignisse zu erhalten.

Beispiel *Auswertung der Ergebnisse auf Basis des Referenzsystems*

In Abbildung 6.10 ist die in der nachfolgenden Definition beschriebene Auswertung der Abhängigkeiten zwischen den MPN-Signaturen abstrakt dargestellt. Zunächst wird der Zustand des gerade betrachteten MPNs ausgewertet. Ist dieser *failed* oder *terminated*, müssen alle nachfolgenden MPN-Instanzen auch beendet werden. Hat die Instanz nicht geschaltet, wird seine direkte Umgebung ausgewertet, um ein global korrektes Ergebnis zu erreichen. Dieser Schritt

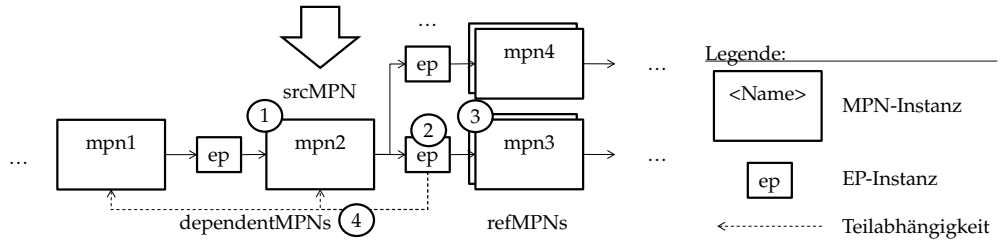


Abbildung 6.10: Auswertung der Ergebnisse auf Basis des Referenzsystems

umfasst die über den gerade betrachteten EP des Basis-MPNs referenzierten MPN-Instanzen und die als im Erweiterungspunkt abhängig (Teilabhängigkeit) markierten Instanzen.

Definition 60 (Auswertung der Referenzen). *Diese Prozedur wertet den Zustand eines parallel oder nachfolgend gestarteten MPNs aus und führt die Nachbearbeitung durch. Es werden ausgehend von den Erweiterungspunkten noch nicht abschließend ausgewertete Zustände der einzelnen MPNs global ausgewertet. Hierbei findet, falls die gerade betrachtete MPN-Instanz keine MPN-Instanzen referenziert, eine spezielle Vorverarbeitung statt. Falls das Basis-MPN fehlschlägt, werden die durch als nicht nebenläufig gekennzeichneten EPs referenzierten MPNs ausgeführt.*

```
processExtensionPoints(inout srcMPN : MPN, g_i : G_i, ep : EP, mres_temp : O[][],
    inout mres : O[][]) :
 if (ep.concurrent ∧ dependentMPNs(ep) ≠ ∅) // parallel mit dependentMPNs
  // nicht final ausgewertete Referenzen (Blätter) vorverarbeiten
  foreach refMPN ∈ refMPNs(ep)
   foreach g_s ∈ srcMPN.subGen(g_i) // über alle SubGens des Basis−MPNs iterieren
    g = (g_i, g_s);
    foreach g_iref ∈ refGens(srcMPN, g, ep) // RefGen bestimmen
     if (monitoringState(refMPN, g_iref, mres) == terminated_temp
       ∨monitoringState(refMPN, g_iref, mres) == failed_temp)
      if (mres[srcMPN][g] == terminated|falied|running
        ∨∃refMPN2 ∈ refMPNs(ep)
        : monitoringState(refMPN2, g_iref, mres) == terminated|falied|running)
       // ein MPN hat geschaltet
       foreach (g_ref ∈ {g_iref × refMPN.subGen(g_iref)})
        mres[refMPN][g_ref] = running;
      elseif (mres[refMPN][g_ref] == terminated_temp)
       foreach (g_ref ∈ {g_iref × refMPN.subGen(g_iref)})
        mres[refMPN][g_ref] = terminated;
      elseif (mres[refMPN][g_ref] == failed_temp)
       foreach (g_ref ∈ {g_iref × refMPN.subGen(g_iref)})
        mres[refMPN][g_ref] = failed;

 // Normalfall für alle anderen MPN−Instanzen
 foreach g_s ∈ srcMPN.subGen(g_i) // alle SubGens des Basis−MPN durchgehen
  g = (g_i, g_s);
  if (mres[srcMPN][g] == failed ∨ mres[srcMPN][g] == terminated)
   // Löschen von srcMPN und refMPNs
```

```
    endMPN(srcMPN, g_i, g_s, null, false, mres);
   elseif (mres[srcMPN][g] == terminated_temp ∨ mres[srcMPN][g] == failed_temp)
    foreach g_iref ∈ refGens(srcMPN, g, ep)
     if (∃refMPN ∈ refMPNs(ep)
       : monitoringState(refMPN, g_iref, mres) == running)
      mres[srcMPN][g] = running;
      if (∃mres_temp[refMPN][g_ref] == terminated
        : g_ref = (g_i, g_s) ∈ refMPN.G : g_i ∈ refGens(srcMPN, g, ep))
       updateEPVar(ep, g, true); // Setzen der blockierenden Variablen auf true
       // schalte srcMPN hinter EP—Bedingung
       srcMPN.macro(srcMPN, i_ϵ, g_i, mresult);
       mres[srcMPN][g] = mresult[g];
    elseif (∃dependentMPN ∈ dependentMPNs(ep)
      : dependentGens(dependentMPN, g_i, ep) ≠⊥
      ∧mres[dependentMPN][dependentGens(dependentMPN, g_i, ep)] == running)
     mres[srcMPN][g] = running;
    else
     if (mres[srcMPN][g] == terminated_temp)
      newstate = terminated;
     elseif (mres[srcMPN][g] == failed_temp)
      newstate = failed;
     // Löschen von srcMPN und refMPNs
     endMPN(srcMPN, g_i, g_s, newstate, false, mres);

 if (¬ep.concurrent) // nachfolgend
  foreach g_s ∈ srcMPN.subGen(g_i) // über alle SubGens des Basis—MPNs iterieren
   g = (g_i, g_s);
   if (mres[srcMPN][g] == failed ∧ (ep.S_act = ∅ ∨ (∃s ∈ ep.S_act | m(s, g) = 1)))
    foreach refMPN ∈ refMPNs(ep)
     activateMPN(refMPN, g, true, mres);
```

Zusätzlich zum Schalten der MPN-Instanzen muss durch das Referenzsystem die vorhandenen Teilabhängigkeiten zwischen den MPNs in einem globalen Auswertungsschritt beachtet werden. Hierzu ist es notwendig, alle zu der gerade verarbeiteten Eingabegeneration zugehörigen äquivalenten Eingabegenerationen zu kennen. Die folgende Prozedur sammelt alle in den MPNs vorhandenen äquivalenten Eingabegenerationen.

Definition 61 (Alle äquivalenten Eingabegenerationen). *Diese Prozedur sammelt alle äquivalenten Eingabegenerationen zu einer Eingabegeneration* g_i *und gibt diese als* G_{iRef} *zurück.*

```
equivalentInputGenerations(g_i : ℕ, inout G_iRef : ℙ(N))
 foreach mpn ∈ MPN
  foreach ep ∈ extP(mpn)
   foreach g_s ∈ mpn.subGens(g_i);
    g = (g_i, g_s);
    G_irefCur = refGens(mpn, g, ep);
    G_iRef = G_iRef ∪ G_irefCur;
```

```
  foreach g_irefCur ∈ G_irefCur
    equivalaentInputGenerations(g_irefCur, G_iRef);
```

Das Referenzsystem wird, bis auf die kleinen Änderungen an der MPN-Semantik, die die Erweiterung der Ausgangsereignisse und das Auslagern des Zurücksetzens eines Monitors beinhaltet, als optionaler Teil außerhalb des eigentlichen MPN-Konzepts definiert. Hierzu wird eine zusätzliche Prozedur eingeführt, die die zusammenfassende Verwaltung mehrerer MPNs und ihre durch das Referenzsystem eingeführten globalen Zusammenhänge verarbeitet.

In der folgenden Prozedur werden zwei zusätzliche Schritte durchgeführt, nachdem die MPNs aufgrund des Eingabeereignisses geschaltet haben. Zusätzlich zum normalen Schalten der MPNs findet eine Auswertung der von der Eingabegeneration abhängigen äquivalenten Eingabegenerationen statt. Alle Ergebnisse der Signaturauswertung werden nach ihrer Generation abgespeichert, um in den folgenden Schritten als Eingabe zu dienen.

Im ersten zusätzlichen Schritt werden alle MPNs nacheinander auf Veränderungen der Referenzen analysiert. Dabei wird von der höchsten zur niedrigsten Eingabegeneration vorgegangen. Dies führt durch die Aktivierungsreihenfolge der MPN-Instanzen dazu, dass implizit die referenzierten MPNs im Bottom-up-Verfahren bearbeitet werden. Hierdurch werden Änderungen, die durch die processExtensionPoints-Prozedur an den Basis-MPNs in diesem Schritt durchgeführt werden, ebenfalls beachtet. Subgenerationen werden bei der Auswertung der Referenzen, wie auch im Makroschritt der MPNs, nach ihrer Subgenerationsnummer in aufsteigender Reihenfolge verarbeitet.

Im zweiten der zusätzlichen Schritte, werden auf dem Ergebnis der beiden ersten Schritte MPNs für bestimmte Generationen aktiviert bzw. deaktiviert. Diese Änderungen haben erst Einfluss auf das nächste verarbeitete Eingabeereignis.

Definition 62 (Kontrollstruktur des Referenzsystems). *Diese Prozedur führt die Verarbeitung von Ereignissen auf allen MPNs aus.*

```
processMPNs(i : I, g_i : G_i, out mres : O[][]) :
 // Sammeln der vorhandenen Eingabegenerationen
 G_iall = {g_i};
 equivalentInputGenerations(g_i, G_iall);

 // Verarbeitung des Ereignisses lokal in den MPNs
 foreach mpn ∈ MPN
  foreach g_icur ∈ G_iall
   if (g_icur ∈ mpn.G_i)
    if (act(mpn, g_icur))
     macro(mpn, i, g_icur, mresult);
     foreach g ∈ {(g_ic, g_sc)|(g_ic, g_sc) ∈ mpn.G ∧ g_ic = g_icur}
      mres[mpn][g] = mresult[g];

 // Globale Analyse der Ergebnisse
 G_ialltemp = G_iall; // Kopie der zu verarbeitenden Eingabegenerationen
```

```
mres_temp = mres; // Kopie der Ergebnisse erstellen
while (|G_ialltemp| > 0) // von der höchsten zur niedrigsten Eingabegeneration
 g_icur = maxGen(G_ialltemp); // höchste Eingabegeneration
 G_ialltemp = G_ialltemp \ {g_icur}; // entferne höchste Generation
 foreach mpn ∈ MPN
  if (g_icur ∈ mpn.G_i ∧ refMPNs(mpn) ≠ ∅)
   foreach ep ∈ extP(srcMPN)
    processExtensionPoints(mpn, g_icur, ep, mres_temp, mres);

// Überprüfung der Aktivierung
foreach mpn ∈ MPN
 foreach g_icur ∈ G_iall
  foreach g ∈ {(g_ic, g_sc)|(g_ic, g_sc) ∈ mpn.G ∧ g_ic = g_icur}
   if (mres[mpn][g] == running)
    analyseDependencies(mpn, g, mres);
```


Mit dem hier eingeführten Konzept ist es möglich, auch rekursive Abläufe aus der Spezifikationssprache in MPNs zu modellieren.

Beispiel *Rekursion in der MPN-Sprache mit Referenzsystem*

Abbildung 6.11 zeigt eine einfache Spezifikation einer rekursiven Signatur in der MPN-Sprache mit Referenzsystem. Die Signatur besteht aus einem MPN, dass über einen nebenläufigen Erweiterungspunkt auf sich selbst referenziert. Tritt das Ereignis *e1* auf wird der Zähler *x* inkrementiert. Darauf folgend findet eine Überprüfung des Zählers statt. Wenn der Zähler kleiner der gleich 5 ist, wird der Aktivierungsplatz des EPs belegt. Hierdurch wird eine neue Instanz mit äquivalenter Eingabegeneration initialisiert und die Basis-MPN-Instanz über die Bedingung *[ep.end]* blockiert. In den weiteren Schritten, in denen *e1* Auftritt, schalten immer alle aktiven MPN-Instanzen. In der „neusten" Instanz schaltet die obere Transition, die einen Initialplatz in ihrem Vorbereich besitzt, und in den anderen Instanzen die Eigentransition. Hierdurch werden die Subgenerationen und damit die Gesamtsignatur nicht beendet.

Tritt das Ereignis *e1* ein und der Zähler überschreitet den Wert 5, beendet sich die aktuelle MPN-Instanz. Hierdurch wird in der Basis-MPN-Instanz die blockierte Transition über die Variable des diese MPN-Instanz referenzierenden Erweiterungspunkts freigeschaltet. Auf diese Art und Weise beenden sich alle vorherigen Instanzen.

Erreichte Ziele: Durch die Einführung eines übergeordneten Referenzsystems ist es gelungen die eigentliche MPN-Semantik, bis auf den Auswertungsfokus, nicht zu verändern. Das Referenzsystem kann somit als optionaler Teil angesehen werden und erlaubt es bei einfachen, weniger komplexen Monitorspezifikationen den potenziellen Overhead des Referenzsystems vollständig zu vermeiden. Falls es sich um eine komplexe Spezifikation mit vielen Abhängigkeiten zwischen den Signaturen handelt, erlaubt es das Referenzsystem von den in der Quellspezifikation angegebenen Zusammenhängen zu profitieren. Des Weiteren unterstützt das

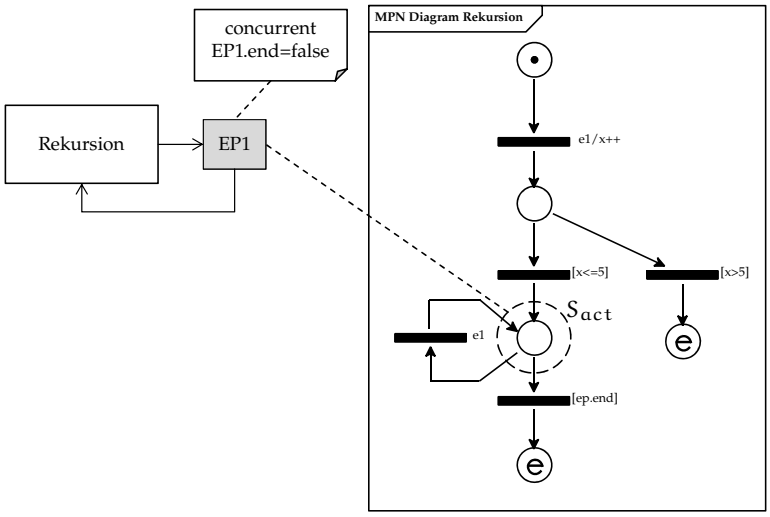


Abbildung 6.11: Rekursion in der MPN-Sprache mit Referenzsystem

Referenzsystem die bedingte Ausführung von komplexen Gegenmaßnahmen, die ebenfalls als MPN modelliert sind.

Zur Bestimmung der Analysereihenfolge der Erweiterungspunkte kann statt der Verwendung von neuen, äquivalenten Eingabegenerationen für jede aktivierte MPN-Instanz auch die Referenzsystem-Netzstruktur herangezogen werden. Dies würde den Vorteil mit sich bringen, dass neu aktivierten MPN-Instanzen nur wenn diese durch Subgenerationen nutzende MPNs aktiviert werden oder wenn MPNs sich rekursiv aufrufen neue äquivalente Eingabegenerationen zugeordnet werden müssen. Eine solche Umsetzung würde zu einem geringeren Speicherverbrauch (RAM) und einer kürzeren Laufzeit führen.

6.2.3 *Modellierungsrichtlinien für das Referenzsystem*

Auch für das Referenzsystem existieren Modellierungsrichtlinien, bei deren Nichteinhaltung fehlerhafte Signaturen entstehen.

- Globale Variablen, die für den gesamten Monitor gelten, dürfen nicht abhängig von der Abarbeitungsreihenfolge der MPNs sein, es sei denn, sie wird durch das Referenzsystem vorgegeben.

- Nicht alle Kombinationen von MPNs mit Subgenerationen und Beziehungen zwischen den MPNs sind sinnvoll. So muss eine ausgelagerte nebenläufig (concurrent) ausgeführte und verschränkt überwachte positive Signatur mit einem eindeutigen Ereignis (keine Epsilontransition) beginnen, wenn es sich nicht um einen antecedent Teil (Vorbedingung) der Signatur handelt. Andernfalls würde die zweite Subgeneration in jedem Fall fehlschlagen, da diese nach ihrer Erzeugung schalten würde und somit keine passive Subgeneration mehr ist. Beim nächsten Ereignis, das nicht mit dem als nächstes erwarteten Ereignis übereinstimmt, würde diese Subgeneration fehlschlagen.

	global	lokal
generationsunabhängig	✓	✓
Eingabegeneration	✓	✓
Subgeneration	-	✓

Tabelle 6.4: Geltungsbereich der Variablen im MPN.

- Referenzierte MPNs müssen wie MPNs im Allgemeinen mit einem eindeutigen Ereignis abschließen, wenn das aufrufende MPN durch eine EP-Variable blockiert wird. Ansonsten ist durch das Referenzsystem kein eindeutiger Abschluss der Signatur erkennbar.
- Rekursion und Schleifen in der Spezifikation des Referenzsystems müssen unter Beachtung der Zielplattform verwendet werden. Die Signaturen müssen für den Einsatz auf eingebetteten Systemen ein eindeutiges Abbruchkriterium aufweisen, um eine obere Grenze der Instanzen festlegen zu können.
- Die Auswertung der Plätze der Aktivierungsplatzmenge S_{act} findet immer nach der Ausführung des Makroschrittes statt. Hierdurch müssen Plätze, die sich in der Menge S_{act} befinden, sich an Stellen der Signatur befinden an denen nach einem Makroschritt ein Token liegen kann. Bei Plätzen mit Epsilontransitionen ohne Bedingung im Nachbereich haben diese keine Auswirkung zur Laufzeit des Monitors und sind somit ein Spezifikationsfehler.

6.3 VARIABLEN UND UMGEBUNG

Die vorher in Kapitel 5 eingeführte Umgebung E speichert Bindungen von Variablen zu Werten während der Ausführung von MPNs. Bisher wurde die Umgebung als Blackbox betrachtet, in der die Variablen gespeichert werden. Im Folgenden werden diese Umgebung und die Variablen genauer definiert.

Variablen der MPNs können in zwei orthogonalen Dimensionen nach ihrem Geltungsbereich kategorisiert werden (Tabelle 6.4). Die erste Dimension beschreibt die Beziehung zu den MPNs, wobei globale Variablen für alle MPNs im Monitor zugreifbar sind und lokale Variablen MPN-spezifisch sind. Die zweite Dimension beschreibt die Abhängigkeit zu den Generationen bzw. den MPN-Instanzen. Die Variablen können hierbei unabhängig oder spezifisch für Eingabe- oder Subgenerationen sein.

Insgesamt existieren fünf Geltungsbereichstypen, die in einer MPN-Beschreibung genutzt werden können. Globale Variablen, die subgenerationsspezifisch sind, sind nicht sinnvoll, da Subgenerationen per Definition mit einem MPN assoziiert sind.

Die in Abschnitt 5.3 und 5.5 verwendete Umgebung wird auf die hier eingeführte Definition von Variablen erweitert.

Definition 63 (Variablen und Umgebung). *Die Variablen* V *werden in einer Umgebung* E *zusammengefasst, über die der Zugriff auf Werte der Variablen stattfindet.*

V *ist eine endliche Menge von Variablenbezeichnern.*

$D^{\perp} = \{\perp\} \cup D$ *ist eine Menge von Werten mit* $\perp$ *für die Kennzeichnung undefinierter Werte.*

$E : V \times MPN \times G \to D^{\perp}$ *liefert zu einer Generation, einem MPN und einer Variablen den aktuellen Wert zurück. Für globale Variablen ist der Wert z. B. unabhängig von* G *und* MPN.

Für eine globale Variable $v \in V$ gilt somit $E(v, mpn, g) = E(v, mpn', g)$ wobei $mpn, mpn' \in MPN$ und für eine generationsunspezifische Variable $E(v, mpn, g) = E(v, mpn, g')$ mit $g \in G$.

Neben der Einteilung nach ihrem Gültigkeitsbereich können Variablen zusätzlich nach ihrem Einsatzzweck in der MPN-Spezifikation eingeteilt werden.

Definition 64 (Variablentypen). *Die Menge der möglichen Variablen eines Monitors* V *setzt sich aus folgenden Untermengen zusammen:*

$$V = V_{MPN} \cup V_{Par} \tag{6.4}$$

Hierbei werden die Variablen aus V_{MPN}*, die allgemein in den MPNs verwendet werden, und die als Parameter zusammen mit den Eingabeereignissen übergebenen Variablen aus* V_{Par} *unterschieden. Die durch Eingabeereignisse (auch zusammen mit Epsilonereignis) übergebenen Variablen aus* V_{Par} *sind immer global.*

VERWENDUNG VON VARIABLEN IN DEN MPN-SIGNATUREN MPNs sind deterministisch definiert. Diese Eigenschaft muss auch bei der Verwendung von Aktionen auf Variablen, die an Transitionen annotiert sind, beachtet werden. Der Determinismus ist in der Definition der MPNs jedoch nur für einzelne MPN-Instanzen sichergestellt. In einem Ereignisverarbeitungsschritt (Def. 38) ist dies realisiert, indem die Auswertung auf einer Kopie der Umgebung stattfindet. Anschließend wird durch die Zusammenführung der aufgetretenen Änderungen an der Umgebung eine neue Umgebungsinstanz erzeugt, die für die weiteren Schritte verwendet wird.

Außerhalb dieses Schrittes, wie der Verarbeitung eines Ereignisses in mehreren MPN-Instanzen, können die globalen Variablen von mehreren MPNs beeinflusst und ausgewertet werden. Somit kann es bei nicht vorgegebener Auswertungsreihenfolge der MPN-Instanzen während der Verarbeitung eines Ereignisses durch Veränderung der Umgebung zu nichtdeterministischen Verhalten kommen. Dies ist, falls nicht ausdrücklich gewünscht, durch entsprechende Modellierung der Signaturen zu verhindern.

Beispiel *Möglicher Nichtdeterminismus in Monitorspezifikationen*

Gegeben sind die folgenden Signaturen mit Annotationen an den Transitionen:

mpn1: e1, e2/a++, ...

mpn2: e2 [a>=2], ...

Die globale Variable *a* hat den Wert 1. Tritt nun das Ereignis *e2* nach Ereignis *e1* auf, hängt es von der Verarbeitungsreihenfolge von *mpn1* und *mpn2* ab, ob die Transition mit der Bedingung *[a>=2]* schaltet oder ob sich *mpn2* beendet.

Bei Subgenerationen, die auf einer MPN-spezifischen (lokalen) Variablen arbeiten, ist die Möglichkeit einer Übernahme der Änderung des Variablenwertes von einer Subgeneration zu nächsten Subgeneration jedoch gewünscht. Durch die festgelegte Abarbeitungsreihenfolge der Subgenerationen ist eine deterministische Ausführung sichergestellt. Variablen aus der Menge V_{Par} dürfen nur von außen auf Basis von Eingabeereignissen aktualisiert werden.

6.4 ZEIT IN MPNS

Bisher wurde die Zeit als Teil der Modellierung in MPNs nicht explizit behandelt. Zeitbedingungen und Timer lassen sich mit den vorhandenen Mitteln der MPN-Sprache umsetzen.

Globale und lokale Uhren werden durch die Verwendung von Epsilonereignissen mit der aktuellen Zeit als Parameterwert, die von außen dem Monitor in regelmäßigen Abständen übergeben werden, umgesetzt. Der aktuelle Wert dieser Uhren kann dann mittels Bedingungen und Aktionen in den MPNs verarbeitet werden.

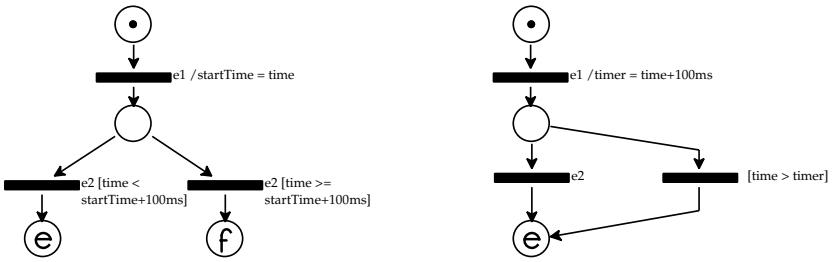
Auch zur Beschreibung von einfachen Signaturen kann ein Zeitbezug zur Erkennung ausbleibender erwarteter Ereignisse notwendig sein. Werden hierfür keine Zeitbedingungen eingeführt, wartet die MPN-Instanz im Zweifelsfall beliebig lange und erkennt nicht, dass ein Fehler, wie die Störung der Kommunikationsstrecke oder ein defekter Kommunikationspartner, aufgetreten ist.

Zeit in Monitor-Petrinetzen wird durch den Einsatz von Variablen in der Umgebung und zeitlichen Bedingungen an den Transitionen modelliert.

Definition 65 (Variablentypen (erweitert)). *Die Menge der möglichen Variablen eines Monitors V setzt sich aus folgenden Untermengen zusammen:*

$$V = V_{MPN} \cup V_{Par} \cup V_{Time} \tag{6.5}$$

Hierbei werden Variablen aus V_{MPN}, die allgemein in den MPNs verwendet werden, die als Parameter mit Eingabeereignissen übergebenen Variablen aus V_{Par} und Zeitvariablen aus V_{Time} unterschieden. Auch die Zeitvariablen können in ihrem Gültigkeitsbereich eingeschränkt werden.



a) b)

Abbildung 6.12: Beispiel (a) der Verwendung von Zeit in MPNs und (b) der Verwendung eines Timers

Beispiel *Zeit in MPNs*

Schon in der CARDME-Monitorspezifikation aus Abschnitt 6.1 in Abbildung 6.6 ist eine solche Zeitbedingung in der Signatur *DoS Attack* modelliert. Sie wird zur Messung der Zeit zwischen dem Auftreten verschiedener Ereignisse eingesetzt. Die Variablen t und *time* sind Elemente der Variablenmenge $V_{\mathtt{time}}$. Die Uhr wird durch die Variable *time* dargestellt, die von außen in regelmäßigen Abständen über Epsilonereignisse mit entsprechender Umgebung aktualisiert wird. Die Variable t wird durch eine Aktion an einer Transition auf die aktuelle Zeit gesetzt. Über eine Bedingung im späteren Verlauf der Signatur wird die vergangene Zeit ausgewertet.

Abbildung 6.12 zeigt zwei weitere einfache Beispiele für den Einsatz von Zeit in einem MPN. Auch hier ist *time* die Uhr, die von außen durch Epsilonereignisse aktualisiert wird. In Abbildung 6.12 a) wird eine globale Uhr (*time*) im MPN genutzt um zwischen positiven und negativen Ausgang zu entscheiden. Zunächst wird der Zeitpunkt zu den das Ereignis *e1* auftritt auf der Variablen *startTime* zwischengespeichert. Beim Auftreten des Ereignisses *e2* wird diese gespeicherte Zeit und der aktuelle Wert der Uhr in deiner Bedingung genutzt.

Signaturen, bei denen ein abschließendes Ereignis nicht eintritt oder nicht eintreten muss, führen dazu, dass die Überwachung einer Signaturinstanz nicht abgeschlossen werden kann, wenn das erwartete Ereignis nicht eintritt. Um zu bestimmen, ob noch auf dieses letzte Ereignis gewartet werden muss, kann hier eine Zeitbedingung eingesetzt werden. Diese Bedingung wird so modelliert, dass sie das Überspringen des optionalen Ereignisses zulässt, wenn die Bedingung zu wahr ausgewertet wird. In Abbildung 6.12 b) ist ein Timer modelliert. Dieser speichert die Deadline basierend auf der aktuellen Zeit auf der Variablen *timer*. Danach wird aus das optionale Ereignis *e2* gewartet. Zusätzlich ist eine Transition mit Bedingung modelliert, die bei Ablauf des Timers den Monitor beendet.

Durch die Aktualisierung der Uhren über Epsilonereignisse wird, wie auch bei der Übergabe anderer Werte an den Monitor, eine Überprüfung der Epsilontransitionen in den MPN-Signaturen ausgelöst. Diese Verarbeitung des Epsilonereignisses führt, wie in Abschnitt 5.3 definiert, zu einem nachfolgenden Schalten der

Epsilontransitionen in den MPN-Signaturen bis sich die Markierung des Netzes nicht mehr ändert.

Zusammenfassend wird Zeit in MPNs somit folgendermaßen abgebildet.

- Uhren werden über Variablen der Menge $V_{\mathtt{Time}}$ realisiert, indem von außen Epsilonereignisse diese Variable mit der aktuellen (System-)Zeit aktualisieren.
- Zeitbedingungen greifen auf diese Variablen zu und vergleichen die Zeit mit gespeicherten und berechneten Zeitwerten.
- Timer werden durch Setzen einer Zeitvariablen auf einen Wert der aktuellen Zeit, der um die Zeit erhöht wird, in der die Deadline liegt, gestartet. Die Erkennung des Ablaufens des Timers wird über Bedingungen realisiert, die bei jeder Aktualisierung der aktuellen Zeit von außen durch Epsilonereignisse ausgewertet werden.

Eine für alle Monitore gemeinsame Zeit, die über Epsilonereignisse von außen aktualisiert wird, kann auf dieselbe Art über eine Variable $v \in V_{\mathtt{Time}}$ realisiert werden. Diese Variablen können MPN- und/oder generationsübergreifend definiert werden, wodurch eine gemeinsame Zeit (Uhr) für alle Signaturen oder spezifischere Uhren zur Verfügung stehen.

Ist dieses Konzept nicht ausreichend, können auch externe Timer eingesetzt werden. Diese können u. a. eine genauere Auflösung liefern, ohne das der Monitor durch viele Aktualisierungen der Zeit von außen, die ebenfalls Rechenzeit benötigen, blockiert wird. Hierzu können externe eigenständige Timer über Aktionen der MPNs gestartet werden, die bei Erreichen eines Zeitpunktes ein Ereignis dem Monitor übergeben, auf das dann im Monitor reagiert werden kann.

6.5 AUSFÜHRBARE MPNS

Während der Überwachung eines Systems kommt es vor, dass ein Fehler im Ablauf des Systems oder in der Kommunikation festgestellt wird. Im Fall eines erkannten Fehlers muss auf das überwachte System eingewirkt werden, sodass es wieder in einen stabilen sicheren Zustand überführt wird. Hierzu wurde in Abschnitt 6.2 im Rahmen des Referenzsystems der MPNs ein nach dem Fehlschlagen einer MPN-Instanz (aus MPN^{+} oder MPN^{-}) nicht nebenläufig ausgeführter Erweiterungspunkt definiert. Dieser referenziert positive MPNs aus der Menge MPN^{+}, in denen Gegenmaßnahmen spezifiziert sind.

Im Allgemeinen Fall eines zentralen oder verteilten Monitors in dem der Monitor die Ereignisse, die im zu überwachenden System eingetreten sind, überwacht, muss das System wieder in einen sicheren Zustand gebracht werden. Hierzu werden in den Gegenmaßnahmen Aktionen spezifiziert, die z. B. Teile des Systems zurücksetzen und entsprechende Teile des Monitors (Signaturen) neu starten. Der Monitor startet für diese Signaturen mit einer leeren Eingabesequenz.

Bei Monitoren, die als Wrapper um zu überwachende Komponenten umgesetzt wurden und hierdurch Ereignisse (Nachrichten) bis zur Überprüfung zurückhalten können, ist die Auswirkung einer in die Kommunikation eingreifenden Ge-

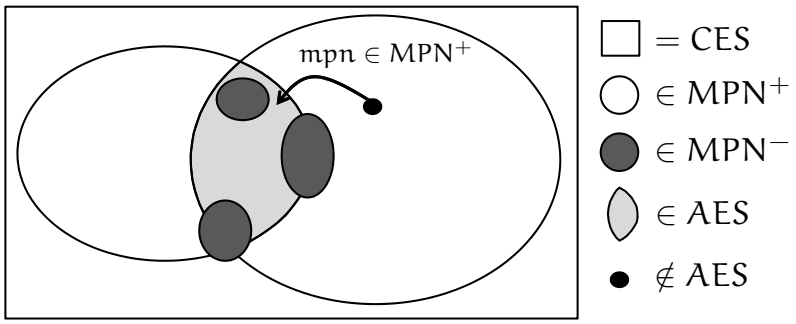


Abbildung 6.13: Anpassung der Kommunikation zum Erreichen einer akzeptierten Ereignissequenz

genmaßnahme beispielhaft in Abbildung 6.13 dargestellt. Es wurde eine Ereignissequenz erkannt, die nicht in der Menge der akzeptierten Ereignissequenzen (AES) liegt. Ziel ist es nach der Erkennung einer solchen Abweichung das System in einen stabilen Zustand und die Kommunikation in den Bereich der akzeptierten Ereignissequenzen (AES) zu überführen. Hierdurch können negative Auswirkungen auf die Kommunikationspartner verhindert werden. Eine solche Gegenmaßnahme wird durch die Anwendung der ausführbaren MPNs durchgeführt und z. B. eine fehlerhafte Nachricht abgeändert oder unterdrückt.

Beispiel *Ausführbares MPN*

In Abbildung 6.6 wurde das sehr einfache auszuführende MPN *Close Connection* vorgestellt. Dieses besteht nur aus der auszuführenden Aktion *Ignore(Vehicle.id)*, die dazu führt, dass die Nachrichten dieses Fahrzeugs zukünftig von der Mautbrücke ignoriert werden. Durch die Verbindung des ausführbaren MPNs mit dem negativen MPN *DoS Attack* über einen Erweiterungspunkt des Referenzsystems der MPNs, wird die Situation definiert, an dem das MPN ausgeführt wird. Sobald die im Basis-MPN spezifizierte negative Ereignissequenz erkannt wird, wird das referenzierte MPN ausgeführt.

Ausführbare MPNs sind in ihrer Syntax und Semantik identisch zu denen, die für die Laufzeitüberwachung verwendet werden. Zur Spezifikation von Gegenmaßnahmen steht die volle Ausdrucksstärke der MPN-Sprache zur Verfügung. Grundlegend können zur Modellierung alle Elemente der MPN-Sprache inklusive Bedingungen und Aktionen an den Transitionen eingesetzt werden. Falls die Ausführung zusätzlich abhängig von Eingabeereignissen sein soll, können auch diese an den Transitionen annotiert werden. Die Ereignisse, die annotiert werden, dienen, wie auch bei allen anderen MPNs, nur zur Überwachung und nicht dazu Ereignisse in das System oder an den Monitor zu schicken. Zum Senden von Ereignissen und zur Beeinflussung des Systems werden Aktionen eingesetzt.

Durch Vorbedingungen und Bedingungen in den MPNs selbst, die auf die Variablen in der Umgebung zugreifen, können auch verschiedene sich ausschließende oder ergänzende ausführbare MPNs als Reaktion auf einen erkannten Fehler spezifiziert werden. Die passende Gegenmaßnahme kann zusätzlich auch von der

Markierung der fehlgeschlagenen (*failed*) MPN-Instanz abhängen. Diese Markierung lässt indirekt auf den Zustand des überwachten Systems schließen. Hierzu kann im Referenzsystem der MPNs auch die Ausführung der modellierten Gegenmaßnahmen durch die Belegung der Aktivierungsplätze S_{act} bedingt werden.

Bei der Aktivierung eines ausführbaren MPNs ist es nötig ein Epsilonereignis diesem zu übergeben, damit dieses mit der Ausführung beginnt. Ab diesem Zeitpunkt wird das MPN bis zum Ende, das durch Erreichen eines Terminalplatzes oder das Nichtschalten auf ein Eingabeereignis ausgelöst werden kann, durchgeführt.

Im Allgemeinen werden zur Spezifikation von Gegenmaßnahmen einfachere Signaturen als für die Überwachung verwendet werden. Durch das Ändern des Systemzustandes durch Aktionen der Gegenmaßnahmen kann es dazu kommen, dass auch schon begonnene Signaturüberwachungen fehlschlagen. Hier ist zu überlegen, ob bestimmte MPN-Instanzen in einem solchen Fall in eine passende Ausgangsmarkierung gebracht werden müssen.

6.6 Aufteilung von Monitoren auf verschiedene Steuergeräte

Bisher wurden Monitore immer als zentrale Komponenten gesehen, die alle Ereignisse eines Systems zur Verfügung gestellt bekommen. Dieses Szenario ist jedoch in der Praxis häufig nicht gegeben. Im Automotiveumfeld und speziell in AUTOSAR wird großen Wert auf die Verteilbarkeit von Softwarekomponenten auf verschiedene Elektronische Steuergeräte (ECUs) gelegt. Diese Verteilung findet oft erst spät im Entwicklungsprozess statt. Werden Spezifikationssprachen wie eLSCs zur Beschreibung verwendet, bilden sie Zusammenhänge zwischen verschiedenen miteinander kommunizierenden Komponenten ab. Da MPNs als Zwischensprache entwickelt wurden, muss es auch möglich sein, alle Komponenten umfassende Spezifikationen, die zu MPNs übersetzt wurden, vor der Monitorgenerierung an die im Zielsystem gegebenen Anforderungen anzupassen.

Das Ziel dieses Abschnittes ist die Aufteilung einer vollständigen Monitorbeschreibung auf ein verteilt laufendes System. Hierbei ist es notwendig, dass auch in der Quellspezifikation (z. B. eLSCs) eine Zuordnung der Signaturteile auf Teilsysteme des zu überwachenden Systems vorhanden ist, die in die Zwischensprache übertragen werden kann. In eLSCs kann dies über die einzelnen Lebenslinien und damit über die Objekte bzw. Komponenten, die auf verschiedene Systeme verteilt werden, stattfinden. Auch die Monitore müssen hierzu in Teilmonitore aufgeteilt werden.

Beispiel *Zuordnung von Monitoren auf verteilte Zielsysteme*

Eine Signatur, die mehrere modellierte miteinander kommunizierende Teilsysteme (*A*, *B* und *C*) einbezieht, kann auf verschiedene Weise auf zwei ECUs verteilt werden. Die Aufteilung der Signaturen hängt von der Verteilung der zu überwachenden Komponenten des Zielsystems ab. So kann u. a. eine Aufteilung der Signaturen in Teilsignaturen, die z. B. Monitorteile für {*A*,*B*} und {*C*} oder für {*A*} und {*B*,*C*} beinhalten, stattfinden. Der Monitorgenerierungsprozess

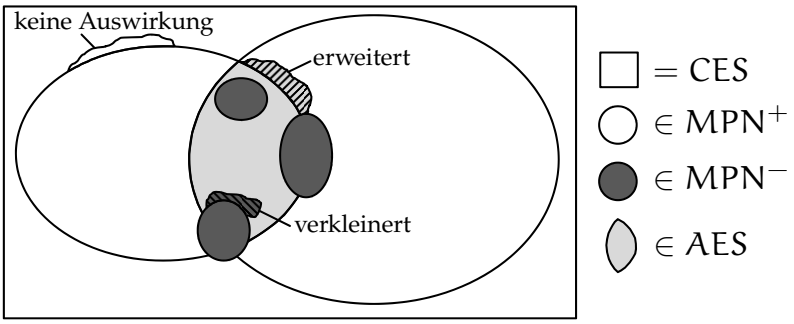


Abbildung 6.14: Informationsverlust durch Trennung der MPNs

muss, um verteilte Systeme zu unterstützen, eine Aufteilung von Gesamtsignaturen in beliebige Teilmonitore unterstützen.

Eine direkte Aufteilung, indem die MPNs an den Synchronisationsplätzen getrennt werden, führt zu einem starken Informationsverlust, da Zusammenhänge wie das Senden einer Nachricht vor dem Empfangen nicht mehr überprüft werden. Abbildung 6.14 zeigt schematisch das auftretende Problem, das beim Trennen ganzheitlich modellierter MPN-Signaturen auftritt, wenn kein Kommunikationskanal zwischen den Teilmonitoren zur Verfügung steht. Geteilte Spezifikationen, in denen keine Synchronisation der entstandenen Teilsignaturen mehr stattfindet, führen zu einer Veränderung der Menge der akzeptierten Ereignissequenzen (AES). Durch aufgetrennte MPNs aus der Menge MPN^{+} kann, bei Überlappung mit anderen MPNs aus MPN^{+}, die Menge der AES erweitert werden, wohingegen aufgetrennte MPNs aus der Menge MPN^{-} diese verkleinern können. Änderungen, die sich in Bereichen außerhalb der Schnittmenge der Signaturen aus MPN^{+} befinden und damit nicht zu zusätzlichen AES führen, haben keine Auswirkungen auf die Genauigkeit des Monitors.

Beispiel *Aufteilung an Synchronisationsplätzen*

Abbildung 6.15 zeigt die Aufteilung der Signatur *CARDME Local* an den Synchronisationsplätzen ohne das die Synchronisation erhalten wird. Hierbei wurde der Teil, der die Unterscheidung zwischen vorausgehendem (antecedent) und nachfolgenden (consequent) Teil der Signatur festlegt, für beide Seiten dupliziert, da er für beide Seiten von Interesse ist. Beim Trennen gehen jedoch Informationen der Gesamtsignatur verloren. Zum einen werden Bedingungen, die festlegen, dass zuerst eine Nachricht gesendet werden muss, bevor sie empfangen wird, nicht mehr überprüft. Zum anderen werden Synchronisationen zwischen Pre- und Mainchart nicht mehr gleichzeitig in beiden Teilsignaturen durchgeführt, was dazu führen kann, dass zum Beispiel das Ereignis *recv.PresRequest* verarbeitet wird bevor *recv.InitResponse* aufgetreten ist. Dies sind beides Fälle, in denen die Menge an akzeptierten Ereignissequenzen erweitert wird, da es sich um eine Signatur aus MPN^{+} handelt.

Des Weiteren führt dies dazu, dass sich eine Teilsignatur noch im vorausgehenden Teil der Signatur befindet, während der andere Teil sich schon im

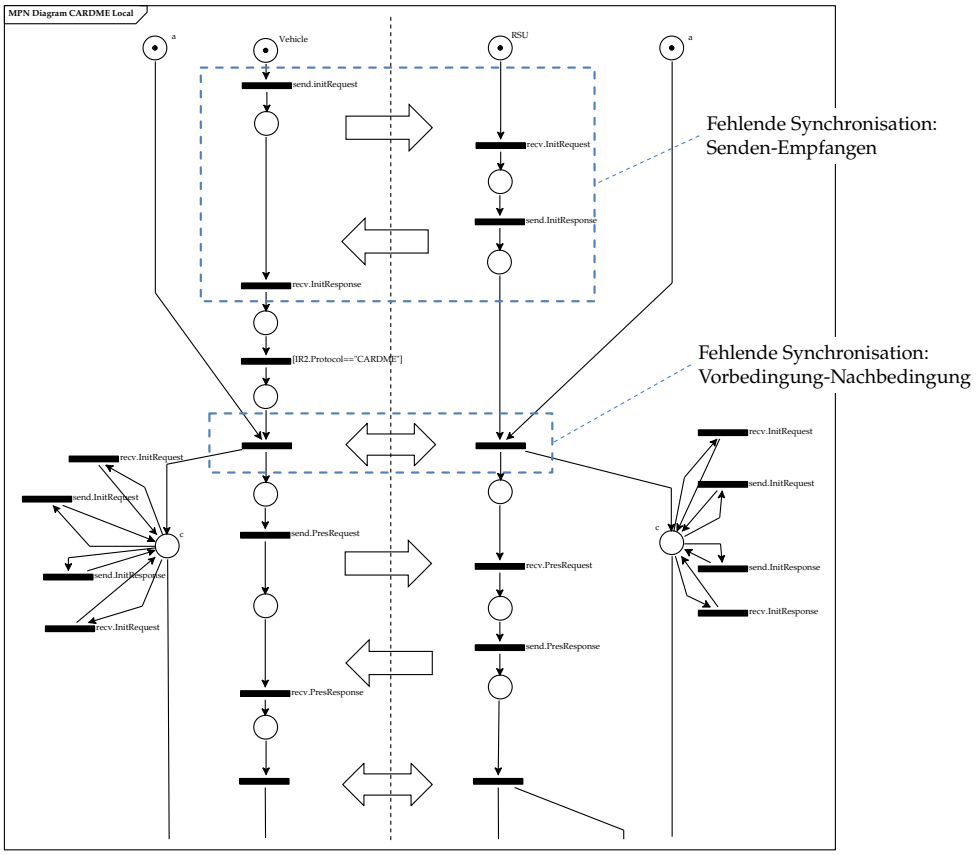


Abbildung 6.15: Auftrennung einer Signatur

nachfolgenden befindet. Falls ein Teil der Signatur sich vorzeitig beendet, da keine seiner Transitionen geschaltet hat oder ein Terminalplatz erreicht wurde, würde der andere Teil weiter ausgewertet werden. Bei der Synchronisation auf einen Terminalplatz entsteht dasselbe Problem, das auch bei Synchronisationstransitionen auftritt. Ein Teil des Monitors, der sein letztes Ereignis schon abgearbeitet hat, würde sich beenden, während der andere Teil noch auf sein letztes Ereignis wartet.

Um diesen Informationsverlust zu verhindern, ist eine Synchronisation dieser Informationen über einen Kommunikationskanal notwendig. Zur Aufteilung wird deshalb eine Zuordnung zwischen Objekten und Plätzen in den Teil-MPNs benötigt. Hierzu bekommen alle Plätze eine einzigartige Identifikationsnummer, über die Zusammenhänge (Plätze die in beiden Teilmonitoren existieren) zwischen den Teilmonitoren erkannt werden können. Beim Aufteilen gibt es Plätze, die direkt zu einem Objekt des eLSCs zuordenbar sind und Plätze, die für ein Objekt von Interesse sind. Diese Synchronisationsplätze, die für den Empfänger von Interesse sind, werden über den Kommunikationskanal synchronisiert. Plätze, die beschreiben, ob eine MPN-Instanz sich in einer Vorbedingung (antecedent) oder im Hauptteil (consequent) einer Signatur befindet, sind für alle Teilmonitore von Bedeutung und werden in alle Teilmonitore übernommen.

Zusammenfassend lassen sich damit in Bezug auf die Aufteilung von Signaturen drei Arten von Plätzen unterscheiden:

- Plätze, die einem Ursprungsobjekt zuzuordnen sind
- Plätze, die für andere Objekte von Interesse sind
- Plätze, die für alle Objekte von Interesse sind

In den folgenden Beispielen wird die Aufteilung an den bei der eLSC-zu-MPN-Transformation entstehenden typischen Mustern vorgestellt.

Beispiel *Aufteilung einer asynchronen Nachricht*

Abbildung 6.16 zeigt die Aufteilung einer asynchronen Nachricht zwischen den Kommunikationspartnern *RSU* und *Vehicle*, wie sie im eLSC modelliert ist. Hierbei wird an dem Synchronisationsplatz, der zu *RSU* gehört und für *Vehicle* von Interesse ist, getrennt. Dieser Platz wird in beiden Teilen der Signatur erstellt und hierdurch eine Synchronisierung des Platzes vom Ursprungsobjekt zum Interessentenobjekt realisiert. Die Synchronisierung des gemeinsamen Platzes kann nach dem Schalten der ersten Transition (*send.PresReq*) stattfinden, da erst bei der Verarbeitung des nächsten Ereignisses (*recv.PresReq*) der Platz im Vorbereich der Transition belegt sein muss.

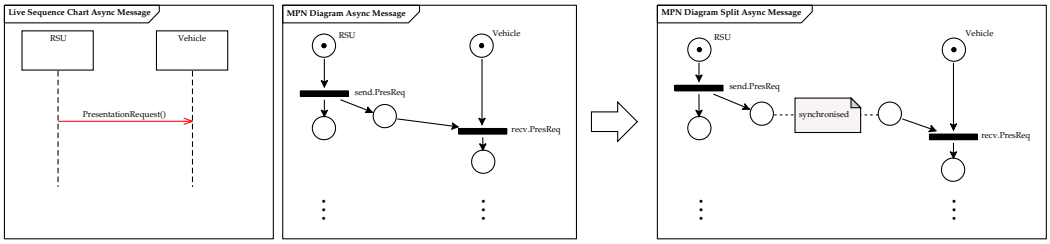


Abbildung 6.16: Aufteilung einer asynchronen Nachricht

Ein zweites Muster, das in den Signaturen in der MPN-Sprache vorkommt, ist eine Synchronisation, die aus der Übersetzung von synchronen Nachrichten, instanzübergreifenden Bedingungen oder die aus Übergängen zwischen Teilsignaturen entsteht.

Beispiel *Aufteilung einer Synchronisation zwischen Pre- und Mainchart*

In Abbildung 6.17 ist die Aufteilung einer Synchronisation zwischen Pre- und Mainchart, bei drei Kommunikationspartnern, abstrakt dargestellt. Hierbei wird jede Änderung eines Platzes im Vorbereich einer Transition in der Teilsignatur des Ursprungsobjektes zu den korrespondierenden Plätzen der Teilsignaturen der Interessentenobjekte übertragen. Eine synchrone Nachricht führt zu demselben Muster mit nur zwei Teilnehmern an der Transition.

Die in den beiden Beispielen vorgestellten Regeln zur Trennung von MPNs reichen aus, um Signaturen so zu trennen, dass verteilte Monitore generiert werden

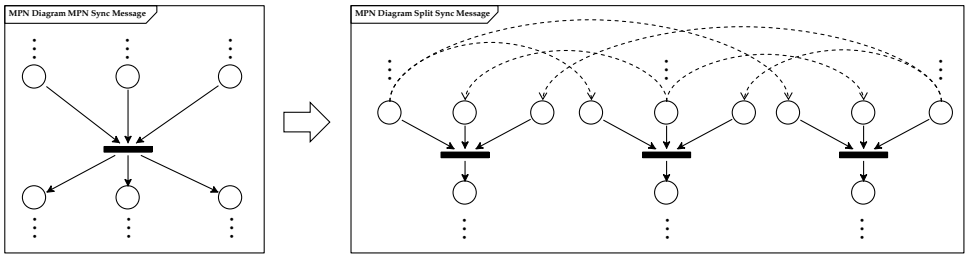


Abbildung 6.17: Aufteilung einer Synchronisation

können. Dies ist auf die MPN-Semantik, die in jedem Makroschritt ein Eingabeereignis verarbeitet und auf die in der Übersetzung aus den eLSCs vorhandenen Zuordnung der Elemente zu Objekten zurückzuführen.

Zusammenfassend lassen sich folgende Ereignisse hervorheben, die synchronisiert werden müssen:

- Erstellen und Löschen eines Token auf Synchronisationsplätzen
- Vorzeitige Abbrüche eines Teil-MPNs (auch durch zusätzliche Terminalplätze an Bedingungen)
- Zuweisung auf instanzübergreifende Variablen

Instanzübergreifende Variablen führen zu einem weiteren Kommunikationsoverhead, da diese an die anderen Teilmonitore übertragen werden müssen. Die Synchronisierung kann wie bei den anderen Ereignissen auch am Ende des Makroschrittes durchgeführt werden, da die aktuellen Werte erst bei der Verarbeitung des nächsten Ereignisses gültig sind. Bei Einsatz von Subgenerationen muss die Synchronisation nach jeder Verarbeitung einer Subgenerationen durchgeführt werden, um aktuelle Variablenwerte bei der Verarbeitung der nächsten Subgeneration zur Verfügung zu haben.

In Signaturen, die Timern und Zeitbedingungen verwenden, muss eine gemeinsame Basiszeitquelle vorhanden sein. Diese aktuelle Zeit kann zum Beispiel, wie in Abschnitt 6.4 beschrieben, über ein Epsilonereignis als Datenwert übertragen und als Variable (Uhr) in der Umgebung beider Teil-MPNs gespeichert werden.

Die Ausführungsreihenfolge der Teilmonitore ist unabhängig von der modellierten Signatur und findet folgendermaßen statt:

1. Schalten des Teilmonitors mit dem aktuellen Ereignis.
2. Wenn ein Ursprungsplatz, der für andere Teilmonitore von Interesse ist, belegt wird:
   a) wird das Token in den anderen Teilmonitor auf den entsprechenden Platz übertragen und aus dem Ursprungsplatz entfernt.
   b) geänderte Variablen werden übertragen.
   c) der entsprechende aktualisierte Teilmonitor mit einem Epsilonereignis angestoßen, was dazu führt, dass er Epsilontransitionen schaltet, bis er sich in einem stationären Zustand befindet.

Zur Reduktion des Kommunikationsoverheads ist folgende Einschränkung bei der Verwendung globaler Variablen gegeben. Globale und MPN-spezifische Variablen, die in einem Teilmonitor (Lebenslinie) zugewiesen werden und in einem anderen Teilmonitor (einer anderen Lebenslinie) verwendet werden, sind nur erlaubt, wenn ihnen eine Synchronisation über einen Synchronisationsplatz vorausgeht. Ansonsten müssen solche Bedingungen, die die Variable setzende Instanz über eine aufgeteilte Synchronisationstransition mit einschließen.

Mit dieser vorgestellten Verarbeitungsreihenfolge für jede Teilsignatur ist die definierte Syntax und Semantik der MPN-Sprache ausreichend für die verteilte Überwachung ganzheitlich modellierter Signaturen. Ein Ereignis kann immer nur ein Sende- oder ein Empfangsereignis sein, wodurch eine Verarbeitung der beiden Ereignisse im selben Makroschritt der MPNs nicht auftreten kann. Synchrone Ereignisse werden wie im Beispiel gezeigt in zwei unabhängige Transitionen bei der Trennung übersetzt, die in beliebiger Reihenfolge schalten können. Selbst zwei aufeinanderfolgende Transitionen, die dasselbe Ereignis annotiert haben und über einen Synchronisationsplatz verbunden sind, dürfen nach der MPN-Semantik nicht im selben Makroschritt schalten. Durch das Epsilonereignis, dass dem der veränderten Teilsignatur übergeben wird, schalten auch Epsilontransitionen, die erst durch die Variablenwertänderung oder durch die Synchronisation von Platzbelegungen aktiviert werden. Hierdurch ist die Reihenfolge der Teilmonitore bei Signaturen in der MPN-Sprache weiterhin frei wählbar und das Ergebnis der MPN-Instanzen deterministisch. Die MPN-Sprache muss für die Aufteilung nicht angepasst werden.

UNABHÄNGIGE TEILMONITORE Der angesprochene Informationsverlust bei der Trennung von Monitoren kann jedoch auch gewollt sein, wenn eine gesamtheitliche Signatur Instanzen beschreibt, auf die im zu überwachenden System kein Zugriff besteht. Dies kann zum Beispiel eine abstrakte Umgebungsinstanz sein, die in eLSCs zur Modellierung der Quelle von Nachrichten genutzt wurde.

Beispiel *Unabhängiger Teilmonitor*

In vielen Fällen kann es vorkommen, dass Objekte die in der Signaturbeschreibung enthalten sind, im Zielsystem nicht vorhanden sind oder dass die entsprechenden Ereignisse für den Monitor nicht zur Verfügung stehen. Die Signatur in Abbildung 6.15 zeigt die Kommunikation zwischen *RSU* und *Vehicle*. Soll nun ein Monitor generiert werden, der auf dem Fahrzeug läuft, kann dieser nicht auf die Ereignisse der *RSU* zugreifen. In diesem Fall muss die *RSU* als Umgebung angesehen werden, und zählt nicht zu den Objekten, die von Interesse sind. Es wird ein Teilmonitor generiert, der nur aus der auf der rechten Seite gezeigten Teilsignatur besteht. Dabei kommt es aufgrund der durch die Verteilung fehlenden Informationen zu der vorher in Abbildung 6.14 gezeigte Vergrößerung der Menge akzeptierten Ereignissequenzen (AES).

Für diese Quelle kann zwar ein Teilmonitor generiert, dieser aber nicht während der Laufzeit des Systems ausgeführt werden. Eine Ausführung auf demselben System wie der zweite Teilmonitor und eine Abbildung der Empfangsereignisse auf Sendeereignisse würde keinen Mehrwert und eine höhere Laufzeit des Mo-

nitors mit sich bringen. Hier ist es sinnvoll den Teil, der die Umgebung abbildet, bei der Monitorgenerierung vollständig zu ignorieren und einen Monitor nur für die empfangende Instanz zu generieren.

AUFTEILUNG MIT REFERENZSYSTEM Die Aufteilung von Monitorspezifikationen kann grundsätzlich auch für Spezifikationen, die das Referenzsystem nutzen, verwendet werden. Hierbei können die Monitorspezifikationen wie vorher beschrieben einzeln aufgeteilt werden. Für die Ausführung muss jedoch auf die durch das Referenzsystem vorgegebene Reihenfolge der Ergebnisauswertung geachtet werden. Hierzu und zur Aktivierung von über Erweiterungspunkten referenzierte Teilmonitore, ist eine zusätzliche Kommunikation zwischen den aufgeteilten Teilmonitoren notwendig.

Wie schon beschrieben, führt die im Referenzsystem eingeführte globale Auswertung der zusammenhängenden Teilmonitore zu einem größeren Kommunikationsoverhead im Fall der Verteilung, da weitere Synchronisationsereignisse versendet werden müssen. Es ist notwendig die verschiedenen Teilmonitore zusammenhängend zu verarbeiten, um die Abarbeitungsreihenfolge, wie sie im Referenzsystem vorgestellt wurde, sicherzustellen.

Im Allgemeinen ist es sinnvoller Monitore, die verteilt werden sollen, ohne das Referenzsystem zu spezifizieren. Dies führt zwar eventuell zu größeren Spezifikationen der Teilmonitore und mehr Speicherbedarf auf den Komponenten, vermeidet jedoch zusätzliche Kommunikation zwischen den Teilmonitoren.

SCHLUSSFOLGERUNG In diesem Abschnitt wurde gezeigt, wie aus ganzheitlichen Signaturen in der MPN-Sprache verteilte Monitore generiert werden können, ohne die MPN-Sprache anpassen zu müssen. Des Weiteren können aus den MPN-Spezifikationen auch unabhängige Teilmonitore generiert werden, die auf Teilsystemen eingesetzt werden können. Hierzu müssen die MPN-Signaturen Informationen über die Zugehörigkeit der Elemente zu den Teilsystemen enthalten, die gegebenenfalls aus der Quellspezifikation gewonnen werden können. Die Vorteile eines solchen verteilten Überwachungsansatzes sind:

- Die lokale Überwachung von Teilsystemen ist möglich.
- Es bestehen Eingriffsmöglichkeiten in den ein-/ausgehenden Datenstrom mittels Wrapper.
- Es entstehen kleinere Monitore direkt an der Ereignisquelle.

Jedoch sind kritische zu beachtende Punkte u. a.:

- Der Monitor hat ohne Synchronisation nur Teilsicht auf das System.
- Bei der Synchronisation muss Sendereihenfolge der Ereignisse über das Kommunikationsmedium übereinstimmen und die Übertragung muss sichergestellt werden.
- Es entsteht eine höhere Belastung des Kommunikationskanals, da Synchronisationspakete Priorität haben müssen.

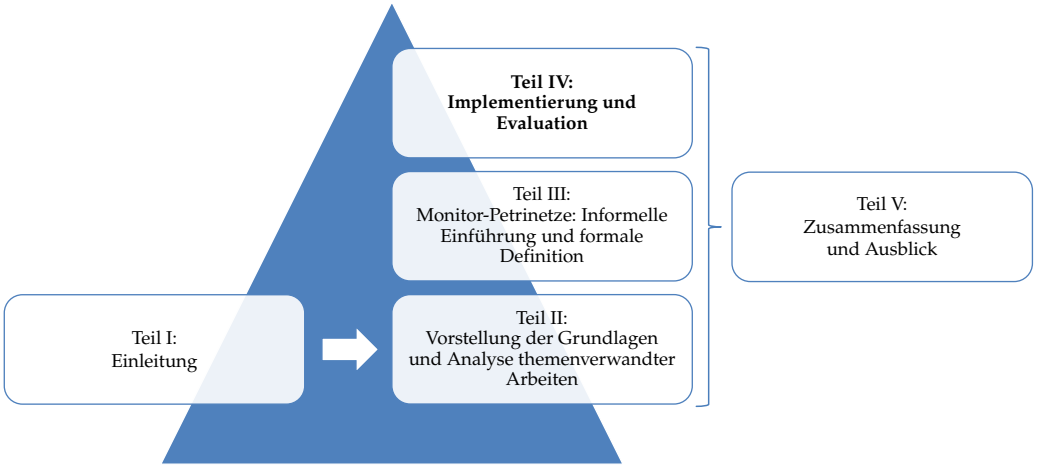
Insgesamt muss die Kommunikation in die Laufzeit des Monitors mit einberechnet werden. Diese ist je nach Kommunikationsmedium, wie in Abschnitt 9.3 betrachtet, im Gegensatz zu den Ereignisverarbeitungszeiten relativ groß.

Durch zusätzliche Garantien des Kommunikationskanals kann ggf. der Kommunikationsoverhead, der durch die Synchronisation entsteht, reduziert werden. So können, wenn auch für Nachrichten des Systems die Übertragung garantiert ist, Synchronisationen, die nicht für die Übertragung von Variablenwerten benötigt werden, ohne Informationsverlust entfernt werden.

Die Aufteilbarkeit der MPNs hängt von ihrer Modellierung und von der Existenz der benötigten Zuordnungen zu den Teilsystemen in der Quellspezifikation ab. Im Fall von eLSCs als Spezifikationssprache erlaubt es die Abbildung in die MPN-Semantik plattformspezifische Monitore auch für verteilte Systeme zu generieren.

Teil IV

IMPLEMENTIERUNG UND EVALUATION



7

IMPLEMENTIERUNG

Nach der Formalisierung der Monitor-Petrinetz-Sprache (MPN) in verschiedenen Ausbaustufen wird in diesem Kapitel die Umsetzung des Prozesses zur Monitorgenerierung in der MBSecMon-Tool-Suite, des MPN-Editors und der generierten Monitore thematisiert. Hierbei werden in Abschnitt 7.1 zunächst der Gesamtprozess und seine Implementierung beschrieben. Abschnitt 7.2 stellt die Funktionalität des MPN-Editors vor, der in das UML-Modellierungswerkzeug *SparxSystems Enterprise Architect* integriert wurde. Darauf folgt in Abschnitt 7.3 ein Überblick über die vor der Generierung von Monitorcode aus Monitor-Petrinetzen durchgeführten Schritte. Es werden beispielhaft einige Optimierungen an den MPN-Spezifikationen vorgestellt, die den Umfang der Spezifikationen reduzieren können. Zur Codegenerierung werden zusätzlich zur Spezifikation in Form von MPNs weitere Informationen über die Zielplattform und die Zielsprache benötigt. Die Gewinnung und die Einbindung dieser plattformspezifischen Informationen in den Generierungsprozess wird vorgestellt. Anschließend wird die Umsetzung der Trennung von MPNs zur Verteilung auf verschiedene Steuergeräte und auf die Übersetzung auf verschiedene Plattformen eingegangen. Abschließend beschreibt Abschnitt 7.4 die umgesetzten Varianten der Codegenerierung für verschiedene Zielsprachen und Zielplattformen, bevor in Kapitel 8 auf die eigentliche Codegenerierung detailliert eingegangen wird.

7.1 MBSecMon-Tool-Suite

In diesem Abschnitt wird der entwickelte Prototyp der MBSecMon-Tool-Suite, der den Prozess, in dem die Monitor-Petrinetz-Sprache als Zwischensprache Verwendung findet, vorgestellt. Das Hauptaugenmerk dieses Abschnitts liegt auf dem MPN-spezifischen Teil des Prozesses und nicht auf der Modellierung der Signaturen in der Spezifikationssprache, die in [Pat14] behandelt wird.

Abbildung 7.1 zeigt das UML2-Komponentendiagramm der MBSecMon-Tool-Suite. Die Quellspezifikation wird dem Rahmenwerk in Form einer XMI-Datei übergeben. Diese Spezifikation kann aus einer beliebigen Modellierungsumgebung stammen. Im Prototyp wird die Quellspezifikation in der MBSecMonSL, die aus eLSCs und MUC-Diagrammen besteht, in einem grafischen Editor, der als Add-in für das UML2-Modellierungswerkzeug *SparxSystems Enterprise Architect* umgesetzt ist, spezifiziert und als Ecore-XMI-Datei dem Prozess zur Verfügung gestellt. Wie auf der rechten Seite dargestellt ist, sind alle Repositories für die Sprachen und die Transformationen zwischen Modellinstanzen der verwendeten Sprachen mit dem Metamodellierungswerkzeug *eMoflon* [ALPS11] modelliert und

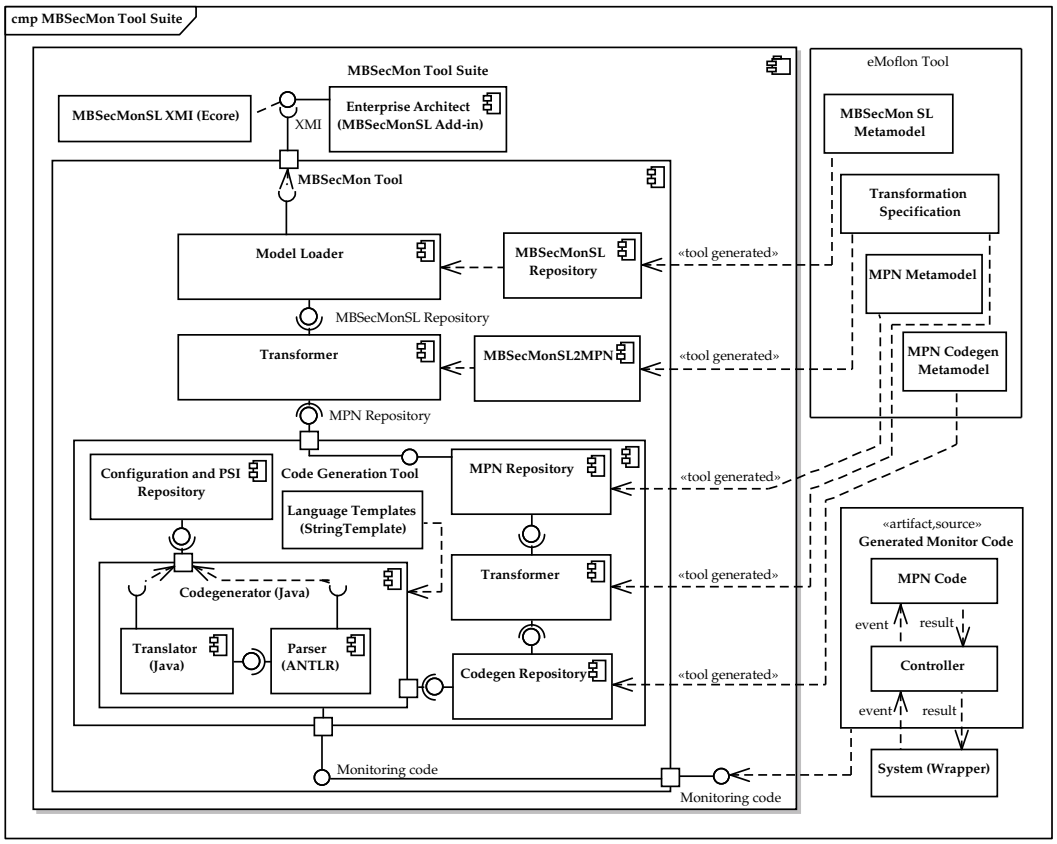


Abbildung 7.1: Architektur der MBSecMon-Tool-Suite

generiert. Hierbei wurde die Metamodellierungssprache *Ecore* für die Spezifikation der Sprach-Repositories und die Transformationssprache *Story Driven Modelling* (SDM) für die Transformation zwischen der MBSecMonSL und den MPNs verwendet. Aus diesen Spezifikationen wird Java-Code generiert, der die Transformationen zwischen den Modellinstanzen in den Repositories durchführt.

Die Quellspezifikation als XMI-Datei wird in das MBSecMon-Repository geladen und mithilfe des generierten Transformationscodes in ein MPN-Modell übersetzt. Diese Transformation muss für jede zu unterstützende Spezifikationssprache erstellt werden. Die daraus entstehende MPN-Repräsentation der Signaturen dient anschließend als Eingabe für den Codegenerator des MBSecMon-Tools.

Der untere Teil von Abbildung 7.1 (*Code Generation Tool*) zeigt die Komponenten des Codegenerierungsprozesses zusammen mit den benötigten Eingabedaten. Eine Konfigurationsdatei, die den Generierungsprozess steuert, wird zusammen mit plattformspezifischen Informationen (PSI) in das *Configuration and PSI Repository* geladen. Die PSI beinhaltet Daten, die aus den MBSecMon-Spezifikationen und weiteren Systemmodellen im Modellierungswerkzeug gewonnen werden. Sie werden bei Bedarf mit zusätzlichen PSI durch den Entwickler oder durch vorliegende Abbildungen auf die Zielplattform ergänzt.

Im ersten Schritt übersetzt der *Transformer* die Modelle, die sich im MPN-Repository befinden, in ein für die Codegenerierung spezialisiertes Repository

(*Codegen Repository*), das nur die für die Codegenerierung benötigten Daten enthält. Hierfür wird das Modell optimiert, indem redundante Plätze und Transitionen entfernt werden. Abhängig von der Konfiguration, werden in diesem Schritt die MPN-Spezifikationen für die Generierung verteilter Monitore aufgeteilt und reduziert. Des Weiteren werden Validitätsüberprüfungen wie die Analyse auf tote Transitionen und das Einhalten der syntaktischen MPN-Regeln durchgeführt. Diese Optimierungen werden in Abschnitt 7.3.2 genauer beschrieben.

Darauf folgend wird die Codegenerierung nach dem Prinzip der *Model Driven Architecture*[1] durchgeführt. Mit Hilfe von *StringTemplates*[2], die durch Javacode auf Basis der optimierten MPNs im *Codegen Repository* gefüllt werden, wird der Monitorcode für die Zielplattform generiert. Annotationen an den Transitionen wie Ereignisse, Bedingungen und Aktionen müssen in die Zielsprache übersetzt werden und an die Konventionen der Zielplattform angepasst werden. Der Codegenerator liest die Annotationen mit einem Parser ein, der aus ANTLR-Grammatiken[3] generiert wurde, und übersetzt diese mithilfe der im PSI-Repository vorhandenen Informationen in eine zielplattformkonforme Repräsentation. Dieser Code fließt in der Codegenerierung an den entsprechenden Stellen der Templates ein.

Für die Codegenerierung wird ein simulationsbasierter Ansatz verwendet, der die Struktur der MPNs direkt in Code abbildet. Durch die explizite Spezifikation mit wenigen verschieden Elementen mit einfacher Semantik in Form von MPNs ist die Codegenerierung sehr geradlinig und einfach. In Abbildung 7.1 unten rechts ist die generelle Struktur des generierten Monitors dargestellt. Für jedes MPN wird eine Komponente in der entsprechenden Zielsprache generiert (*MPN Code*). Für jedes MPN-Element (Plätze und Transitionen) wird ein Codefragment generiert, das in eine Kontrollstruktur eingefügt wird. Der entstandene Code für ein MPN wertet selbstständig den aktuellen Zustand seiner MPN-Instanzen (Generationen) aus. Zusätzlich wird ein *Controller* generiert, der die Schnittstellen nach außen, die Verarbeitung der Eingaben (Ereignisse und Parameterwerte) und die Ansteuerung der einzelnen MPNs übernimmt.

Die Codegenerierung wurde prototypisch für Java, C und VHDL implementiert. Diese Sprachen repräsentieren verschiedene Klassen von Zielsprachen – die objektorientierte Sprache Java zum Einsatz auf verschiedenen Plattformen, die imperative Sprache C, zum Einsatz auf Mikrocontrollern und die Hardwarebeschreibungssprache VHDL für die Generierung von Hardwaremonitoren.

7.2 Enterprise Architect Add-in

Zusätzlich zur Umsetzung des Codegenerierungsprozesses wurde im Rahmen dieser Arbeit ein Add-in für SparxSystems Enterprise Architekt (EA) entwickelt. Dieses erweitert EA um die graphische Modellierung von MPNs, einen Import von MPNs, die in Form von XMI-Dateien vorliegen, sowie die Simulation (Visualisierung) von aufgezeichneten Monitordurchläufen.

[1] OMG MDA-Website: http://www.omg.org/mda/
[2] StringTemplate-Website: http://www.stringtemplate.org
[3] ANTLR-Website: http://www.antlr.org

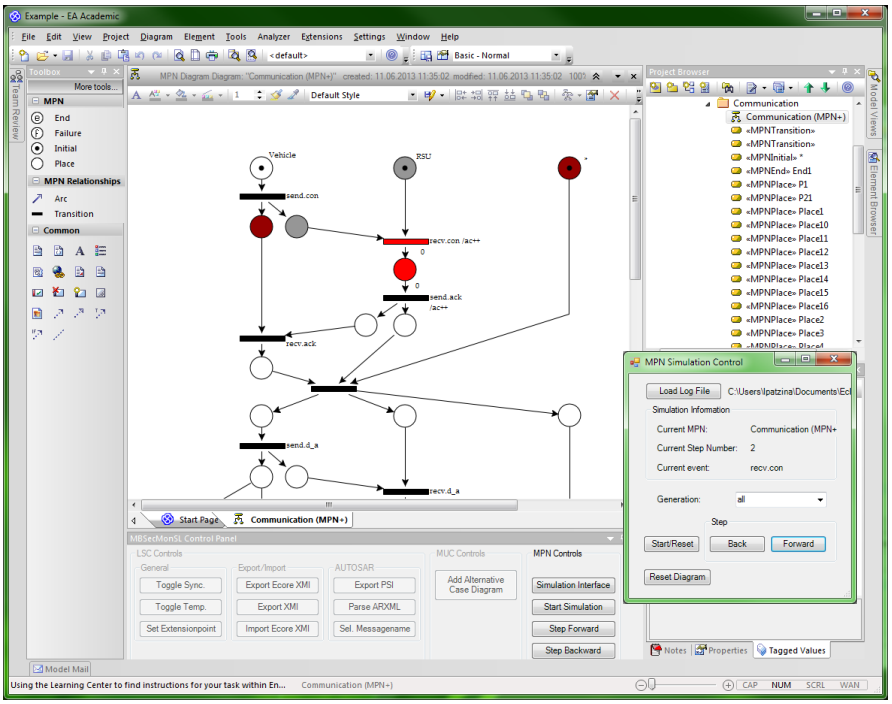


Abbildung 7.2: MPN-Editor als EA-Add-in

7.2.1 *Monitor-Petrinetz Editor*

In Abbildung 7.2 ist der Editor für Monitor-Petrinetze in EA zu sehen. Er unterstützt die grafische Modellierung von Monitor-Petrinetzen sowie den Import und die Simulation bzw. das Abspielen von vorher aufgezeichneten Durchläufen durch das MPN zu Debuggingzwecken.

Auf der linken Seite befindet sich die *Toolbox* mit den vier verschiedenen Platztypen – *Initial*, *Place*, *End* und *Failure*. Des Weiteren enthält diese Transitionen (*Transition*) und Kanten (*Arc*). Mithilfe dieser Elemente können auf der Arbeitsfläche (Mitte) MPNs modelliert werden. Auf der rechten Seite befindet sich der *Project Browser*, in dem die einzelnen Diagramme und deren Elemente in einer Baumstruktur mit Paketen eingeordnet sind. Zur Simulation der MPNs befindet sich unten rechts das *MPN Simulation Control*-Fenster und eine vereinfachte Version unten im *MBSecMonSL Control Panel*.

7.2.2 *Import*

Nach der Transformation der Spezifikationen von der Spezifikationssprache in die MPN-Sprache kann die entstandene MPN-Repräsentation als Ecore-XMI-Datei gespeichert werden. Dieses XML-Format ist jedoch nicht dazu geeignet, das Ergebnis der Transformation zu überprüfen, da es in der XML-Repräsentation nur schwer lesbar ist. Aus diesem Grund unterstützt das MPN-EA-Add-in den Import von Ecore-XMI-Repräsentationen der MPNs. Die Elemente des MPNs werden

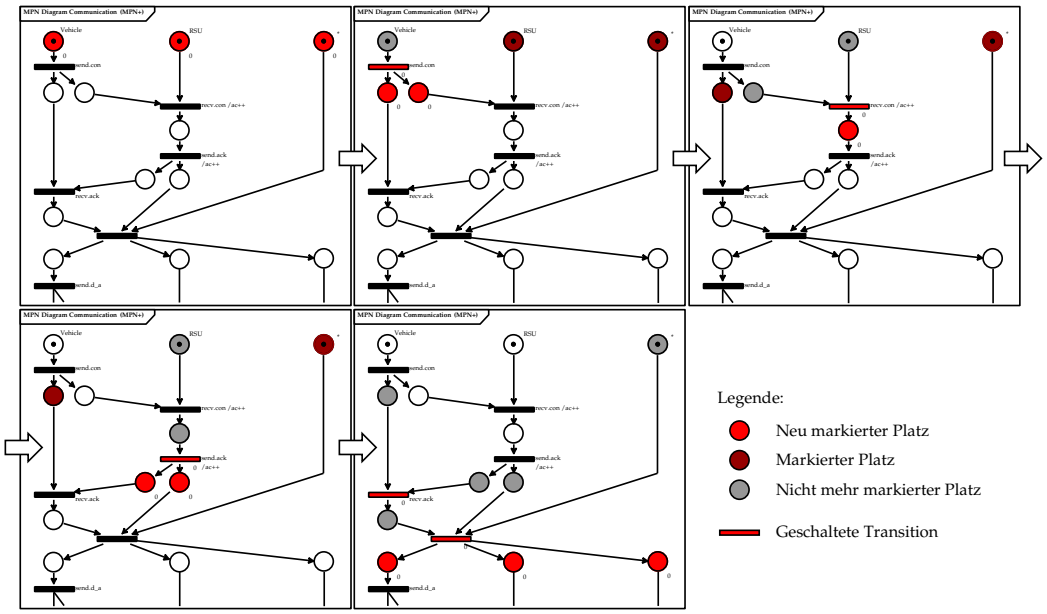


Abbildung 7.3: Beispiel für eine Trace-Visualisierung im MPN-Editor

nach dem Import automatisch in einem Diagramm angeordnet, sodass sich ein übersichtliches MPN-Diagramm entsteht. Dieser Algorithmus ist auf die Struktur der MPNs optimiert, die aus Sequenzdiagrammen, insbesondere aus eLSCs, entsteht.

7.2.3 *Simulation/Visualisierung*

Der MPN-Editor unterstützt die schrittweise Visualisierung von Traces (Schaltschritten des MPNs), die während der Ausführung eines aus Monitor-Petrinetzen generierten Monitors aufgezeichnet wurden. Hierzu lässt sich während der Monitorgenerierung zusätzlicher Code zur Aufzeichnung, der durch den Monitor zur Laufzeit durchgeführten Schritte, generieren. Diese Traces können nicht nur aus generiertem Monitorcode sondern auch aus der Ausführung der MPN-Signaturen in einem MPN-Interpreter stammen. Diese in einer Datei gespeicherten Traces können zusammen mit dem MPNs in den Editor importiert werden und dort die Ausführung für verschiedene Generationen schrittweise durchgespielt werden.

Abbildung 7.3 zeigt die ersten fünf Schritte in der Ausführung der *Communication*-Signatur aus Abschnitt 5.1. Die Schritte sind von links nach rechts und von oben nach unten zu lesen, in denen die Ereignisse *send.con*, *recv.con*, *send.ack*, *recv.ack* verarbeitet werden. Hierbei werden die Plätze, auf denen Token liegen, in denen Token gelöscht wurden und in denen neue Token generiert wurden farbig hervorgehoben. Transitionen, die im gerade angezeigten Makroschritt gefeuert haben, werden ebenfalls hervorgehoben. Die Zahl unten rechts an den hervorgehobenen Plätzen und Transitionen repräsentieren die Generation der Token im entsprechenden Platz. Im hier gezeigten Durchlauf existiert nur die Generation 0. Der letzte Schritt, in dem das Ereignis *recv.ack* verarbeitet wird, zeigt einen Ma-

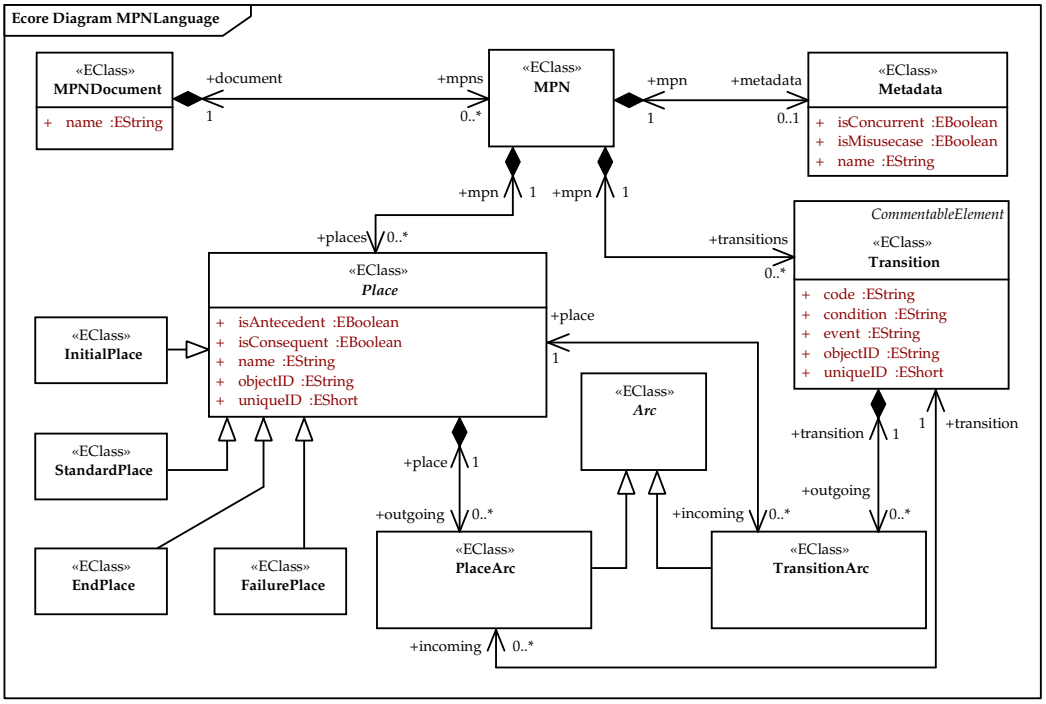


Abbildung 7.4: Metamodell der MPNs

kroschritt in dem zusätzlich eine Epsilontransition schaltet. Hier werden beide im Makroschritt schaltenden Transitionen hervorgehoben und alle Plätze, die belegt waren, als nicht mehr markiert gekennzeichnet.

Diese Visualisierung kann im Nachhinein zu Debuggingzwecken genutzt werden, um festzustellen, ob der Monitor sich so wie erwartet verhält oder um zu erkennen in welchem Schritt einen Fehler erkannt wurde. Durch die Kopplung der Visualisierung mit dem MPN-Interpreter ist eine Simulation der Ausführung von MPN-Signaturen umsetzbar.

7.3 Generierung von Monitoren

Bei der Generierung des Monitors aus MPN-Spezifikationen findet wie beschreiben zunächst eine Übersetzung von dem *MPN Repository* in eine für die Codegenerierung angepasste Datenstruktur, das sogenannte *Codegen Repository*, statt. Dieses bietet über standardisierte Schnittstellen Zugriff auf die für die Codegenerierung optimierte Form der MPN-Signaturen.

7.3.1 *Das MPN-Metamodell*

Die Implementierung der MPN-Datenstrukturen ist im Metamodellierungswerkzeug eMoflon realisiert. Abbildung 7.4 zeigt das Ecore-Metamodell der MPN-Sprache, in das durch eine Transformation verschiedene Spezifikationssprachen übersetzt werden. Die Klasse *MPNDocument* bildet den obersten Behälter in dem beliebig viele MPNs (*MPN*) einer Spezifikation enthalten sind. Über die von der

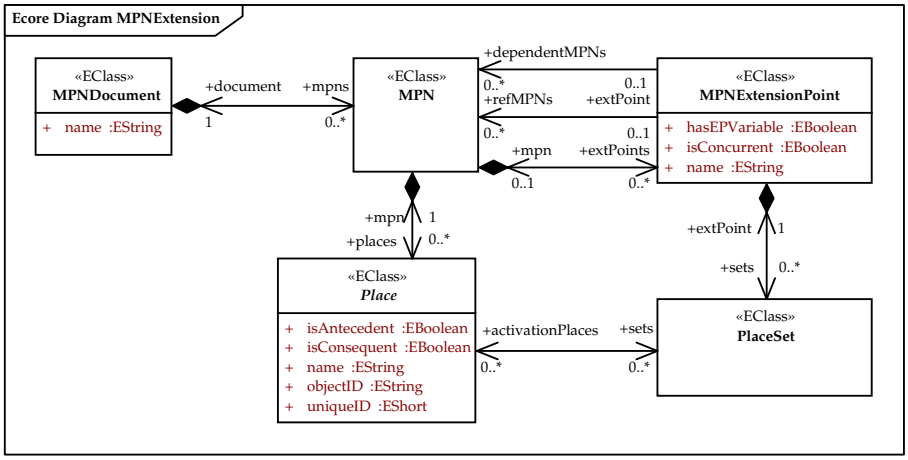


Abbildung 7.5: Metamodell des MPN-Referenzsystems

Klasse *MPN* referenzierte Klasse *Metadata* werden grundlegende Konfigurationen, die nicht zielplattformspezifisch sind, der einzelnen MPNs gespeichert. Hierunter fallen Informationen zur Überwachung und Auswertung, wie ob Subgenerationen genutzt werden (*isConcurrent*) und ob es sich bei dem MPN um einen Misusecase handelt (*isMisusecase*). Die Klasse *MPN* enthält Plätze (*Place*) und Transitionen (*Transition*). Beide Klassen haben hierbei das Attribut *objectID*, das eine Zuordnung der MPN-Elemente zu bestimmten zu überwachenden Instanzen ermöglicht. Diese werden, wie in Abschnitt 6.6 besprochen, zur Aufteilung der Signaturen auf verschiedene Steuergeräte benötigt. Des Weiteren haben diese Klassen ein Attribut *uniqueID*, das eine eindeutige Identifikation der Elemente während der Monitorgenerierung und zur Laufzeit für die Synchronisierung verteilter Teilmonitore erlaubt. Plätze existieren hierbei, wie in Kapitel 5 definiert, in den Ausprägungen *InitialPlace*, *StandardPlace* und den Terminalplatztypen *EndPlace* und *FailurePlace*. Ihr Attribut *isAntecedent* bzw. *isConsequent* bestimmt, ob der Platz für die Auswertung des Monitorzustandes den vorhergehenden Teil (eine Vorbedingung) oder den nachfolgenden Teil der Signatur symbolisiert. Annotationen an Transitionen werden als Strings in den Attributen *event* für das Ereignis, *condition* für die Bedingung und *code* für Aktionen gespeichert. Verbunden werden Plätze mit Transitionen durch die Klasse *PlaceArc* und Transitionen mit Plätzen durch die Klasse *TransitionArc*, die jeweils von der abstrakten Oberklasse *Arc* erben.

Abbildung 7.5 zeigt die Erweiterung des MPN-Metamodells um das Referenzsystem der MPN-Sprache, wie es in Abschnitt 6.1 eingeführt ist. Diese besteht aus den zwei zusätzlichen Klassen *MPNExtensionPoint* und *PlaceSet*. Der *MPNExtensionPoint* beschreibt hierbei die Erweiterungspunkte zwischen den MPNs und referenziert weitere MPNs. MPNs kennen hierbei beliebig viele Erweiterungspunkte. Zur Repräsentation der Aktivierungsplatzmenge S_{act} existiert die Klasse *PlaceSet*, die eine Referenz zu den Plätzen des Basis-MPNs besitzt. Hierdurch können beliebig viele Plätze des referenzierenden MPNs (*Basis-MPN*) in die Menge S_{act} übernommen werden. Zur Laufzeit des Monitors muss einer der referenzierten Plätze mit einem Token belegt sein, damit eine neue Instanz des referenzierten MPNs erzeugt wird. Das Attribut der Klasse *ExtensionPoint isConcurrent* definiert,

ob das referenzierte MPN parallel zu dem Basis-MPN oder erst nach dessen Fehlschlagen ausgeführt wird. Das Attribut *hasEPVariable* bestimmt, ob der Erweiterungspunkt eine generationsspezifische Variable besitzt, die im referenzierenden MPN zum Blockieren von Teilabläufen genutzt wird. Zur Spezifikation der Teilabhängigkeiten, die zusammen mit dem EP ausgewertet werden müssen, dient die Referenz mit dem Ende *dependentMPNs*.

Das vorgestellte MPN-Metamodell repräsentiert die Konzepte der MPNs sehr übersichtlich, ist jedoch für die Codegenerierung nicht optimal geeignet. Plätze müssen bei der Codegenerierung erst nach Typ gefiltert werden und *Arc*s müssen über zwei Schritte verfolgt werden. Dies führt zu einer aufwendigeren Codegenerierung, da Informationen nicht direkt aus diesem Metamodell abfragbar sind.

Abbildung 7.6 zeigt das zur Codegenerierung optimierte Metamodell. Es besteht nur noch aus sechs Klassen. Abhängigkeiten zwischen Plätzen und Transitionen, die im vorherigen Metamodell als *Arc* repräsentiert wurden, sind nun direkt als Referenzen modelliert. Alle Plätze werden in die allgemeine Klasse *GenViewPlace* übersetzt und die verschiedenen Ausprägungen als Referenzen repräsentiert, die einen direkten Zugriff auf die Platzmengen ermöglichen. Durch die Übersetzung indirekter Zusammenhänge in klare Abhängigkeiten über zusätzliche Referenzen kann nun z. B. direkt auf die Plätze im Vor- und Nachbereich einer Transition zugegriffen werden. Instanzen verschiedener Platztypen sind nun durch spezialisierte Referenzen direkt erreichbar, wodurch sich die Codegenerierung vereinfacht.

Zur Generierung von verteilten Monitoren, die aufgeteilte Signaturen überwachen (Abschnitt 6.6), werden die Plätze der Signaturen in zwei zu synchronisierende Mengen eingeteilt. Das Attribut der Klasse *Place isSynchIn* beschreibt hierbei Plätze, die durch die Belegung eines andern Platzes in einem anderen Teilmonitor der mit *isSynchOut* gekennzeichnet ist, synchronisiert werden. Zum direkten Zugriff auf diese Mengen befinden sich zwischen *GenViewMPN* und *GenViewPlace* die Referenzen *MPNHasSyncIn* und *MPNHasSyncOut*. Bei der Übersetzung in diese Repräsentation werden Optimierungen an den MPN-Signaturen und die Aufteilung in Teilmonitore durchgeführt.

Die Erweiterung um das Referenzsystem findet sich im oberen Teil des Metamodells zur Codegenerierung in den Klassen *GenViewExtensionPoint* und *GenViewPlaceSet* wieder. Im Endeffekt wird der Teil des in Abbildung 7.5 gezeigten Metamodells in das Codegenerierungsmetamodell direkt übernommen.

7.3.2 *Optimierung von MPNs*

Monitor-Petrinetze (MPNs) sind eine Zwischensprache, die dazu dient verschiedene Spezifikationssprachen in einer für die Codegenerierung und Interpretation gut geeigneten, expliziteren Form abzubilden. Wie bei den bisher betrachteten Übersetzungen von erweiterten Live Sequence Charts (eLSCs) zu MPNs zu sehen ist, führt die Transformation einer Spezifikationssprache, in der die Kommunikationspartner einzeln modelliert sind, nicht immer zu minimalen MPNs. Das in Abschnitt 6.1 eingeführten MPNs für die Signatur *CARDME Local* und *DoS Attack* zeigen dies deutlich. Die dedizierten Plätze für jeden Kommunikationspartner

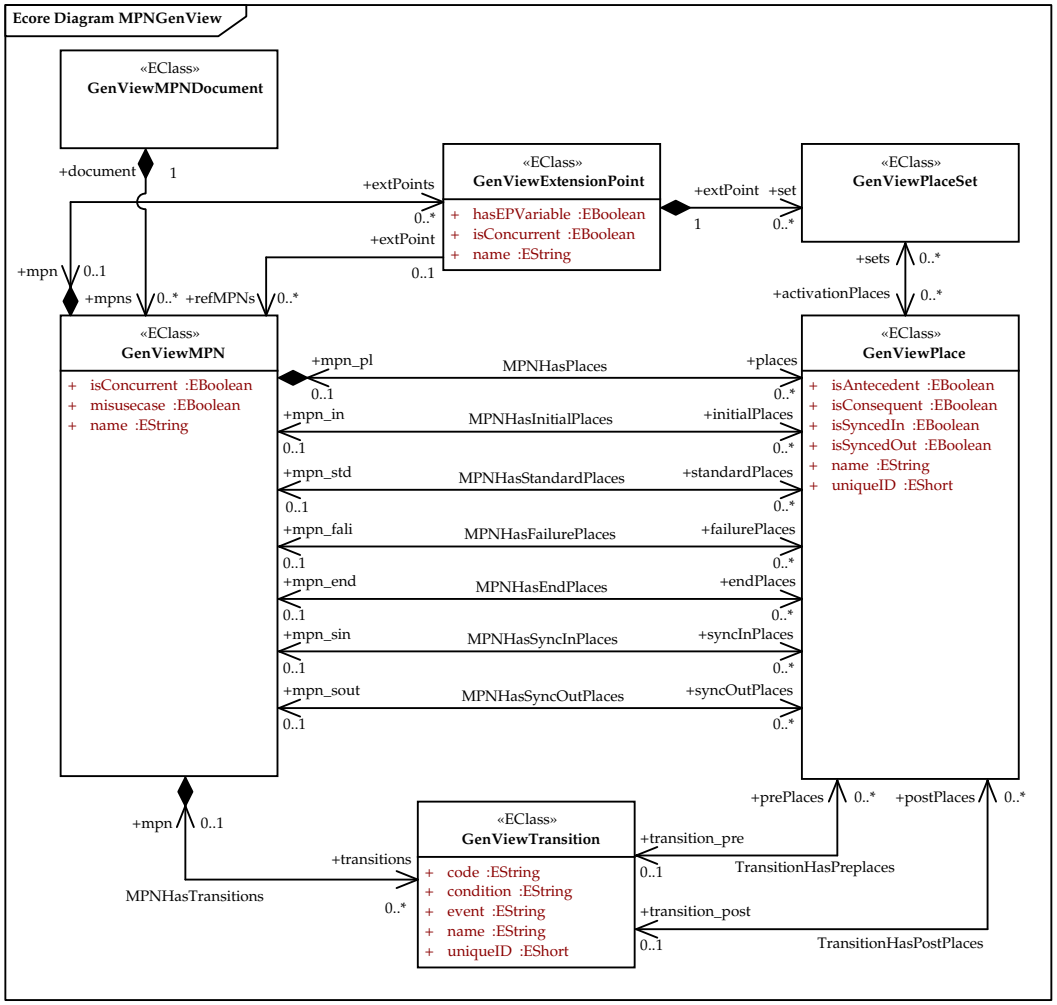


Abbildung 7.6: MPN-Codegenerierungsmetamodell

nach einer Synchronisation ermöglichen die Aufteilung der Signaturen nach der tatsächlichen Verteilung der Komponenten im Zielsystem (Abschnitt 6.6). Jedoch führen diese dazu, dass mehr Plätze in der übersetzen Signatur vorhanden sind, als zur Überwachung benötigt werden.

REDUNDANTE PLÄTZE UND TRANSITIONEN Abbildung 7.7 zeigt Optimierungen, die vor der Codegenerierung auf den MPNs durchgeführt werden, um den Speicherverbrauch und die maximale Laufzeit für eine Ereignisverarbeitung zu reduzieren. In Optimierungstransformation a) werden parallele Plätze, auf die eine weitere synchronisierende alle Plätze umfassende Transition folgt, zu einem gemeinsamen Platz verschmolzen. Hierbei können vor und nach den zu verschmelzenden Plätzen beliebig viele Transitionen, die beide Plätze im Vor- bzw. Nachbereich haben, existieren. Bei diesem Schritt müssen keine Annotationen an den Transitionen beachtet werden, da sich an den akzeptierten Ereignissequenzen (AES) nichts ändert. Optimierungstransformation b) zeigt, wie aufeinander folgende Transitionen, die in einem Makroschritt schalten würden, zusammenge-

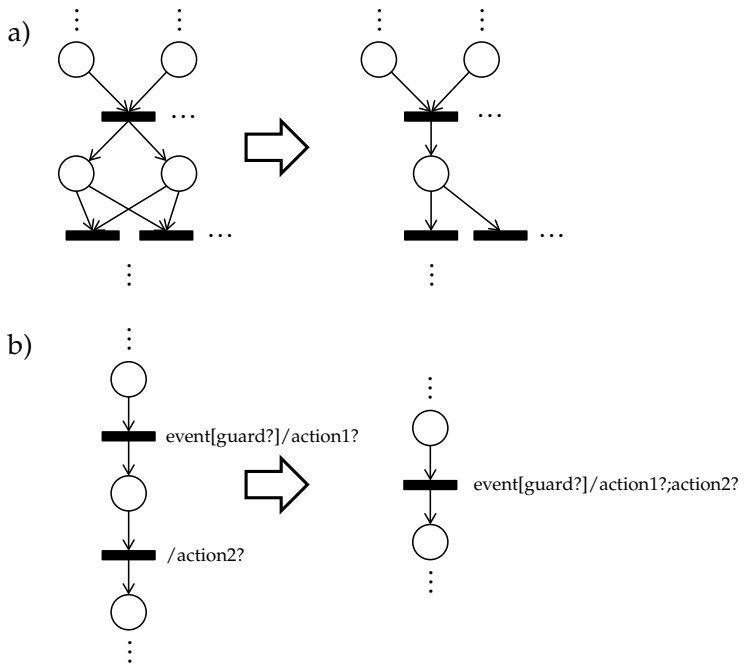


Abbildung 7.7: Mögliche Optimierungen in MPNs

fasst werden können. Hierbei wird eine Transition mit einem annotierten Ereignis mit einer nachfolgenden Epsilontransition ohne Bedingung zu einer Transition zusammengefasst. Diese Struktur entsteht zum Beispiel bei der Übersetzung eines eLSCs mit einer auf den Empfang einer Nachricht folgenden Zuweisung, wie sie in der Signatur *DoS Attack* auftritt. Bei der Verschmelzung der Transitionen werden annotierte Aktionen nach ihrer Reihenfolge im Netz aneinandergehängt, wodurch die durch die ursprüngliche Signatur beschriebene Abfolge nicht verändert wird. Diese Transformation kann beliebig oft angewendet werden, um mehrere Epsilontransitionen hintereinander mit der vorhergehenden Transition zu kombinieren.

Bei der Verwendung des Referenzsystems sind bei der Optimierung zusätzlich die Platzmengen, die Erweiterungspunkte aktivieren (S_{act}), zu beachten. Werden durch die Optimierung Plätze zusammengefügt, müssen die Platzmengen entsprechend angepasst werden. In Fall a) auf den neuen die vorherigen Plätze ersetzenden Platz und in Fall b) auf den Platz nach der kombinierten Transition. Durch die Semantik des Referenzsystems, in der die referenzierten MPNs erst nach Durchführung des Makroschrittes aktiviert bzw. deaktiviert werden, führt diese Anpassungen der Platzmengen bei Einhaltung der Modellierungsrichtlinien des Referenzsystems zu keiner Änderung des Laufzeitverhaltens.

Zwei aufeinanderfolgende Transitionen mit identischem Ereignis können nicht kombiniert werden, da sich hiermit die Aussage des MPNs von zwei aufeinanderfolgenden Ereignissen auf nur ein erwartetes Ereignis verändern würde. Des Weiteren ist eine Zusammenführung zweier aufeinanderfolgender Transitionen, von denen die zweite eine Bedingung besitzt, nicht generell semantisch äquivalent, da die Variablen, auf die diese Bedingung zugreift, nicht zum Zeitpunkt des

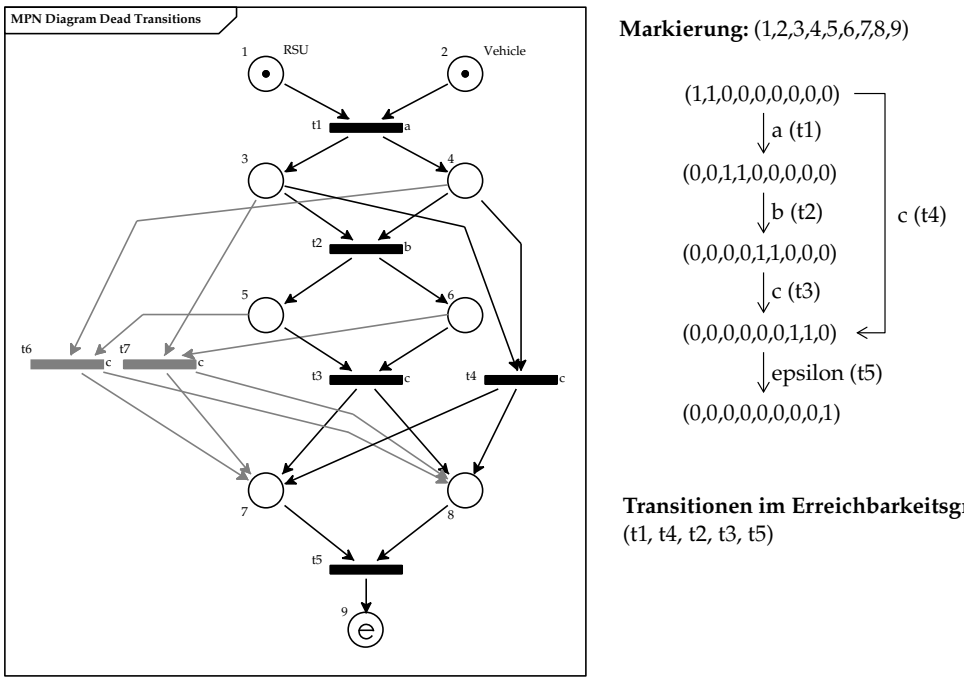


Abbildung 7.8: Erkennen von toten Transitionen in einem MPN

Auftretens des Ereignisses der ersten Transition mit den erwarteten Werten belegt sein muss.

TOTE TRANSITIONEN Eine weitere Möglichkeit der Optimierung der MPNs vor der Monitorgenerierung ist die Bestimmung toter Transitionen.

Definition 66 (Tote Transitionen). *Eine Transition in einem Petrinetz ist genau dann tot, wenn zur Laufzeit keine Markierung eintreten kann, die diese Transition aktiviert.*

Die MPN-Signaturen werden hierbei durch den Aufbau eines Markierungsgraphen auf Basis der an den Transitionen annotierten Ereignisse analysiert. Ist eine Transition nicht im Markierungsgraphen vorhanden, kann diese auch nicht schalten und kann aus der Signatur entfernt werden.

Beispiel *Tote Transitionen durch Übersetzung von optionalen Nachrichten*

Abbildung 7.8 zeigt ein Beispiel-MPN, in dem tote Transitionen (*t5* und *t6*) durch die Übersetzung einer kalten Nachricht aus der eLSC-Sprache vorhanden sind. Beim Aufbau des Markierungsgraphen werden die Transitionen, die in ihn übernommen werden, aufgezeichnet. Im Beispiel sind dies die Transitionen *t1* bis *t5*. Transition *t6* und *t7* können durch die Struktur des MPNs nie schalten und sind somit tote Transitionen, die aus dem MPN entfernt werden können.

Durch Entfernung toter Transitionen können wie im Beispiel weitere Optimierungen durchgeführt werden, die vorher nicht möglich waren. So können nach Regel a) in Abbildung 7.7 jeweils die parallelen Plätze {1, 2}, {3, 4}, {5, 6} und {7, 8} kom-

biniert werden. Hierdurch kommt es zu einer Reduktion des benötigten Codespeichers durch weniger Transitionen, des benötigten Datenspeichers (RAMs) durch weniger Plätze und der Laufzeit durch weniger generierte und gelöschte Token in einem Makroschritt. Zusätzlich führt die Reduzierung der Anzahl der Transitionen mit identischen Ereignistypen zu einem Rückgang der Laufzeit des Monitors bei der Verarbeitung dieser Ereignisse. Durch diese Analyse auf tote Transitionen kann des Weiteren bei der Übersetzung von Quellspezifikationen in die MPN-Sprache die Anzahl bzw. der Umfang der benötigten Transformationsregeln reduziert werden, da weniger Spezialfälle betrachtet werden müssen.

Diese Analyse der Signaturen kann auch dazu genutzt werden, dem Modellierer der Signatur Rückmeldung zu geben und hierdurch mögliche Spezifikationsfehler aufzudecken. Die so erreichten Ergebnisse sind nicht zwingend optimal, da nur die an den Transitionen annotierten Ereignisse mit in die Analyse einbezogen werden. Bedingungen bzw. Aktionen, die zur Laufzeit ebenfalls zu toten Transitionen führen können, werden jedoch ignoriert.

7.3.3 *Gewinnung von plattformspezifischen Informationen*

Zur Generierung von Monitoren aus diesen optimierten Signaturen werden plattformspezifische Informationen (PSI) benötigt. Diese Informationen können aus Systembeschreibungen, die während der Spezifikation in einem modellbasierten Entwicklungsprozess entstehen, gewonnen werden. Dies können zum Beispiel Komponentendiagramme sein, die das zu überwachende System beschreiben. Hieraus werden die nötigen Informationen der Schnittstellenbezeichnungen, die für die Generierung des Monitors und die Instrumentierung des Systemcodes notwendig sind, ausgelesen. Zusätzliche Konfigurationen, die monitorspezifisch sind, können aus der Spezifikation des Monitors extrahiert werden und gegebenenfalls durch den Entwickler vervollständigt werden.

Im MBSecMon-Tool wird bei der Transformation von Signaturen in der MBSecMonSL zu MPNs eine Datei mit plattformspezifischen Informationen automatisch angelegt.

Beispiel *PSI-Datei für einen Monitor*

Abbildung 7.9 zeigt eine vereinfachte Signatur (ohne Subgenerationen) des CARDME-Protokolls. Nachdem die InitialisationRequest-Nachricht (*InitReq*) gesendet wurde, antwortet das Fahrzeug (Vehicle) mit einer InitializationResponse-Nachricht (*InitRes*), die in diesem Fall ein Datenpaket *ir* mit Informationen des zu verwendenden Protokolls mitführt. Wenn das Datenpaket das Protokoll mit dem Identifizierer 1 beinhaltet, muss abschließend ein *ClosingRequest* verschickt werden.

Zur Übergabe der Ereignisse an den Monitor werden zur Instrumentierung des zu überwachenden Zielsystems Schnittstellen nach außen benötigt. Diese Konfiguration des Generierungsprozesses findet über eine Konfigurationsdatei statt. Eine solche plattformspezifische Informationsdatei (PSI-Datei) mit Konfigurationsinformationen und plattformspezifischen Informationen für dieses

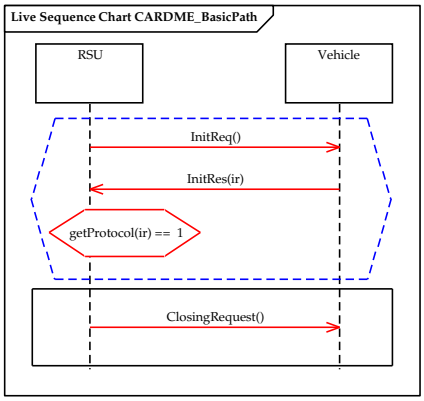


Abbildung 7.9: Signatur zur Überwachung des CARDME-Protokolls

Beispiel ist in Auflistung 7.1 gezeigt. Im Folgenden wird auf diese PSI-Datei im Detail eingegangen. Die PSIs sind in drei Teile eingeteilt:

- Die grundlegende Konfiguration, die die Zielsprache, die Zielplattform und allgemeine Konfigurationsdetails spezifiziert.
- Die plattformspezifischen Schnittstelleninformationen, die Namen für benötigte externe Schnittstellen, Parameter und deren Datentypen enthält. Diese werden auch für die Abbildung zwischen externen Schnittstellen und internen Ereignissen genutzt.
- Die spezifische Monitorkonfiguration und die Abbildung von Annotationen an den Transitionen auf die Zielplattform bzw. Zielsprache. Viele der grundlegenden Konfigurationen wie die maximale Anzahl der Subgenerationen können auch hier MPN-spezifisch spezifiziert werden. Hierbei ist durch Referenzen (@...) ein Zugriff auf andere Schlüssel möglich.

Bei der Generierung der PSI-Datei aus der Systemspezifikation werden Standardschlüssel für die Konfiguration der Codegenerierung zur Anpassung an die Zielplattform generiert. Zusätzlich wird für jedes MPN eine Menge von Konfigurationsparametern generiert, die zum Beispiel bestimmen, ob es sich um ein verschränkt ausgeführtes MPN handelt und, wenn ja, wie viele Subgenerationen benötigt werden. Annotationen an den Transitionen der MPNs werden durch einen ANTLR-Parser eingelesen und z. B. für Methodenaufrufe und Variablen, die eine Anpassung an die Zielsprache benötigen, Einträge erzeugt. Alle diese Informationen können, wenn eine ausreichend umfangreiche Systemspezifikation vorliegt, aus dieser automatisch gewonnen werden. Die Übersetzung von Methoden und Funktionen auf die Zielsprache ist durch vorgegebene Bibliotheken möglich oder vom Entwickler einmalig in dieser Datei durchzuführen.

7.3.4 *Einbindung von plattformspezifischen Informationen*

Die im vorherigen Abschnitt beschriebenen in einer XML-Datei gespeicherten plattformspezifischen Informationen werden zur Codegenerierung in das MBSec-

Auflistung 7.1: Plattformspezifische Informationen

```
<?xml version="1.0" encoding="UTF—8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd"[]>
<properties>
  <comment>Mapping</comment>
  <!--Configuartion-->
  <entry key="genLanguage">C</entry> // Zielsprache
  <entry key="genTargetPlatform">Standard</entry> // spezifizierte Zielplattform
  <entry key="instancesToGenerate"></entry> // zu generierender Teilmonitor
  <entry key="checkEvent">true</entry> // filtern von Ereignissen auf Vorkommen in Signatur
  <entry key="activateMPNInteractions">false</entry> // Aktivierung des Referenzsystems
  <entry key="maxGenerationCount">5</entry> // maximale Anzahl der Eingabegenerationen
  <entry key="maxSubgenerationCount">3</entry> // maximale Anzahl der Subgenerationen
  <entry key="optimizeMPNs">false</entry> // Optimierungen vor Generierung aktivieren
  <entry key="inlineTransitionFiring">true</entry> // Inlining des Codes zum Schalten der
       Transitionen
  <entry key="tempPlaces">false</entry> // Kopie der Plätze statt Änderungsvariablen verwenden
  <entry key="enableLog">false</entry> // Logging aktivieren
  <entry key="contextIncludes">../include/Datatypes.h</entry> // Zusätzliche Includes des Kontextes

  <entry key="contextVariableName">contextTemp</entry> // Name der Kontextvariablen
  <entry key="Controller.packageName">monitor</entry> // Paketname des Controllers
  <entry key="Controller.className">Controller</entry> // Name des Controllers
  <entry key="Controller.imports"></entry> // Zusätzliche Imports des Controllers

  <!--external to internal event mapping-->
  ...
  <entry key="event.SEND_INITRESPONSE">EVENT_SEND_INITRES</entry>
  <entry key="event.SEND_INITRESPONSE.param">ir_param</entry>
  <entry key="event.SEND_INITRESPONSE.param_type">IRData</entry>
  <entry key="event.SEND_INITRESPONSE.param_init">{0,""}</entry>
  ...

  <!--specific mappings for LSC: CARDME_BasicPath-->
  <entry key="CARDME_BasicPath.tempPlaces">false</entry>
  <entry key="CARDME_BasicPath.classVariables"></entry>
  <entry key="CARDME_BasicPath.includes"></entry>
  <entry key="CARDME_BasicPath.subgenerations">false</entry> // nicht verschränkt
  <entry key="CARDME_BasicPath.context.ir">@event.SEND_INITRESPONSE.param</entry>
  <entry key="CARDME_BasicPath.context_type.ir">@event.SEND_INITRESPONSE.param_type</entry>
  <entry key="CARDME_BasicPath.context_gendependent.ir">true</entry>
  <entry key="CARDME_BasicPath.context_method.getProtocol">getProtocolId</entry>

  ...
</properties>
```

Mon-Tool geladen und sind dort während der Codegenerierung über die angegebenen Schlüssel einheitlich abrufbar. Für die Codegenerierung wird zunächst der grundlegende Teil der PSI ausgewertet und die Codegeneratoren des MBSecMon-Tools entsprechend konfiguriert. Nachdem die MPN-Signaturen, wie in Abschnitt 7.3.2 beschrieben, optimiert und gegebenenfalls Teilsignaturen erstellt wurden, werden diese an den Codegenerator übergeben. Grundlegend besitzt jeder Codegenerator des MBSecMon-Tools eine Übersetzung der MPN-Elemente in eine Repräsentation in der angegebenen Zielsprache. Während der Codegenerierung, die im Prototyp mit Java und StringTemplate umgesetzt wurde, werden die Annotationen an den Transitionen der MPNs mittels eines ANTLR-Parsers eingelesen und mithilfe der PSI-Informationen in die Zielsprache übersetzt. Aus der Kombination der plattformunabhängigen Signaturbeschreibungen und den PSI entsteht ein ausführbarer Monitor.

7.3.5 *Trennung von Monitorspezifikationen*

In Abschnitt 6.6 wurde die Aufteilung von Monitorspezifikationen in mehrere Teilsignaturen für die Verteilung auf mehrere Steuergeräte betrachtet. Im Folgenden wird auf die Auswirkungen für die Implementierung eingegangen.

Zur Aufteilung der Monitore werden zusätzliche Informationen benötigt. Zum einen sind dies, welche Kommunikationspartner miteinander über einen Kanal kommunizieren können bzw. sollen. Diese Information wird benötigt, um zu bestimmen, ob die Teilsignaturen als Einzelsignaturen ausgeführt werden oder sich untereinander während der Laufzeit des Monitors synchronisieren. Des Weiteren müssen die Kommunikationsschnittstellen (Sende- und Empfangsmethoden) definiert werden, die für diese Synchronisation zwischen den Teilmonitoren auf dem Zielsystem verwendet werden. Diese Informationen wurden als Einträge in die PSI-Datei ausgelagert. Die Plätze eines MPNs wurden vor der Trennung der MPNs in Teilsignaturen zusätzlich mit eindeutigen Identifizierern (*engl. unique IDs*) markiert, sodass eine eindeutige Identifikation der zu synchronisierenden Plätze gegeben ist. Hierdurch bleibt die Monitorspezifikation mit den MPNs unabhängig von der für den Kommunikationskanal verwendeten Technik.

Der Kommunikationskanal zur Übertragung der Synchronisationsnachrichten muss sicherstellen, dass die Reihenfolge der Nachrichten erhalten bleibt. Außerdem muss die Übertragung und Verarbeitung der Synchronisation abgeschlossen sein, bevor das nächste Ereignis vom empfangenden Monitorteil verarbeitet werden kann. Hierzu muss die Synchronisierung schneller bzw. gleich schnell oder priorisiert zu anderen Nachrichten erfolgen und zum Einsatz in realen Systemen möglichst leichtgewichtig sein.

Für die Überprüfung von Inter-Step-Bedingungen, die Variablenwerte oder Zeitpunkte von Ereignissen benötigen, müssen auch diese, soweit sie den Teilmonitoren nicht zur Verfügung stehen, über den Kommunikationskanal übertragen werden.

7.4 Umsetzungen der Codegenerierung für verschiedene Zielsprachen und Einsatzzwecke

Die Generierung der Monitore wurde für verschiedene Zielsprachen und Plattformen umgesetzt. Hierbei wurde besonders die Generierung von Java-Code als objektorientierte Sprache und C-Code für optimierte Monitore auf eingebetteten Systemen betrachtet. Bei der Java-Implementierung existiert eine universelle Variante und eine erweiterte, die die Instrumentierung von Java-Code des zu überwachenden Systems unterstützt. Hierzu werden Aspekte in der Sprache AspectJ generiert, die die Erzeugung von Ereignissen und deren Zuordnung zu Generationen zur Laufzeit übernehmen. Für die C-Codegenerierung existiert eine universelle und eine an die Anforderungen von AUTOSAR angepasste Variante. Als Machbarkeitsstudie wurde des Weiteren eine Generierung einer VHDL-Implementierung der MPNs mit eingeschränktem Funktionsumfang und ein Interpreter umgesetzt.

In Tabelle 7.1 sind die verschiedenen Implementierungen und die von ihnen unterstützten MPN-Sprachanteile aufgeführt. Die Grundlagen der MPNs umfassen die Syntax und die Schaltsemantik, wie sie in Kapitel 5 definiert wurden. Diese werden durch alle Implementierungen unterstützt. Die Unterscheidung von Eingabegenerationen ist in der AUTOSAR-Variante der C-Generierung nicht aktiviert, da die verwendete AUTOSAR-Entwicklungsumgebung keine Mehrfachinstanziierung von Komponenten unterstützt. Somit wird dieses Konzept hier nicht benötigt. Subgenerationen sind in der prototypischen Umsetzung der VHDL-Variante nicht umgesetzt. Bedingungen, Aktionen sowie die Aufteilung von Signaturen können in allen Implementierungen grundlegend genutzt werden. Komplexe Bedingungen und Aktionen sind jedoch in der VHDL-Implementierung nicht direkt unterstützt, sondern nur Bedingungen auf Basis boolescher Variablen. Eine Generierung von Teilschaltungsbeschreibung in VHDL für komplexe Bedingungen könnte auf Basis des Parseranteils des Generierungsprozesses umgesetzt werden. Die Generierung von Kommunikationsschnittstellen für aufgeteilte Monitore, über die sich die Teilmonitore synchronisieren, ist nur für die C-Code-Generierung umgesetzt.

Im Rahmen einer Arbeit wurde eine Variante der Java-Generierung umgesetzt, die es ermöglicht automatisch Aspekte in AspectJ zu generieren, die die Anbindung an das zu überwachende System ermöglichen. Für die AUTOSAR-Variante ist es notwendig, sich an die Schnittstellenkonventionen des AUTOSAR-Standards zu halten. Die Instrumentierung, die diese Schnittstellen nutzt, wird durch ein externes Werkzeug umgesetzt. Hierzu können dem Prozess Schnittstellenkonventionen, wie sie im Beispiel in Abschnitt 7.3.3 vorgestellt wurden, übergeben werden. Die generierten Schnittstellen des Monitors können dann über Instrumentation des zu überwachenden Systems angesprochen werden.

Das Referenzsystem (Abschnitt 6.1) ist prototypisch für die C-Implementierung umgesetzt. Hierbei werden exemplarisch, die aus der «threaten»- und «mitigate»-Beziehung der MUC-Sprache, entstandenen Referenzen unterstützt.

Interpreter Zusätzlich zu den Umsetzungen in Code bzw. VHDL wurde eine grundlegende Version eines MPN-Interpreters in der Programmiersprache

Feature	Java		C		VHDL	Interpreter
	Standard	Aspekt	Standard	AUTOSAR		
Grundlage MPN	✓	✓	✓	✓	✓	✓
Eingabegenerationen	✓	✓	✓	-	✓	✓
Subgenerationen	✓	✓	✓	✓	-	-
Bedingungen	✓	✓	✓	✓	✓	✓
- komplex	✓	✓	✓	✓	-	✓
Aktionen	✓	✓	✓	✓	✓	✓
- komplex	✓	✓	✓	✓	-	✓
Aufteilbarkeit	✓	✓	✓	✓	✓	-
- mit Kommunikation	-	-	✓	-	-	-
Instrumentierung	-	✓	-	(✓)	-	-
Spezielle Schnittstellen	-	✓	-	✓	-	(✓)
Referenzsystem	-	-	✓	✓	-	-

✓: unterstützt; -: nicht unterstützt

Tabelle 7.1: Unterstütze Features der Monitorgenerierungen der Implementierungen

Java entwickelt [Lan11]. Hierbei wurde im Metamodellierungswerkzeug MOFLON [AKRS06] die Transformationssprache *Story Driven Modelling* (SDM) eingesetzt, um die Semantik der MPNs zu definieren. Der Interpreter liest die XMI-Repräsentation eines MPNs in ein durch MOFLON generiertes MPN-Repository ein und kann Ereignissequenzen auf Basis der eingelesenen MPN-Signaturen überwachen. Der Interpreter fungiert hier als Server, an den über das *XML Remote Procedure Call*-Protokoll das zu überwachende System per Instrumentierung angebunden werden kann. Diese Schnittstelle ist austauschbar. Zur Auswertung der Annotationen an den Transitionen der MPNs wird eine eingeschränkte Grammatik unterstützt.

8

MONITOR-PETRINETZE ZU CODE

Nachdem im vorherigen Kapitel die grundlegende Umsetzung des MBSecMon-Prozesses und insbesondere die Vorbereitung der Codegenerierung aus Monitor-Petrinetzen betrachtet wurde, wird in diesem Kapitel die Generierung von Monitoren für verschiedene Zielplattformen mit dem Fokus auf der Codegenerierung vorgestellt. Hierbei werden zur Vorbereitung der Evaluation in Kapitel 9 die grundlegenden Eigenschaften der Codegenerierung in die verschiedenen Zielsprachen betrachtet.

Bei der Codegenerierung in die in Abschnitt 7.4 vorgestellten Zielsprachen wurden folgende Prinzipien verfolgt, um die Anpassungen der Codegenerierung an das Zielsystem zu vereinfachen. Der *Controller* des Monitors sollte möglichst einfach strukturiert sein, da er zielsystemspezifische Anteile, wie Schnittstellen nach außen und die Parameterzwischenspeicherung enthält. Des Weiteren übernimmt er die Ansteuerung der einzelnen Signaturen. Der *MPN-Code* sollte möglichst generisch sein, sodass er für verschiedene Zielplattformen in seiner Struktur nicht angepasst werden muss. Hierdurch ergibt sich für jede Zielsprache eine grundlegende Übersetzung, die an die Zielplattform angepasst werden kann. So werden unter anderem die Schnittstellen für verschiedene Plattformen wie AUTOSAR angepasst und eine obere Grenze für die maximal mögliche Generationsanzahl im zu überwachenden System bestimmt.

In Abschnitt 8.1 wird zunächst auf die unterschiedlichen, im Generierungsprozess exemplarisch eingesetzten Zielsprachen und deren Eignung für verschiedene Zielplattformen eingegangen. Darauf folgend werden in Abschnitt 8.2 existierende Konzepte zur Codegenerierung aus petrinetzähnlichen Sprachen betrachtet und die Besonderheiten der Codegenerierung aus MPNs besprochen. Anhand der Codegenerierung für C wird dann in Abschnitt 8.3 die Umsetzung der Monitore, die später auch in der Evaluation verwendet wird, vorgestellt. Die generierten plattformunabhängigen Monitore werden in Abschnitt 8.4 anschließend durch zusätzliche plattformspezifische Informationen an die Zielplattform angepasst. Abschließend wird in Abschnitt 8.5 ein die Codegenerierung optimierender Ansatz zur automatischen Bestimmung der maximal möglichen Änderungen in einem Schritt im MPN beschrieben.

8.1 Unterschiede der Zielsprachen

Zielsprachen, in denen Monitore generiert werden sollen, bieten verschiedenste Programmierkonzepte an. Diese erlauben unterschiedliche Umsetzungen der MPN-Semantik. Diese Umsetzungen haben je nach Zielplattform, auf der die

Monitore laufen sollen, ihre eigenen Vor- und Nachteile. Zur Evaluation des Konzepts der Codegenerierung aus MPNs wurden die folgenden repräsentativen Sprachen und Zielsysteme gewählt:

JAVA: Bei Java handelt es sich um eine objektorientierte Programmiersprache, die hauptsächlich auf PCs eingesetzt wird. Die Umsetzung der MPNs findet durch objektorientierte Konzepte statt, wobei Generationen durch Klasseninstanzen der MPNs repräsentiert werden und komplexe Datentypen wie *HashSet* zur Speicherung von Platzbelegungen eingesetzt werden. Des Weiteren dient diese Sprache zur Anbindung der Monitore an das zu überwachende System mittels aspektorientierter Programmierung [Lan13].

C: Die Programmiersprache C wurde als nicht objektorientierte Programmiersprache für den Einsatz auf PC und Mikrocontroller gewählt. In dieser Umsetzung wird eine möglichst speicher- und laufzeiteffiziente Implementierung der Monitore in den Fokus gerückt. So werden statt Switch-Blöcken der Java-Implementierung Funktionspointer-Arrays verwendet und zur Speicherung der Markierung eines MPNs, die bitweise Speicherung in *Char*-Arrays, auf die mittels Bit-Operationen zugegriffen wird, eingesetzt. Des Weiteren werden zur Unterstützung von AUTOSAR Anpassungen an den Schnittstellen des Monitors nach außen im Generierungsprozess unterstützt und die Konfiguration einer Obergrenze für Generationen eingeführt. Diese Erweiterungen haben das Ziel, dass bei der Ausführung auf einem Mikrocontroller ein minimaler statischer Datenspeicherbedarf (RAM) benötigt wird.

VHDL: Bei VHDL handelt es sich, wie in Abschnitt 2.2 eingeführt, um eine Hardwarebeschreibungssprache, die für die Programmierung rekonfigurierbarer Hardware (FPGA) eingesetzt wird. Ziel dieser Monitorimplementierung ist die Umsetzung der MPNs in eine Parallelität unterstützende Repräsentation. Hierbei werden die MPNs auf eine statische Struktur mit parallelen Netzen für jede Generation abgebildet, um eine möglichst niedrige Laufzeit zu erreichen. Durch die Umsetzung der VHDL-Beschreibung in Hardware können auch komplexe Makroschritte in der MPN-Ausführung, bei denen viele Transitionen überprüft und eventuell geschaltet werden müssen, schnell verarbeitet werden. Somit bietet sich diese Variante als Monitorlösung für die Überwachung von Kommunikationen mit hoher Datenrate an.

8.2 GENERIERUNG VON CODE AUS PETRINETZÄHNLICHEN SPRACHEN

In der Literatur werden zwei Klassen der Codegenerierung aus petrinetzähnlichen Beschreibungen betrachtet. Diese sind die Ansätze der Codegenerierung für Petrinetze sowie verschiedene Implementierungskonzepte für Statecharts. Für die Codegenerierung aus Petrinetzen stellt [Phi06] Ansätze basierend auf struktureller Analyse, Markierungsgraphen und Simulation vor. Die *strukturelle Analyse* produziert zwar durch Übersetzung von Mustern im Petrinetz in korrespondierende Codefragmente der Zielsprache lesbaren und effizienten Code, schränkt jedoch

die Modellierungsfreiheiten der Petrinetze stark ein. Der Ansatz über den *Markierungsgraphen* übersetzt alle möglichen Zustände (Platzkombinationen) in einen Automaten. Dies führt schon bei der Erstellung des Markierungsgraphen zu einer Zustandsraumexplosion. Das Generieren von *zustandsbasiertem Simulationscode* ist im Gegensatz zu den beiden anderen Ansätzen zwar sehr vielseitig, bringt jedoch durch viele Tests auf aktive Transitionen Effizienzprobleme mit sich.

Im Gegensatz zur Codegenerierung aus Petrinetzen gehört die Codegenerierung aus Statecharts zu einer lang erforschten und in der Praxis und Industrie verbreiteten Technik. Viele kommerzielle UML-Werkzeuge, wie IBM Rational Rhapsody[1], bieten eine Generierung von universell einsetzbarem ausführbarem Code für verschiedene Programmiersprachen. Hierbei wird eine Spezifikation vorausgesetzt, die eine Kombination aus Statecharts und explizitem Code in der Zielsprache ist.

In [Sam08] werden für die Codegenerierung aus Statecharts verschiedene klassische Implementierungskonzepte, die den Ansatz der Simulation verfolgen, vorgestellt. Zu diesen gehören unter anderem die *State Table*, das *State Design Pattern* und die *Nested-Switch-Statement*-Implementierung. Die *State Table* beschreibt eine Matrix zwischen Zuständen und Ereignissen und das *State Design Pattern* ist eine objektorientierte Variante, die auf Vererbung und Delegation basiert. Diese beiden Ansätze sind für eingebettete Systeme aufgrund des Speicherverbrauchs bzw. der objektorientierten Basis nicht geeignet. Im Gegensatz hierzu hat der *Nested-Switch-Statement*-Ansatz einen kleinen Speicherverbrauch, benötigt keine Hierarchie und ist einfach zu generieren. Er überprüft auf oberster Ebene der Switch-Anweisung den Zustand, der in einer skalaren Zustandsvariablen codiert ist, und wertet dann das aktuell zu verarbeitende Ereignis aus.

Die Generierung von effizientem Code aus Petrinetzen stellt, wie in [GV03] erläutert wird, eine größere Herausforderung dar und ist nicht weit verbreitet. Als Grundlage für die Codegenerierung aus MPNs wird eine abgewandelte Variante des simulativen *Nested-Switch-Statements* verwendet. Die Effizienzprobleme dieses Ansatzes für die Generierung von ausführbaren Programmen werden in [Phi06] adressiert und Lösungsvorschläge vorgestellt, die auf dem Konzept der strukturellen Analyse basieren. Bei der Generierung von Monitoren für eingebettete Systeme aus MPNs ergeben sich jedoch von [Phi06] nicht adressierte Herausforderungen, die im folgenden Abschnitt behandelt werden.

8.3 HERAUSFORDERUNGEN DER CODEGENERIERUNG AUS MPN-SIGNATUREN

Die in Abschnitt 6 eingeführte Definition der MPNs lässt Details der Implementierung offen. Da die Zielplattform unter anderem eingebettete Systeme sind, müssen ein niedriger Code- (ROM) und Datenspeicherverbrauch (RAM) sowie eine kurze Rechenzeit erreicht werden.

Die kritischen Aspekte bei der Implementierung der MPN-Definition sind:

1. Effiziente Ausnutzung der MPN-Semantik

[1] IBM-Website: http://www-03.ibm.com/software/products/us/en/ratirhapfami

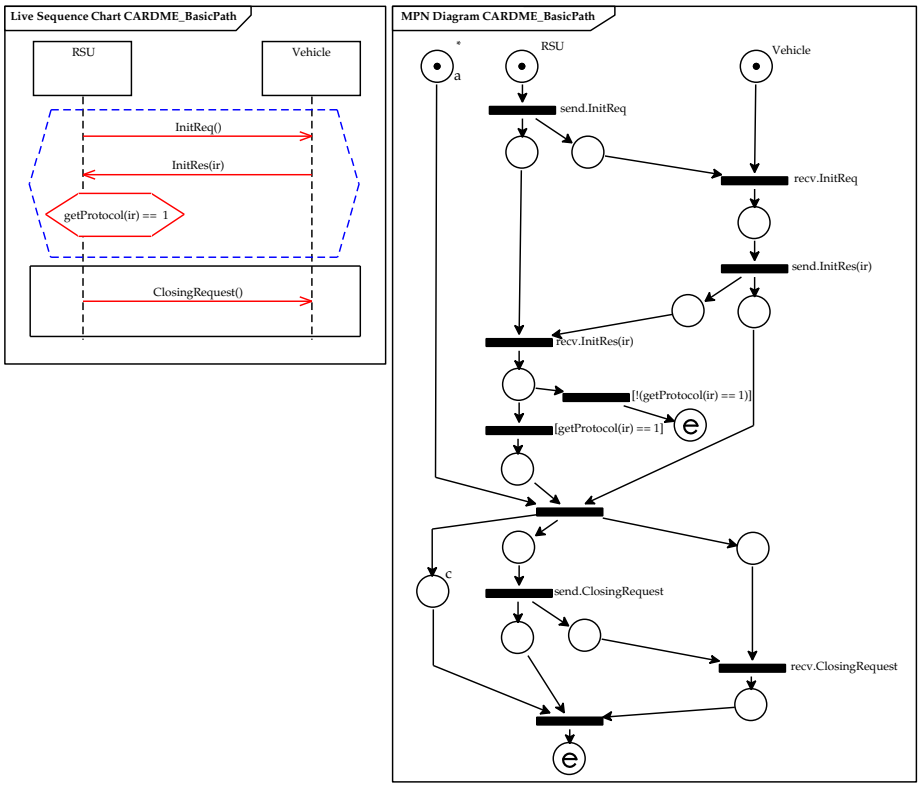


Abbildung 8.1: Einfacher Ausschnitt aus dem CARDME-Protokoll

2. Effiziente Speicherung der aktuellen Markierung
3. Effiziente Zuordnung und Identifizierung der Ereignisse
4. Effiziente Änderungsverfolgung der Markierung

Als Beispielsignatur wird der in Abbildung 8.1 dargestellte vereinfachte Ausschnitt des CARDME-Protokolls eingesetzt. Diese Signatur wird zusammen mit der in Abschnitt 7.3.3 vorgestellten PSI-Datei als Ausgangspunkt für die Codegenerierung genutzt. Die Signatur beschreibt einen positiven Ablauf, in dem nach einer erfolgreichen Initialisierungsphase ein *ClosingRequest* gesendet werden muss. Die *InitRes*-Nachricht trägt einen Parameter mit Informationen zu vom Fahrzeug unterstützen Protokollen mit sich. Diese muss das Protokoll 1 unterstützen.

Im Folgenden werden Lösungen zu den beschriebenen Herausforderungen vorgestellt, die es erlauben auch komplexe, parallele Muster, die in der MPN-Sprache spezifiziert sind, auf eingebetteten Systemen zu überwachen. Hierzu wird als Beispiel die Monitorgenerierung für die Zielsprache C verwendet.

1. Effiziente Ausnutzung der MPN-Semantik Die MPN-Semantik wurde so entworfen, dass nur ein Bruchteil der möglichen Kontrollflussenden explizit als Terminalplatz im MPN modelliert und damit implementiert werden muss. Dies wird durch die Auswertung nach Vorbedingung (antecedent) und nachfolgendem (consequent) Bereich bei nicht-schaltenden MPN-Instanzen erreicht.

Hierdurch verringert sich die Anzahl der zu verwaltenden Plätze, woraus ein niedrigerer Datenspeicherverbrauch resultiert. Die damit einhergehende Reduktion der Anzahl der zu überprüfenden Transitionen führt zu geringerer Rechenzeit und verringertem Codespeicherverbrauch.

Im Gegensatz zur Überprüfung, ob ein Platz der Menge der nachfolgenden Plätze belegt ist, findet in der hier gezeigten Implementierung die Überprüfung der Vorbedingungsplätze statt. Dies hat den Vorteil, dass in Signaturen, die keine Vorbedingung besitzen, diese Überprüfung der Belegung nach dem Nicht-Schalten der Signatur im generierten Monitor vollständig entfallen kann. Hierzu muss jedoch in den MPNs durch die Transformation sichergestellt sein, dass auf Vorbedingungsplätzen beim Übergang in den nachfolgenden Bereich keine Token liegen bleiben. Dies ist zum Beispiel bei der Transformation aus der MBSecMon-Sprache gegeben.

Die Zuordnung von Plätzen und Transitionen zu den Instanzen des zu überwachenden Systems erlaubt die für zentrale Monitore generierten MPNs auf ein oder mehrere ausgewählte Instanzen aufzuteilen und hierdurch nur Code für die relevanten Teilsignaturen zu generieren. Hierdurch kann der Speicherverbrauch auf den einzelnen Systemen deutlich reduziert werden.

Beispiel *Kompakte Spezifikation*

Das MPN in Abbildung 8.1 besitzt durch die in der MPN-Semantik enthaltene Auswertung nach aktueller Markierung des MPNs beim Nichtschalten auf ein Ereignis nur zwei explizit modellierte Endplätze. Müssten alle möglichen Enden des MPNs explizit modelliert werden, würde es zu einer deutlich höheren Anzahl von Terminalplätzen und Transitionen in der Signatur und damit im generierten Monitorcode führen.

2. EFFIZIENTE SPEICHERUNG DER AKTUELLEN MARKIERUNG Die Markierung der MPNs wird in einem zweidimensionalen (bei Subgenerationen in einem dreidimensionalen) Array gespeichert, dessen erste (bzw. zweite) Dimension die Eingabegenerationen (bzw. Subgeneration) widerspiegeln und die letzte die Belegung der Plätze des MPNs. Die Speicherung findet somit in einem Array in der folgenden Art und Weise statt:

$$\mathtt{Markierung}[\mathtt{Eingabegeneration}, (\mathtt{Subgeneration}), \mathtt{Platz}] \rightarrow \{0, 1\} \quad (8.1)$$

Aufgrund des limitierten Datenspeichers eingebetteter Systeme muss die maximale Anzahl der zu überwachenden Kommunikationsstränge eingeschränkt werden. Diese Einschränkung erlaubt eine Abschätzung der oberen Grenze des Datenspeicherverbrauchs zur Laufzeit. Es muss jedoch eine Strategie entwickelt werden, wie mit dem Erreichen dieser oberen Grenze umgegangen wird. Hierbei gibt es verschiedene Möglichkeiten wie die Ausgabe einer Fehlermeldung oder das Verwerfen der ältesten überwachten Kommunikation.

Die Anzahl der notwendigen Subgenerationen kann durch Analyse der Signaturen bestimmt werden, wenn wiederholende Ereignisse im MPN existieren. Ansonsten muss auch hier der Entwickler eine obere Grenze definieren.

Des Weiteren kann durch die Speicherung der Markierung der Plätze in einem Array je nach Zielsystem unnötig viel Datenspeicher belegt werden. Hier kann durch die bitweise Codierung mehrerer Plätze in einer Variablen (z. B. char) der Datenspeicherplatzbedarf je Platz auf bis zu 1 Bit reduziert werden. Dies führt jedoch zu zusätzlichen Bitoperationen, um die Änderungen der Platzbelegung zu schreiben und auszulesen.

Beispiel *Speicherung der Markierung*

Jedem Platz in einem MPN wird eine eindeutige aufsteigende natürliche Zahl zugeordnet. Die Speicherung der Markierung eines MPNs findet in einem Array mit Elementen des Typs *CARDME_BASICPATH_Places* statt, in dem je Platz des MPNs ein Bit verwendet wird. Bei einem MPN ohne Subgenerationen stellt die erste Dimension dieses Arrays die Eingabegenerationen dar, wobei die zweite Dimension die nach Platznummern gespeicherte Markierung der Plätze enthält. In der PSI-Datei ist die obere Grenze der Eingabegenerationen mit 5 spezifiziert und die Signatur verwendet keine Subgenerationen.

Auflistung 8.1: Speicherung der aktuellen Markierung

```
typedef struct CARDME_BASICPATH_plc {
  char list[3];
} CARDME_BASICPATH_Places;

//Helper for initialization of arrays
CARDME_BASICPATH_Places CARDME_BASICPATH_initplaces;
//Arrays, which contain active places of respective generation
CARDME_BASICPATH_Places CARDME_BASICPATH_activePlaces[5];
//Indicates which generation is active
char CARDME_BASICPATH_activeGenerations[1] = {0};
```

Auflistung 8.1 zeigt die Deklaration der Datenstrukturen zur Speicherung der Markierung der Plätze im MPN. *CARDME_BASICPATH_Places* ist ein Struct mit einem Array von *Chars*, das passend zu der Platzanzahl im MPN dimensioniert ist. Das MPN in Abbildung 8.1 hat 18 Plätze. Die Struktur *CARDME_BASIC_PATH_Places* enthält ein Char-Array der Größe 3. Dieses bietet Platz für die Speicherung von bis zu 24 Platzmarkierungen, da ein MPN nur ein Token pro Platz hält. Auch die Speicherung von schon im MPN aktiven Generationen wird bitweise mit der Variablen *CARDME_BASICPATH_activeGenerations* durchgeführt.

Die Änderungen auf dieser Speicherstruktur finden durch Bitoperationen statt. Bei der Verwendung von Subgenerationen wird eine weitere Dimension der Datenstruktur hinzugefügt.

3. Effiziente Zuordnung und Identifikation der Ereignisse Die Zuordnung eines Ereignisses zu den entsprechenden Transitionen kann, wie in den in Abschnitt 8.2 beschriebenen klassischen Ansätzen, direkt in den MPNs geschehen. Dies führt jedoch dazu, dass der Abgleich für jedes MPN und für jedes zu verarbeitende Ereignis ausgeführt wird. Durch die Auslagerung der Identifi-

kation des Ereignisses aus dem MPN in den Controller kann der Aufwand für die Zuordnung der Ereignisse auf eine Komplexität von $O(1)$ reduziert werden. Hierzu wird jedes Ereignis im Controller einmalig auf einen Wert eines kompakten Bereichs (Enumeration) übersetzt oder falls vorhanden schon passend aus dem System extrahiert. Dieses vorverarbeitete Ereignis dient dann als Eingabe für das MPN. Erst danach werden die Transitionen, die dieses Event besitzen, evaluiert. In einem Monitor mit festgelegten Schnittstellen für jedes Ereignis kann bei Aufruf der Schnittstelle direkt das entsprechende Literal der Ereignis-Enumeration dem Controller übergeben werden. Hierdurch werden die rechenzeitintensiven Auswertungen der Aktivierungsbedingungen an den Transitionen deutlich reduziert.

Beispiel *Schnittstelle mit direkter Bestimmung des Ereignisses*

Alle in der Monitorspezifikation vorkommenden Ereignisse werden gemeinsam in einer zentralen Enumeration für alle MPNs gespeichert. Hierdurch besitzen alle MPNs dieselbe Zuordnung zwischen Ereignis und dem kompakten Bereich. Auflistung 8.2 zeigt für einen Monitor, der die CARDME-Signatur beinhaltet, eine solche Enumeration.

Auflistung 8.2: Enumeration der Ereignisse

```
enum monitorEvents{
  EVENT_EPSILON,
  EVENT_SEND_INITREQ,
  EVENT_RECV_INITREQ,
  EVENT_SEND_INITRES,
  EVENT_RECV_INITRES,
  EVENT_SEND_PRESREQ,
  EVENT_RECV_PRESREQ,
  EVENT_SEND_CLOSINGREQUEST,
  EVENT_RECV_CLOSINGREQUEST,
  EVENT_LAST_ELEMENT
};
```

Die zusätzlichen nicht in der vorgestellten Signatur vorkommenden Ereignisse der Präsentationsphase resultieren aus einer weiteren Signatur im Monitor.

Diese Zuordnung der Ereignisse auf einem kompakten Bereich kann auch im weiteren Verlauf der Verarbeitung der Ereignisse ausgenutzt werden.

Beispiel *Auswertung der Ereignisse im MPN*

Der Controller des Monitors ruft, nachdem er die Eingaben auf interne Ereignisse abgebildet hat, die *evaluateEvent*-Methode der MPNs auf. Zur effizienten Zuordnung der Ereignisse zu den auszuwertenden Transitionen im MPN wird statt eines Switch-Statements ein Funktionspointer-Array eingesetzt. Auflistung 8.3 zeigt am Beispiel der Signatur diese Methode und das dazugehörige Funktionspointer-Array. Dieses Array enthält für jedes für die Signatur relevante Ereignis einen Funktionspointer auf die entsprechende spezielle ereignisspezifische *evaluateEvent*-Methode. Hierbei wird die durch die Enumeration aller Ereignisse vorgegebene Reihenfolge eingesetzt.

Auflistung 8.3: Funktionspointer-Array zur Ansteuerung der Auswertung der Ereignisse

```
functionPtrCARDME_BASICPATH CARDME_BASICPATH_functionArray[
    EVENT_LAST_ELEMENT] = {
 &CARDME_BASICPATH_evaluateEvent_EPSILON,
 &CARDME_BASICPATH_evaluateEvent_SEND_INITREQ,
 &CARDME_BASICPATH_evaluateEvent_RECV_INITREQ,
 &CARDME_BASICPATH_evaluateEvent_SEND_INITRES,
 &CARDME_BASICPATH_evaluateEvent_RECV_INITRES,
 &0,
 &0,
 &CARDME_BASICPATH_evaluateEvent_SEND_CLOSINGREQUEST,
 &CARDME_BASICPATH_evaluateEvent_RECV_CLOSINGREQUEST
};

//Evaluate an event with a context on the MPN
char CARDME_BASICPATH_evaluateEvent(unsigned char event, unsigned char
    generation){
 char result = 0;
 result = (*CARDME_BASICPATH_functionArray[event])(generation) || result;
 if (result) CARDME_BASICPATH_applyChanges(generation);
 return result;
}
```

Die *evaluateEvent*-Methode des MPNs ruft zunächst die dem Ereignis entsprechende Methode auf und wendet die dabei entstehenden Änderungen falls nötig danach mit der *applyChanges*-Methode an.

In den MPNs kann bestimmt werden, ob die Signaturen vollständig modelliert wurden oder nur die modellierten Ereignisse beachtet werden sollen. Falls ein MPN nur mit eingeschränkten Ereignistypen überwacht werden soll, kann das Funktionspointer-Array zur Bestimmung, ob ein Ereignis für ein MPN relevant ist, herangezogen werden. Liefert das Array eine 0 für ein Ereignis zurück, muss das Ereignis nicht vom MPN verarbeitet werden.

Basierend auf den bisher vorgestellten Konzepten wird in der ereignisspezifischen *evaluateEvent*-Methode der Vorbereich der Transitionen mit diesem Ereignis überprüft und eventuell Bedingungen an den Transitionen ausgewertet.

Beispiel *Schalten einer Transition*

Für jedes Ereignis im MPN wird wie beschrieben eine *evaluateEvent*-Methode generiert. Die Implementierung für das Ereignis *send.InitReq* ist in Auflistung 8.4 dargestellt. In dieser werden die Plätze im Vorbereich aller Transitionen, an denen dieses Ereignis annotiert ist, überprüft. In diesem Fall ist dies nur eine Transition mit einem Platz im Vorbereich. Wenn alle Plätze im Vorbereich mit einem Token belegt sind, werden Änderungen, die durch diese Transition ausgelöst werden, in den Arrays *deletedToken* und *createdToken*, die für alle MPNs im Monitor gemeinsam genutzt werden, zwischengespeichert, bis alle Transitionen in diesem Makroschritt geschaltet haben. Erst danach löst der Controller

eine Übertragung dieser Änderungen auf die aktuelle Markierung des MPNs aus.

Auflistung 8.4: Auswertung und Schalten einer Transition

```
int CARDME_BASICPATH_evaluateEvent_SEND_INITREQ(unsigned char generation){
 int result = 0;
 if (((CARDME_BASICPATH_activePlaces[generation].list[0]&1) == 1)){
  //Store changes to the marking
  deletedToken[nextChangeDelete++] = 0;
  createdToken[nextChangeCreate++] = 3;
  createdToken[nextChangeCreate++] = 4;
  //Action
  //fprintf("CARDME_BASICPATH generation %d transition 7 fired\n",
      generation);
  result = 1;
 }
 return result;
}
```

4. EFFIZIENTE ÄNDERUNGSVERFOLGUNG DER MARKIERUNG In der Definition der MPN-Semantik in Kapitel 5 wird eine Kopie der aktuellen Markierung des MPNs angelegt, auf der die Analysen durchgeführt werden. Dies führt jedoch zu einer Verdopplung des linear mit der Anzahl der Plätze des MPNs wachsenden Datenspeicherverbrauchs. Zur Optimierung wird ein Mechanismus eingeführt, der nur die Änderungen protokolliert und anschließend, nachdem das Ereignis verarbeitet wurde, die Änderungen an der aktuellen Markierung durchführt. Die obere Grenze der zu protokollierenden Änderungen wird vor der Generierung des Codes für jedes MPN analysiert. Diese Analyse, die in Abschnitt 8.5 genauer beschrieben wird, basiert auf dem Überdeckungsgraphen der MPNs und erlaubt es eine obere Grenze an zu löschenden und zu erzeugenden Token zu bestimmen. Dieser Änderungsspeicher wird gemeinsam für alle MPNs und zu überwachenden Generationen im Controller zur Verfügung gestellt.

Beispiel *Änderungsverfolgung*

In Auflistung 8.5 ist sowohl die Datenstruktur als auch die Methode zur Übertragung der Änderungen dargestellt. Zur Änderungsverfolgung sind zentral im Controller zwei Arrays für die Speicherung von Änderungen in einem beliebigen MPN im Monitor definiert. Des Weiteren existieren für jedes der Arrays ein Zähler, der den aktuellen Befüllungsstand repräsentiert. Diese Zähler werden sowohl bei der Speicherung der Änderungen der Belegung als auch bei der Übertragung der Änderungen auf die aktuelle Markierung des MPNs verwendet.

Zur Übertragung der Änderungen ist die *applyChanges*-Methode zuständig, die, nachdem ein Ereignis von einem MPN verarbeitet wurde, die Änderungen auf die Markierung mittels Bitoperationen überträgt.

Auflistung 8.5: Änderungsverfolgung und Übertragung der Änderungen

```
int deletedToken[3];
int createdToken[3];
int nextChangeDelete = 0;
int nextChangeCreate = 0;

void CARDME_BASICPATH_applyChanges(unsigned char generation){
 int i;
 for(i = 0; i < nextChangeDelete; i++){
  CARDME_BASICPATH_activePlaces[generation].list[(deletedToken[i])/8] &=
      ~(1<<(deletedToken[i]%8));
 }
 for(i = 0; i < nextChangeCreate; i++){
  CARDME_BASICPATH_activePlaces[generation].list[(createdToken[i])/8] |=
      1<<(createdToken[i]%8);
 }
 nextChangeDelete = 0;
 nextChangeCreate = 0;
```

Nachdem die grundlegende Übersetzung der MPNs zu Code in diesen Abschnitt beschrieben wurde, wird im nächsten Abschnitt auf die Übersetzung von Annotationen an Transitionen wie Bedingungen und Aktionen eingegangen.

8.4 Generierung von plattformspezifischen Monitoren aus MPNs

Die Generierung von Monitoren für verschiedene Zielplattformen endet nicht damit die passende Zielsprache bei der Generierung aus Monitor-Petrinetzen zu verwenden. Eine weitere Herausforderung bei der Umsetzung von Monitor-Petrinetzen in Code ist die Übersetzung von Annotationen in die Zielsprache. Dies betrifft die Annotationen an den Transitionen der MPNs, wie Aktionen und Bedingungen, die in die Zielsprache übersetzt werden müssen.

Die Programmiersprachenabhängigkeit betrifft u. a. Funktionen, Vergleiche und Stringvergleiche sowie deren Negation und Stringkopien, die in die Notation der Zielsprache übersetzt werden müssen.

Beispiel *Stringvergleich in verschiedenen Zielsprachen*

Häufig müssen an Transitionen komplexe Vergleiche zwischen Werten verschiedener Datentypen durchgeführt werden. Die Beschriftung an einer Transition könnte für einen Stringvergleich folgendermaßen aussehen: [str1==str2] Die Variablen str1 und str2 sind hierbei Zeichenketten. Das korrekte Codefragment für den Vergleich unterscheidet sich in den Zielsprachen Java und C deutlich.

Java: if (str1.equals(str2))
C: if (strcmp(str1, str2)== 0)

Hierbei ist es notwendig während der Codegenerierung zu erkennen, dass es sich um einen Stringvergleich handelt und es sich somit bei den Variablen nicht um Integer-Variablen sondern um String-Variablen handelt. Auch die Negation

dieser Ausdrücke für die Modellierung von Alternativen wie [!(str1==str2)] im MPN unterscheiden sich in den verschiedenen Zielsprachen.

Java: if (!str1.equals(str2))
C: if (strcmp(str1, str2)!= 0)

Die Annotationssprache der MPNs ist nicht der Fokus dieser Arbeit, weshalb hier nur grundlegend auf die Verarbeitung dieser Annotationen eingegangen wird. Zur Vereinfachung des Übersetzungsprozesses kann im MBSecMon-Prozess eine beliebige Annotationssprache gewählt werden, die im Extremfall auch die Zielsprache sein kann. Eine solche Wahl würde jedoch die Zielsprachenunabhängigkeit der Spezifikation aufheben. Als Standard ist im MBSecMon-Prozess eine einfache als ANTLR-Grammatik definierte Sprache implementiert, die je nach Komplexität der zu beschreibenden Signaturen erweitert werden kann. Hierbei werden die Ausdrücke in Teile wie Variablennamen, logische Operatoren, mathematische Operatoren und Methodennamen und ihre Parameter aufgeteilt und dem Übersetzungsmodul des Codegenerators übergeben.

Beispiel *Parsen einer Bedingung*

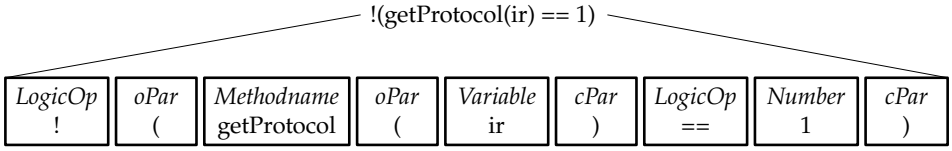


Abbildung 8.2: Parsen einer logischen Operation

Abbildung 8.2 zeigt, wie ein komplexer Ausdruck der Annotationssprache in seine Teilausdrücke zerlegt wird. Zur weiteren Verarbeitung werden die Teilausdrücke mit ihrem Typ (*LogicOp*, *Variable*, ...) zusammen an den Übersetzer übergeben, der dann unter Einbeziehung der PSI-Datei Quellcodefragmente in der Zielsprache für den Codegenerator erzeugt.

Der Eintrag aus der PSI-Datei in Auflistung 7.1 <entry key="CARDME_BasicPath .context_method.getProtocol">getProtocolID</entry> führt, falls sich der Ausdruck im entsprechenden MPN befindet, dazu, dass der Methodenname aus der Annotation im MPN getProtocol durch die auf dem Zielsystem vorhandene Methode getProtocolID ersetzt wird.

Dieses Übersetzungsmodul ändert auf Basis der grundlegenden Regeln für die Zielsprache und den zusätzlichen plattformspezifischen Informationen (PSI) die ihm übergebenen Ausdrücke zu zielsprachen- und zielplattformspezifischen Code.

Die Übersetzung der MPNs wie ihre statische Struktur und ihre Semantik ist für jede Programmiersprache nur einmal zu definieren. Auch zielsprachenspezifische Übersetzungen der annotierten Ausdrücke sind als Bibliothek (als wiederverwendbare PSI-Datei) in den Generierungsprozess einbindbar.

Eine weitere Herausforderung ist die Plattformabhängigkeit, die u. a. die Namensgebung von Methoden, Variablen und die durchgängige Verwendung plattformspezifischer Datentypen verlangt.

Beispiel *Plattformspezifische Schnittstellen und Datentypen*

Die folgenden Einträge stammen aus Auflistung 7.1.

```
<entry key="event.RECV_INITRESPONSE">EVENT_SEND_INITRES</entry>
<entry key="event.RECV_INITRESPONSE.param">ir_param</entry>
<entry key="event.RECV_INITRESPONSE.param_type">IRData</entry>
<entry key="event.RECV_INITRESPONSE.param_init">{0,""}</entry>
```

Sie beschreiben die Abbildung von externen zielsystemkonformen Schnittstellen auf die in einem Monitor verwendeten Ereignisse. Zusätzlich wird noch der Name des Parameters der Schnittstelle und dessen Typ *IRData* definiert. Der Typ ist dabei ein aus der Systemspezifikation gewonnener festgelegter komplexer Datentyp.

Der folgende Ausschnitt (Auflistung 8.6) zeigt eine generierte plattformspezifische Schnittstelle, in der nach der Zwischenspeicherung des übergebenen Parameters ein Ereignis als Literal der *MonitorEvents*-Enumeration dem Monitor übergeben wird. Somit kann der Monitor ereignisspezifische Schnittstellen nach außen zur Verfügung stellen, die durch Intrumentierung des zu überwachenden Systems angesprochen werden.

Auflistung 8.6: Ereignisspezifische Monitorschnittstelle

```
void dispatchEvent_RECV_INITRESPONSE(IRData* param, int gen){
 contextTemp.ir_param[gen]=(IRData)* param;
 CARDME_BASICPATH_dispatchEvent(EVENT_RECV_INITRES, gen);
}
```

Im Monitor selbst wird zur Zwischenspeicherung des Variablenwerts ir_param der Typ IRData verwendet.

Diese plattformspezifischen Anpassungen des generierten Monitors sind besonders bei Einbindung in ein zu überwachende System wie AUTOSAR, das in auf Bezug auf zu verwendenden Schnittstellen und Datentypen sehr restriktiv ist, notwendig. Hierbei werden auf Basis der im Prozess extrahierten PSI u. a. Datentypen aus dem zu überwachenden System übernommen und durchgängig im Monitor eingesetzt. Des Weiteren müssen standardkonforme Schnittstellen generiert werden.

Neben der hier betrachteten Umsetzung in die prozedurale Programmiersprache *C* wurden noch Monitorgenerierungen für die objektorientierte Programmiersprache *Java* und in die Hardwarebeschreibungssprache *VHDL* implementiert. Zusammenfassend ist der Generierungsaufwand des Monitors in eine Hardwarebeschreibungssprache wie VHDL im Gegensatz zu einer prozeduralen oder objektorientierten Programmiersprache deutlich geringer. Dies resultiert zum einen aus dem eingeschränkten Funktionsumfang und zum anderen daraus, dass viele Teile, wie der Controller, durch die Generierung der parallel zueinander liegenden MPN-Beschreibungen bis auf die Verknüpfung der Signale zu den einzelnen MPN-Instanzen wenig an die tatsächlich eingebundenen Signaturen angepasst werden muss. Der Aufwand der Implementierung eines Codegenerators steigt mit speziellen Anforderungen an den Code des Zielsystems bzw. der Zieldomäne, sei es durch einzuhaltende Standards oder eingeschränkte Hardwareressourcen.

Hierdurch müssen dem Generierungsprozess deutlich mehr plattformspezifische Informationen zur Verfügung gestellt werden.

8.5 Optimierung des MPN-Codes

Die in Abschnitt 5.3.2 in Definition 28 beschriebene MPN-Semantik und ihre Erweiterung ist darauf angewiesen, die Änderungen in einem MPN basierend auf der alten Markierung des MPNs durchzuführen. Um das vollständige Kopieren der aktuell zur Verarbeitung des Ereignisses genutzten Markierung zu verhindern, wird in den Implementierungen ein Änderungsspeicher verwendet, in dem die Änderungen bis zum Abschluss der Verarbeitung eines Ereignisses zwischengespeichert werden. Erst danach werden die Änderungen auf die Markierung des MPNs angewendet.

Zur Bestimmung der maximalen Anzahl möglicher Änderungen der Platzbelegung im MPN während einer Ereignisverarbeitung existieren verschiedene Möglichkeiten. Zum einen eine einfache statische Analyse der MPN-Signaturen und zum anderen die Verwendung eines Markierungsgraphen, wie er für die Analyse von Petrinetzen eingesetzt wird.

Statische Analyse Bei der statischen Analyse wird die Anzahl der Plätze im Vor- bzw. Nachbereich aller Transitionen im MPN betrachtet. Die Plätze im Vor- bzw. Nachbereich der Transitionen mit demselben annotierten Ereignis werden addiert und anschließend das Maximum für die Anzahl der Plätze bestimmt.

Markierungsgraph/Überdeckungsgraph Bei der Nutzung eines Markierungsgraphen (Abschnitt 3.2.3) wird dieser eingesetzt, um die partielle Ordnung der MPNs in die Berechnung einzubeziehen. Der Markierungsgraph wird basierend auf den erreichbaren Markierungen und den Transitionen des MPNs (zunächst ohne Betrachtung der Ereignisse) aufgebaut. Basierend darauf wird jede Markierung betrachtet und es werden alle ausgehenden Transitionen verfolgt. Bei Transitionen, die dasselbe Ereignis annotiert haben, werden die Plätze im Vor- bzw. Nachbereich addiert. Aus der Menge der Ergebnisse wird die maximale Anzahl möglicher Änderungen in einem Schritt für den Vor- und den Nachbereich bestimmt.

Beispiel *Ein Monitor-Petrinetz und sein Markierungsgraph*

Abbildung 8.3 zeigt ein MPN zusammen mit seinem Markierungsgraphen. Dieser basiert sowohl auf den Transitionen mit als auch ohne annotiertem Ereignis. Epsilon-Transitionen werden explizit als ein Ereignistyp betrachtet. Der Graph beginnt mit der Anfangsmarkierung und zeigt alle erreichbaren Markierungen des MPNs. Dabei sind die entsprechenden Ereignisse, die den Markierungswechsel auslösen, an den Übergängen annotiert.

Mit der statischen Analyse bzw. basierend auf dem Markierungsgraphen findet die Bestimmung der Anzahl der maximal auftretenden Änderungen in einem Verarbeitungsschritt des MPNs statt.

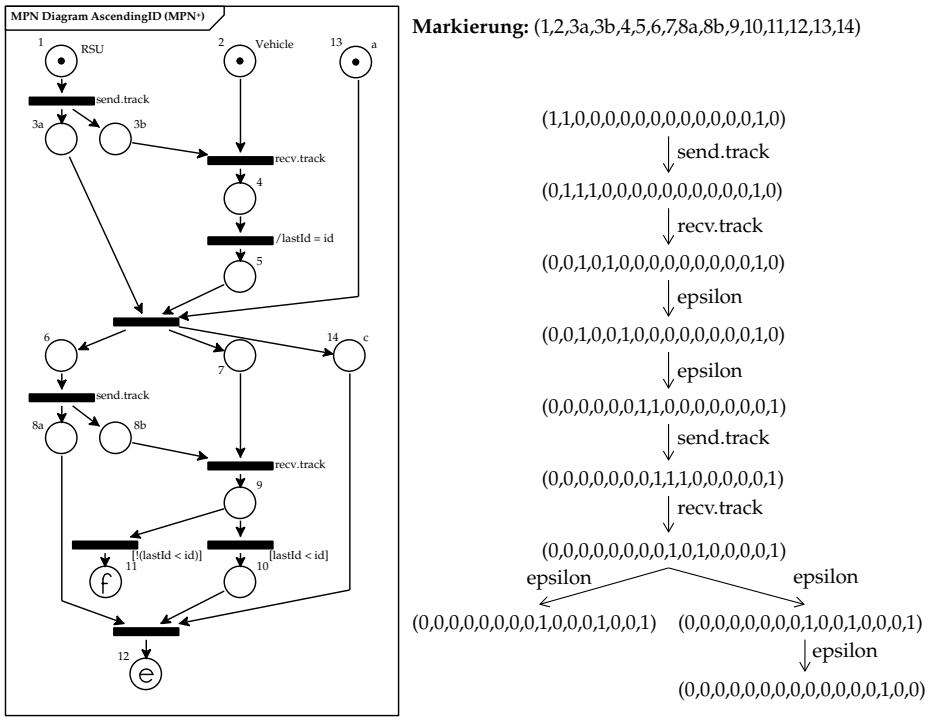


Abbildung 8.3: Markierungsgraph eines MPNs

Beispiel *Analyse des Petrinetzes auf benötigte Änderungsspeicherplätze*

Im Folgenden wird die ermittelte maximale Anzahl von Änderungen einer Markierung als Paar dargestellt. Es wird zwischen gelöschten und erzeugten Token unterschieden und dies im Format *(#gelöscht, #erzeugt)* notiert.

Das in Abbildung 8.3 gezeigte MPN würde nach der ersten Variante folgende Werte bestimmen:

send.track: (1+1, 2+2) → (2, 4)
recv.track: (2+2, 1+1) → (4, 2)
epsilon: (1+3+1+1+3, 1+3+1+1+1) → (9, 7)

Nach dieser Analyse würden folglich Speicher für 9 Löschaktionen und 7 Erzeugungsaktionen reserviert werden. Dies ist ein unnötig hoher Wert und resultiert aus vielen Epsilontransitionen im MPN.

Die zweite Methode unter Einbeziehung des Markierungsgraphen liefert im Gegensatz hierzu folgende Ergebnisse:

send.track: {(1, 2); (1, 2)} → (1, 2)
recev.track: {(2, 1); (2, 1)} → (2, 1)
epsilon: {(1, 1); (3, 3); (1, 1); (1+1, 1+1); (3, 1)} → (3, 3)

Hierbei werden alle möglichen Markierungen betrachtet und aus derselben Markierung ausgehende Kanten mit identischem Ereignis durch Addition zusammengefasst. Das Ergebnis sind somit maximal 3 Löschaktionen und 3 Erzeugungsaktionen, die unter Einbeziehung der partiellen Ordnung im MPN entstehen können. Auch dieser Ansatz liefert nicht immer ein optimales Ergebnis. Wenn Bedingungen wie an der Transition im Nachbereich von Platz 9 auftreten und mehrere Plätze im Vor- und Nachbereich der Transition existie-

ren, kann dies durch die Addition der Plätze zu einer zu hohen Anzahl führen. In diesem Beispiel ist diese Ungenauigkeit durch die Epsilontransition mit den Werten (3, 3) zwischen Vorbedingung und Hauptteil verdeckt worden.

Während die statische Analyse in vielen Fällen schnell zufriedenstellende und korrekte Ergebnisse liefert, ist die Berechnung des Markierungsgraphen deutlich rechenintensiver, liefert jedoch deutlich bessere Ergebnisse. Statt des Markierungsgraphen kann für MPNs ein gewöhnlicher Überdeckungsgraph der P/T-Netze genutzt werden, da paralleles Schalten in einer Ereignisverarbeitung durch das Zusammenfassen von Transitionen mit demselben Ereignis, die aus derselben Markierung ausgehen, in der vorgestellten Analyse abgefangen wird. Schleifen im MPN führen bei der Berechnung des Überdeckunsgraphen zu keinen neuen Markierungen und die Verfolgung wird somit abgebrochen.

Analysetechniken auf Petrinetzen leiden größtenteils unter dem Komplexitätsproblem [Mur89]. Alle Entscheidungsalgorithmen zur Bearbeitung des Ereichbarkeitsproblems, somit auch die Erstellung Markierungsgraphen sind mindestens EXPSPACE kompliziert. Sie haben also einen exponentiellen Speicherbedarf. Der Markierungsgraph entspricht bei MPNs für diese Analyse dem Überdeckungsgraphen, da in einem MPN Token derselben Generation verschmolzen werden und damit MPNs 1-beschränkt sind. Die Berechnung der Anzahl der benötigten Änderungsspeicherplätze findet während der Monitorgenerierung und auf einer überschaubaren Menge an MPN-Signaturen, deren Überdeckungsgraph eine begrenzte Anzahl Markierungen beinhaltet, statt.

Die vorgestellt Analyse über den Überdeckungsgraphen führt im Gegensatz zur statischen Analyse in MPNs mit vielen Transitionen, die dasselbe Ereignis besitzen, zu einer deutlich besseren oberen Abschätzung der benötigten Änderungsspeicherplätze. Da nur Ereignisse bei der Analyse betrachtet werden, und andere Annotationen wie Bedingungen und Aktionen ignoriert werden, führt dieser Ansatz immer zu einer ausreichenden Menge an Änderungsspeicherplätzen, die jedoch in der Regel nicht minimal ist.

9

EVALUATION

In diesem Kapitel findet die Evaluation der Monitor-Petrinetze als Zwischensprache und der Monitorgenerierung aus ihnen statt. Hierbei werden insbesondere die Generierung für verschiedene Zielsprachen (C, Java) und Zielplattformen wie eingebettete Systeme, PC und rekonfigurierbare Hardware, betrachtet. Auf Basis von Laufzeit und Speicherbedarf werden diese Implementierungen verglichen.

Hierzu werden zunächst in Abschnitt 9.1 die Java- und die C-Implementierung gegenübergestellt. Diese sind mit einem grundsätzlich unterschiedlichen Ziel entwickelt worden. Während die Java-Implementierung stark auf die Anwendung von Objektorientierung setzt, ist die C-Implementierung stark optimiert und setzt soweit wie möglich auf primitive Datentypen, um in Laufzeit und Speicherverbrauch auch auf Mikrocontrollern einsetzbar zu sein. Der Hauptfokus dieser Betrachtung ist die Skalierbarkeit des MPN-Ansatzes als Zwischensprache für die Codegenerierung. Nach der Evaluation auf einem PC wird die C-Implementierung auch auf einem Evaluationsboard mit Mikrocontroller auf Skalierbarkeit und Einsetzbarkeit getestet.

Die Semantik der MPNs unterstützt das parallele Schalten von Transitionen in einem Makroschritt. In der Implementierung für klassische Programmiersprachen werden diese Transitionen sequenziell verarbeitet und wenn nötig geschaltet. Abschnitt 9.2 betrachtet aus diesem Grund eine Implementierung der Monitore in der Hardwarebeschreibungssprache VHDL auf einem FPGA und vergleicht diese mit der optimierten C-Implementierung auf einem Mikrocontroller. Diese VHDL-Implementierung erlaubt ein paralleles Schalten aller Transitionen, was zu hoher Performanz führt.

In Abschnitt 6.6 wurde vorgestellt, dass ganzheitlich modellierte Monitorspezifikationen nach Übersetzung von der Spezifikationssprache in die MPN-Sprache verteilt generiert werden können. Abschnitt 9.3 betrachtet die Auswirkung getrennter Monitore gegenüber vollständiger Monitore.

In Abschnitt 4 wurde das Monitor-oriented Programming (MOP) als verwandter Ansatz zum MBSecMon-Prozess eingeführt und es wurden die Konzepte verglichen. Hierbei handelt es sich um einen sehr ähnlichen Ansatz mit teilweise ähnlichen Zielen. In Abschnitt 9.4 werden die generierten Monitore aus der Instanziierung JavaMOP des MOP und die um einen aspektorientierten Anteil erweiterte Java-Monitorgenerierung des MBSecMon-Ansatzes gegenübergestellt.

Bisher wurden die einzelnen Monitorinstanzen nur in einfachen Szenarien eingesetzt. Abschnitt 9.5 zeigt den Einsatz von durch den MBSecMon-Prozess generierten Monitoren in einer in Bezug auf Schnittstellen und Datentypen restriktiven, standardisierten Softwarearchitektur – die AUTOmotive System ARchitec-

ture (AUTOSAR) und beleuchtet die notwendigen Anpassungen am Generierungsprozess. Hier wird betrachtet, wie weit die automatisierte Monitorgenerierung für diesen Standard angepasst werden muss, um die Spezifikation für die automatische Generierung zu nutzen. Es wird erreicht, dass auch komplexe Monitore in AUTOSAR-Systemen eingesetzt werden können.

Im MOP werden neben den Kontextfreien Grammatiken als Spezifikationssprache Endliche Automaten (FSM) sowohl als Spezifikationssprache als auch als Zwischensprache zur Repräsentation vieler anderer Spezifikationssprachen eingesetzt. Somit müssen alle Sprachen, die in FSMs übersetzt werden auch die Nachteile von FSMs in der eigentlichen Implementierung teilen. Abschnitt 9.6 vergleicht anhand der parallelen Konzepte der MBSecMonSL die Vor- und Nachteile der FSMs gegenüber den MPNs als Zwischensprache.

Abschließend wird in Abschnitt 9.7 eine erste Evaluation mit einem Teil des in Abschnitt 6.2 eingeführten Referenzsystems, durchgeführt. In dieser wird beispielhaft eine Signatur mit Use- und Misusecase als unabhängige Teilsignaturen und mit Einsatz des Referenzsystems auf Laufzeit und Speicherverbrauch verglichen.

9.1 Vergleich von Java- und C-Monitoren

In diesem Abschnitt ist das Ziel eine objektorientierte Java-Implementierung mit einer prozeduralen C-Implementierung zu vergleichen. Eine Evaluation am Beispiel des CARDME-Protokolls würde nur zu einem einzigen Vergleichswert führen und Codefragmente, die in den Signaturen modelliert sind, enthalten. Diese Codefragmente, deren Laufzeit maßgeblich von der verwendeten Zielsprache abhängt, würden die Ergebnisse der Skalierbarkeitsanalyse verzerren. Deshalb werden anhand eines konstruierten Beispiels die beiden Implementierungen verglichen und die Skalierbarkeit des Ansatzes evaluiert.

9.1.1 *Ziel*

Das Ziel dieser Evaluation ist die Unterschiede im Konzept der verschiedenen Umsetzungen der MPNs in Code und deren Auswirkungen auf Laufzeit und Speicherverbrauch zu untersuchen. Des Weiteren soll die Skalierbarkeit der MPNs bei wachsenden Netzgrößen untersucht werden. Diese ist insbesondere für den Einsatz auf eingebetteten Systemen mit Mikrocontroller von Bedeutung.

Bei diesem Vergleich wird der Speicherverbrauch nicht zwischen Java- und C-Implementierung verglichen, da dieser durch das objektorientierte Konzept der Java-Implementierung deutlich höher ausfällt. Die C-Implementierung hat einen vorher festgelegten Datenspeicherbedarf, während die Java-Implementierung dynamisch neue Objekte zur Laufzeit anlegt. Der Daten- sowie der Codespeicherbedarf wird in Abschnitt 9.1.4 für eingebettete Systeme genauer betrachtet, um die Anwendbarkeit des Generierungsansatzes aus MPN-Spezifikationen zu evaluieren.

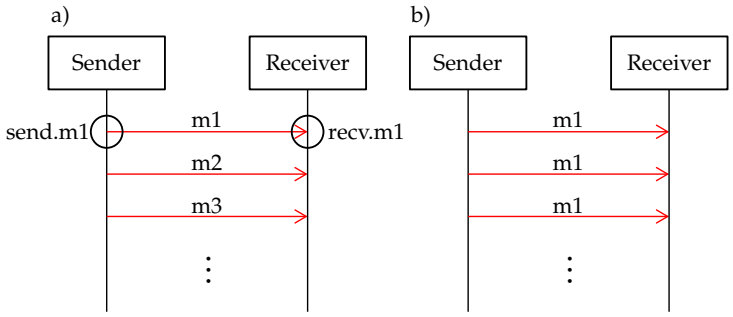


Abbildung 9.1: Konstruierte Signaturen zur Messung der Laufzeit

9.1.2 *Monitorsignaturen*

Zum Vergleich der beiden Implementierungen der Monitore in Java und in C werden folgende Monitorsignaturen verwendet. Abbildung 9.1 zeigt zwei Signaturtypen, die zwei Extreme in der Ausführung von MPN-Signaturen darstellen. Zum einen kommunizieren zwei Instanzen asynchron miteinander (a), wobei jede der Nachrichten einen anderen Nachrichtentyp besitzt. Zum anderen besitzt im zweiten Signaturtyp (b) jede Nachricht den selben Nachrichtentyp.

In der MPN-Repräsentation ergibt sich somit ein MPN, in dem jede Transition einen anderen Ereignistyp besitzt bzw. ein MPN, das nur zwei Ereignistypen im gesamten MPN besitzt. Bei der Verarbeitung der Ereignisse eines MPNs werden alle Transitionen, die mit dem übergebenen Ereignistyp annotiert sind, geprüft, ob sie schalten können. Hierdurch repräsentiert der erste Signaturtyp das untere und der zweite Signaturtyp das obere Extrem bei der Verarbeitung von Ereignissen in dem generierten Monitor.

Die hier verwendeten Signaturen besitzen keine zusätzlichen Annotationen an den Transitionen wie Bedingungen und Aktionen, die in die Zielsprache übersetzt werden. Hierdurch werden die MPN-Konzepte an sich betrachtet.

9.1.3 *Messungen auf dem PC*

Aus den Monitorsignaturen werden MPNs generiert, die anschließend zur Generierung von Java- und C-Code genutzt werden. Diese Monitore werden dann mit Ereignissen stimuliert und es werden die Laufzeit und der Speicherverbrauch gemessen. Die Messungen werden auf einem PC[1] durchgeführt. Es werden 100 komplette Durchläufe durch den Monitor durchgeführt und die Laufzeit gemessen. Hierfür werden die Win32-API-Funktionen QueryPerformanceCounter (QPC) und QueryPerformanceFrequency genutzt, die eine CPU-Tick-Granularität bieten. Es werden für jedes Eingabemodell 20 Messwerte bestimmt und der Median über diese gebildet.

[1] CPU: Intel Core2 Duo, P8600, 2.40 GHz; Arbeitsspeicher: 8 GB RAM; Betriebssystem: Windows 7 Professional 64 Bit

JAVA Zur Messung des Laufzeitverhaltens in Java ist zu beachten, dass die Java-VM[2] zur effizienten Ausführung von Code Optimierungsstrategien einsetzt. Ein Beispiel hierfür ist der Java HotSpot Client VM's Adaptive Compiler[3], der während der Ausführung eines Programms oft verwendete Teile des Codes nicht interpretiert, sondern in nativen Code kompiliert. Hierbei werden auch Techniken wie Inlining eingesetzt, um die Ausführungsgeschwindigkeit weiter zu erhöhen. Diese Maßnahmen führen zu starken Laufzeitschwankungen während der Messungen. Ein Abschalten dieser Optimierungsstrategien führt zwar zu einer konstanteren Laufzeit, jedoch auch zu starken Leistungseinbußen, bis zu einem Faktor von 10.

Um dem entgegenzutreten, wird im Folgenden für die Messungen ein Vorlauf vor die eigentliche Messung geschaltet, die dafür sorgt, dass die Programmausführung eingeschwungen ist. Hierzu wird der Monitor 1000-mal komplett durchlaufen. Erst dann findet die eigentliche Laufzeitmessung mit 100 Durchläufen statt. Dies ist ein realistisches Szenario für einen Monitor, da dieser während der gesamten Laufzeit des Programms parallel mitläuft.

Zur Messung wird die System.nanoTime()-Methode, die auf dem QPC basiert und die Zeit in einer Auflösung von Nanosekunden bietet genutzt. Trotz dieser Maßnahmen kann in Java jedoch eine Verfälschung der Werte durch den Garbage-Collector nicht ausgeschlossen werden, da dieser besonders während längerer Messungen immer wieder anspringt. Die hierdurch entstehende Streuung ist jedoch typisch für die Ausführung eines Java-Programms.

C Für die Messung in C werden direkt die Methoden der Win32-API genutzt, um die vergangene Zeit zu messen. Hier ist ein Vorlauf nicht notwendig, da C-Code direkt in Maschinencode kompiliert wird und zur Laufzeit keine Optimierungen mehr stattfinden. Zur Kompilierung des Monitorcodes wird die Cygwin GCC-Werkzeugkette mit Standardeinstellungen verwendet.

AUSWERTUNG Abbildung 9.2 und Abbildung 9.3 zeigen die Laufzeit, die für die Verarbeitung eines Ereignisses benötigt wird. Bei steigender Anzahl von Transitionen im Monitor werden die beiden in Abschnitt 9.1.2 beschriebenen Signaturen betrachtet. Diese beiden Signaturtypen bilden wie erwähnt zwei Extreme bei der Struktur einer Signatur ab. Die höchsten Abweichungen der Messwerte sind als vertikale Fehlerbalken dargestellt und es wurde der Median über die Messwerte gebildet.

Die Implementierung in C (Abb. 9.2) zeigt hier in beiden Fällen einen linearen Anstieg in der Laufzeit. Der flache Anstieg der Laufzeit zwischen 100 und 200 Transitionen in der Signatur lässt sich auf die Verarbeitung auf dem PC zurückführen, da in Abschnitt 9.1.4 in diesem niedrigen Bereich auf einem Mikrocontroller ein durchgängiges lineares Verhalten auftritt.

Im Gegensatz dazu zeigt die weniger optimierte, auf objektorientierter Programmierung aufbauende Implementierung in Java (Abb. 9.3) bei der Signatur

[2] Oracle Java SE, Vers. 1.7_25
[3] Java Virtual Machine Technology: http://docs.oracle.com/javase/7/docs/technotes/guides/vm/

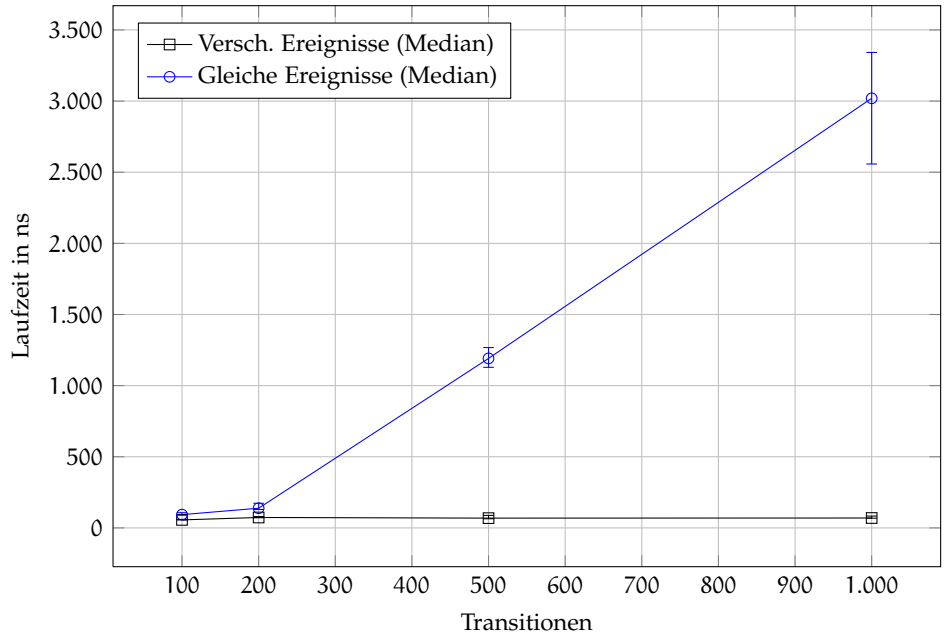


Abbildung 9.2: Laufzeiten der C-Monitore

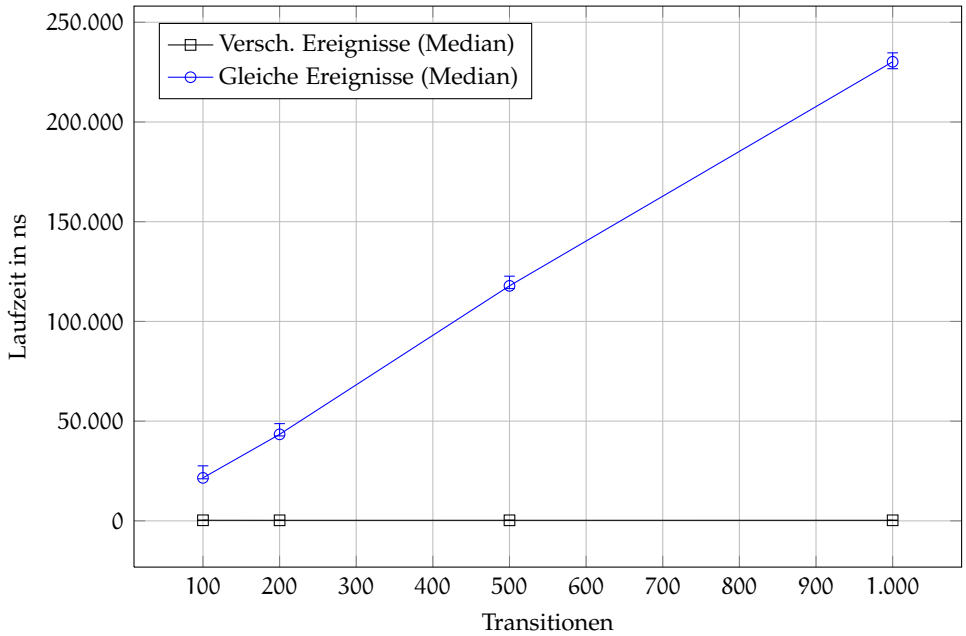


Abbildung 9.3: Laufzeiten der Java-Monitore

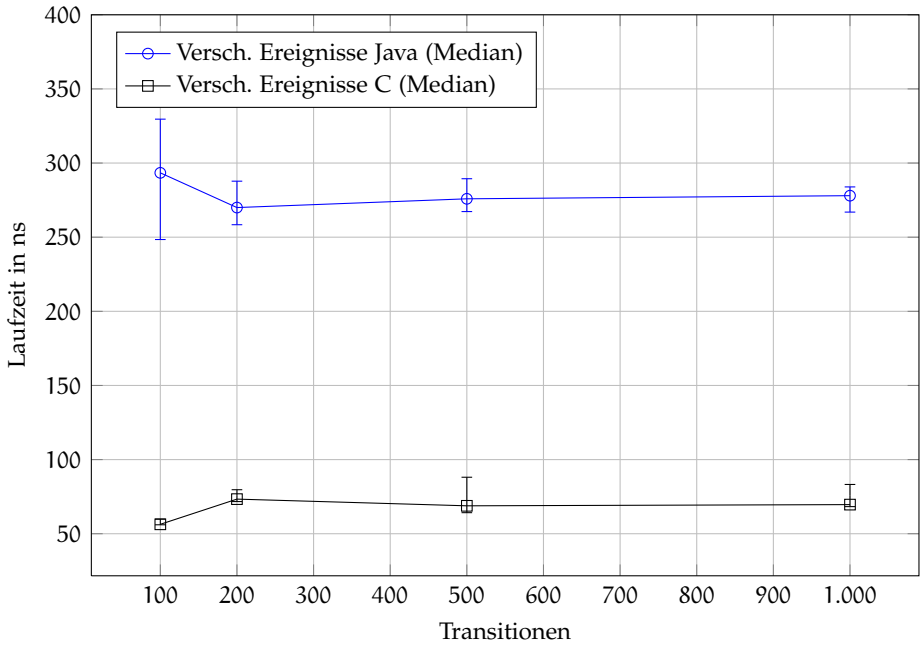


Abbildung 9.4: Laufzeiten der Monitore bei verschiedenen Ereignissen

mit identischen Ereignissen eine höhere Laufzeit. In der Implementierung werden viele Objekte und Datenstrukturen aufgebaut, die durch den Garbage-Collector aus dem Heap entfernt werden. Besonders die Verwaltung der belegten Plätze in HashSets, führt zu längeren Laufzeiten bei der Überprüfung, ob eine Transition aktiv ist. An dieser Stelle ist eine Umstellung des Speicherns der belegten Plätze auf das Konzept der C-Codegenerierung sinnvoll.

Zum direkten Vergleich der beiden Implementierungen sind in Abbildung 9.4 und 9.5 die Laufzeit der Monitore für denselben Signaturtyp dargestellt. Hierbei ist zu sehen, dass die Laufzeiten der Implementierungen bei verschiedenen Ereignistypen an jeder Transition konstant bleiben. Die Java-Implementierung ist hierbei deutlich langsamer als die C-Implementierung. Dies ist auf die Optimierung des C-Codes im Gegensatz zur Java-Implementierung und die Ausführung des Java-Codes in der Java Virtuellen Maschine zurückzuführen.

SCHLUSSFOLGERUNG Ein realistischer Monitor würde nur wenige Transitionen, die mit demselben Ereignis annotiert sind, enthalten. Quellen für solche Transitionen sind zum einen Epsilontransitionen und zum anderen Transitionen, die aus optionalen Elementen in der Quellspezifikation (z. B. eLSC) stammen. Aus diesem Grund liegt die tatsächliche Laufzeit eines realen Monitors im unteren Bereich zwischen diesen beiden Kurven. Des Weiteren sind eher kleinere Signaturen mit einer Größe zwischen 20 und 100 Transitionen realistisch. Wenn weitere Signaturen dem Monitor hinzugefügt werden, wächst die Laufzeit des Monitors linear zu der Laufzeit der einzelnen Signaturen.

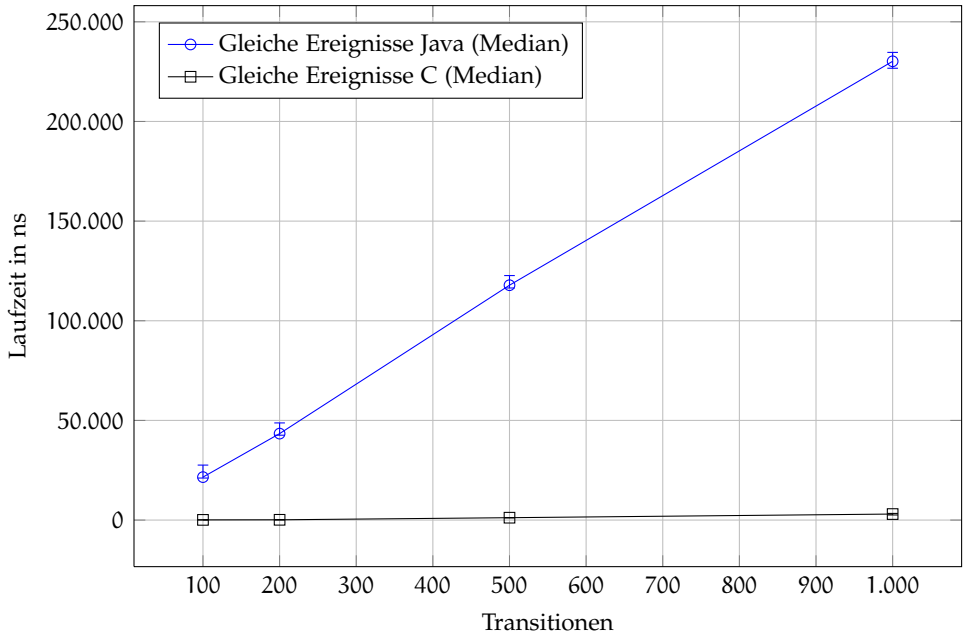


Abbildung 9.5: Laufzeiten der Monitore bei identischen Ereignissen

9.1.4 *Messungen auf einem Mikrocontroller*

Die Messungen auf dem PC im vorherigen Abschnitt haben die Unterschiede zwischen den Implementierungen verdeutlicht. Jedoch konnte bei kleinen Signaturen beobachtet werden, dass kleine Monitore im Verhältnis schneller ausgeführt wurden, als große Monitore. Des Weiteren gab es bei längeren Messungen relativ starke Streuungen. Um diesen Effekt genauer zu betrachten, werden im fraglichen Bereich kleinere Signaturen, die dem Muster aus Abschnitt 9.1.2 entsprechen, zwischen einer Nachricht (zwei Transitionen) und 100 Nachrichten (200 Transitionen) auf einem Mikrocontroller untersucht. Für den Einsatz eines Monitors auf einem eingebetteten System stellt der benötigte Speicher einen kritischen Aspekt dar. Hierbei ist besonders der Datenspeicher von Bedeutung, der in einem solchen System in der Regel sehr klein ist.

Die Messung wird auf einem *Fujitsu SK-16FX-100PMC*-Evaluationsboard (24 kB RAM, 544 kB Flash), das mit einem *F²MC-16FX MB96340 series*-Mikrocontroller (16 bit, 56 MHz) bestückt ist, ausgeführt. Der C-Code wird durch den mit dem Board ausgelieferten Fujitsu Softtune C-Compiler in Maschinencode übersetzt.

Es wird die Laufzeit, die für 1000 Durchläufe durch den Monitor benötigt wird, gemessen und die Laufzeit auf ein verarbeitetes Ereignis heruntergerechnet. Die Ausführungszeit auf einem Mikrocontroller ohne Betriebssystem ist deterministisch, weshalb eine Messung pro Signatur ausreicht. Zusätzlich zur Laufzeit wird hier der Speicherbedarf der Monitore gemessen, der insbesondere auf eingebetteten Systemen eine große Rolle spielt. Die Werte des benötigten Datenspeichers (RAM) enthalten hierbei 800 Bytes, die nicht vom Monitor selbst stammen.

AUSWERTUNG Abbildung 9.6 zeigt die Laufzeit der Monitore nach wachsender Größe der Spezifikation. Bei sehr kleinen Modellen spielt die Initialisierung

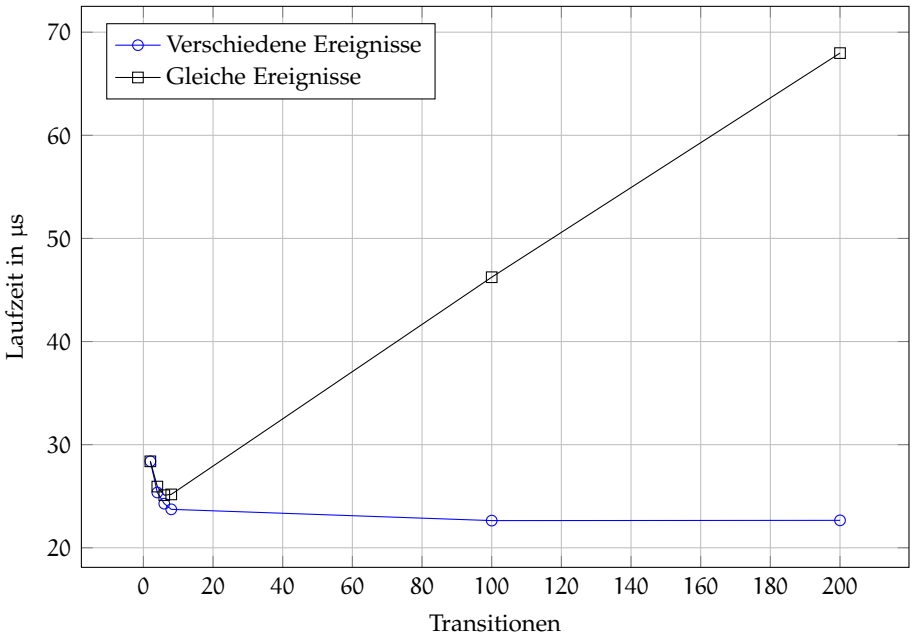


Abbildung 9.6: Laufzeiten der Monitore auf einem Mikrocontroller

der Monitore und der Anteil des Controllers an der Laufzeit noch eine große Rolle, weshalb zunächst bei steigender Signaturgröße eine Laufzeitabnahme für die Verarbeitung eines Ereignisses bei kleinen Signaturen zu sehen ist. Auch hier ist ein linearer Anstieg bei identischen Ereignissen an den Transitionen der Signatur zu erkennen. Bei unterschiedlichen Ereignissen ist die Laufzeit mit wachsender Größe der Signatur konstant.

Abbildung 9.7 stellt den wachsenden Codespeicherverbrauch der C-Implementierung der generierten Monitore dar. Dieser wächst linear zu der Größe der MPNs. Für jeden Ereignistyp wird eine Funktion generiert, die alle Transitionen mit diesem Ereignis zusammenfasst. Hierbei ist insbesondere die Anzahl der Transitionen mit verschiedenen Ereignissen im MPN der treibende Faktor, der den Unterschied in der Steigung der Kurve verursacht.

Abbildung 9.8 zeigt den Datenspeicherverbrauch (RAM) des Monitors bei wachsender Signaturgröße. Hierbei unterscheiden sich die beiden betrachteten Signaturtypen nur unwesentlich. Dies ist darauf zurückzuführen, dass der Hauptteil des Datenspeichers für die Speicherung der Belegung der Plätze im MPN aufgewendet wird. Der Speicherverbrauch steigt mit wachsender Signaturgröße sehr langsam an, da eine Codierung der Platzbelegung in einen Integer-Array stattfindet, in dem die Platzbelegung bitweise gespeichert wird. In einem MPN kann ein Platz nur belegt sein oder nicht belegt sein, wodurch im Idealfall nur ein Bit pro Platz und Generation im MPN zur Speicherung des Zustandes benötigt wird.

SCHLUSSFOLGERUNG Es lässt sich feststellen, dass Monitore, die aus Monitor-Petrinetzen generiert wurden, auch auf eingebetteten Systemen einsetzbar sind. Hierfür sprechen insbesondere der niedrige Bedarf an Datenspeicher und der lineare Anstieg des benötigten Codespeichers. Auch im niedrigen Bereich (bei kleinen Signaturen) wächst die Laufzeit höchstens linear zur Größe der Signatur bei

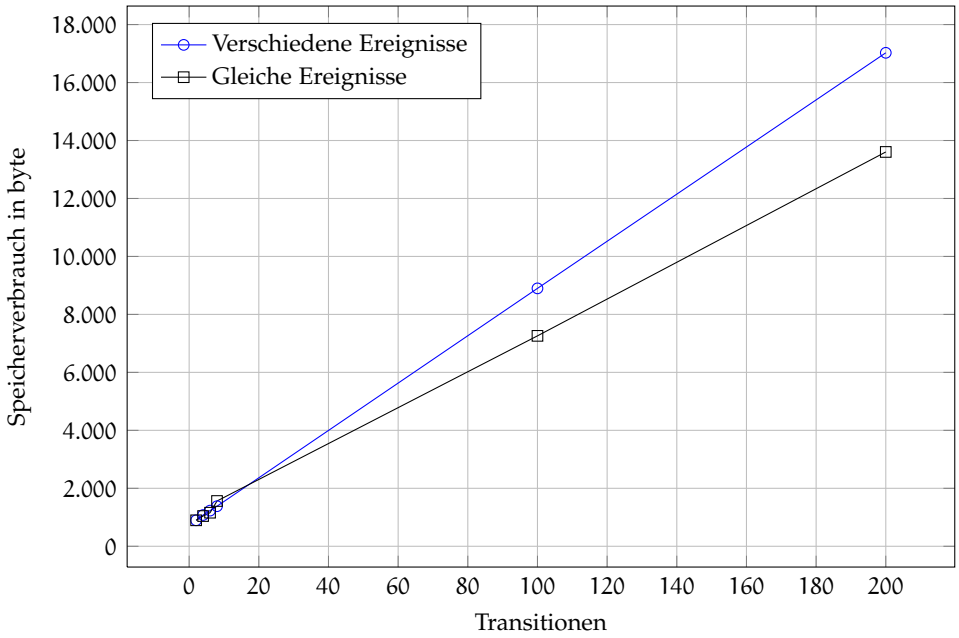


Abbildung 9.7: Codespeicherverbrauch der Monitore auf einem Mikrocontroller

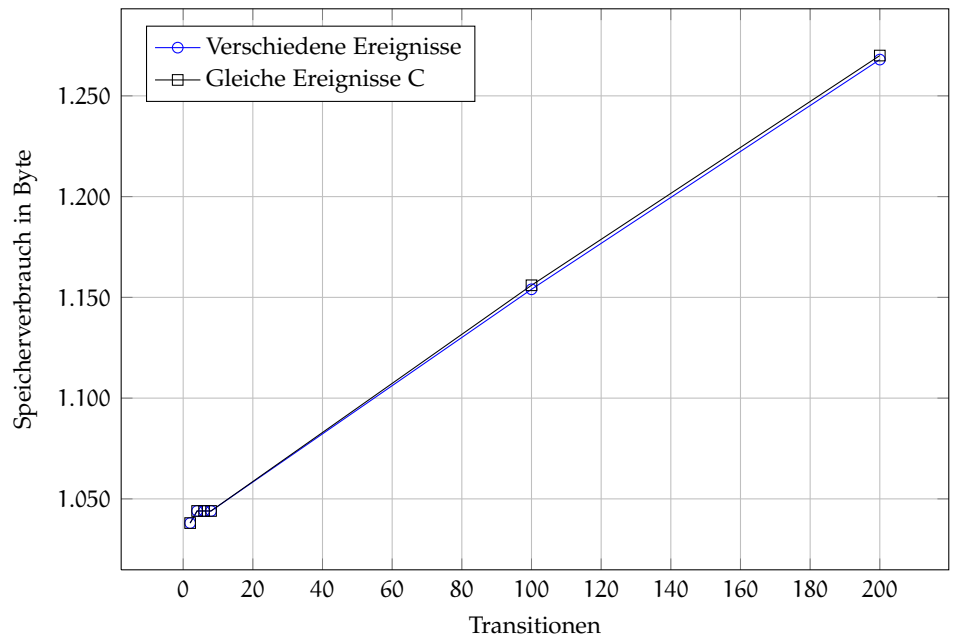


Abbildung 9.8: Datenspeicherverbrauch der Monitore auf einem Mikrocontroller

identischen Ereignissen an den Transitionen. Bei unterschiedlichen Ereignissen bleibt die Laufzeit pro Ereignis konstant.

In diesem Teil der Evaluation wurden nur die MPNs und ihre Implementierung an sich betrachtet. Diese Messungen wurden ohne Einsatz zusätzlicher Bedingungen und Aktionen in den Signaturen durchgeführt, die entsprechend ihrer Komplexität in die Laufzeit des Monitors einberechnet werden müssen.

9.2 Vergleich von C auf einem Mikrocontroller und VHDL auf einem FPGA

Die Abbildung von Monitor-Petrinetzen auf die Hardwarebeschreibungssprache VHDL basiert grundlegend auf [SP05], die die Übersetzung von Petrinetzen zu VHDL-Code beschreibt. Dieser Ansatz benötigt zwei Taktzyklen für das Feuern einer Transition im Petrinetz und kombiniert je einen Platz mit einer Transition in einem Modul.

Der Ansatz wurde in [Ill12] für die Generierung von Monitor-Petrinetzen soweit angepasst und optimiert, sodass nur noch ein Takt für das Schalten einer Transition benötigt wird. Des Weiteren wurde die Repräsentation von Platz und Transition auf einzelne Blöcke aufgeteilt, um einen optimierten Verbrauch an Slices zu erreichen.

In gewöhnlichen Petrinetzen kommt es im Gegensatz zu den MPNs zu Konflikten. Für die Abbildung des Makroschrittkonzepts der MPNs, in denen mehrere Transitionen in einem Schritt schalten können, wurde ein *Execute*-Signal eingeführt, das das Schalten der Transitionen nach dem Setzen der Bedingungen und des Ereignisses synchron auslöst. Hierdurch kommt es nicht schon durch das Setzen eines der Bedingungseingänge zu einem Schalten einzelner Transitionen, die Token von Plätzen entfernen, die von einer anderen Transition im selben Makroschritt benötigt werden.

Die Repräsentation der Generationen kann durch Vervielfachung des Netzes auf dem FPGA oder durch einen Speicher, in dem der Zustand jeder Generation gespeichert wird, umgesetzt werden. Durch Lese- und Schreibzugriffe auf einen Speicher würde der Vorteil des FPGAs, die hohe Verarbeitungsgeschwindigkeit, jedoch stark reduziert werden. Deshalb wird bei dieser Implementierung für jede Generation ein eigene Kopie des MPNs auf dem FPGA generiert.

9.2.1 *Beispielsystem Ampelanlage*

Abbildung 9.9 zeigt das für den Vergleich zwischen Mikrocontroller- und FPGA-Implementierung herangezogene Szenario. Hierbei handelt es sich um eine Ampelanlage, die an einer Kreuzung mit Haupt- und Nebenstraße eingesetzt wird. Diese Ampelanlage wird, wie auf der rechten Seite von Abbildung 9.9 dargestellt ist, von zwei Mikrocontrollern gesteuert. Dabei übernimmt der eine Mikrocontroller die Hauptstraße und der andere die Steuerung der Ampel an der Nebenstraße. Die Kommunikation zwischen diesen beiden Steuerungseinheiten findet über einen CAN-Bus statt, über den sie sich synchronisieren.

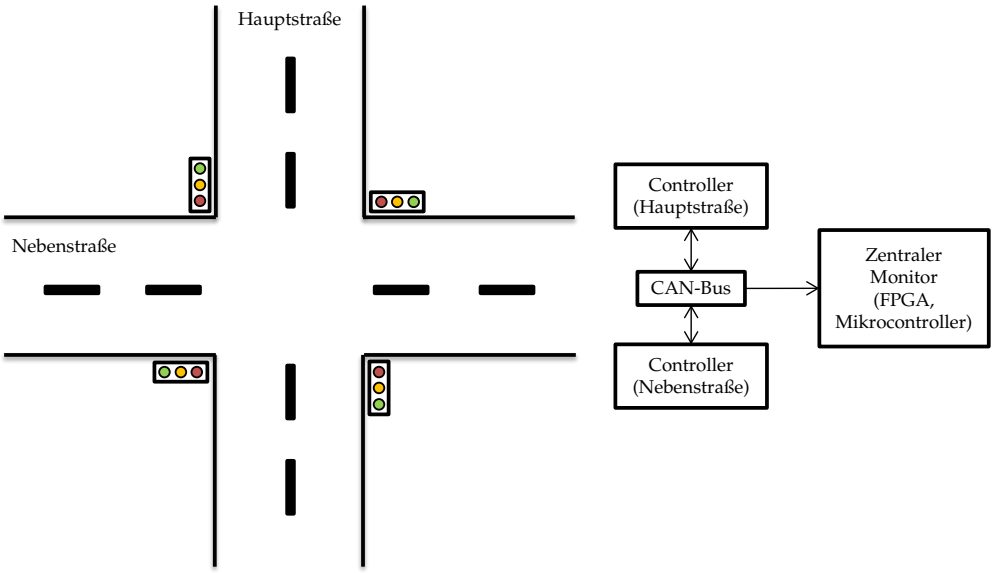


Abbildung 9.9: Ampelszenario mit CAN-Bus

Element	Anzahl
Plätze	22
Transitionen	17
Ereignisse	12
Bedingungen	0
Generationen	1

Tabelle 9.1: Eigenschaften des TrafficLight-MPNs

Diese Kommunikation über den CAN-Bus soll im Folgenden durch einen Monitor auf Korrektheit überwacht werden. Hierzu wird im MBSecMon-Prozess ein Monitor in der MBSecMon-Spezifikationssprache (eLSCs) spezifiziert. Diese Spezifikation wird dann verwendet, um C-Code bzw. eine VHDL-Beschreibung zu generieren.

Die Signatur, die den korrekten Ablauf der Kommunikation der Ampel beschreibt, ist in Abbildung 9.10 dargestellt. Die Kommunikation zwischen der Ampel auf der Hauptstraße (*Mainroad*) und der Ampel auf der Nebenstraße (*Sideroad*) läuft asynchron ab. Zunächst wird eine *SwitchToGeen*-Nachricht an die Nebenstraße geschickt. Daraufhin kann je nach Konfiguration der Ampeln die Nebenstraßenampel eine Nachricht (*AmberRed*) verschicken, dass die Ampel auf Gelb-Rot geschaltet wurde. Verpflichtend muss danach die Nachricht *Acknowledgement-Green* von der Nebenstraßenampel aus gesendet werden, die der Hauptstraßenampel signalisiert, dass die Ampel auf Grün geschaltet wurde. Derselbe Ablauf folgt darauf für die Schaltung der Nebenstraßenampel auf Rot.

In der MPN-Repräsentation besitzt diese Signatur die in Tabelle 9.1 angegebenen Eigenschaften. In diesem Szenario wird nur eine Generation verwendet, da nur die Kommunikation zwischen zwei Steuergeräten (ein Kommunikationsstrang) überwacht werden soll.

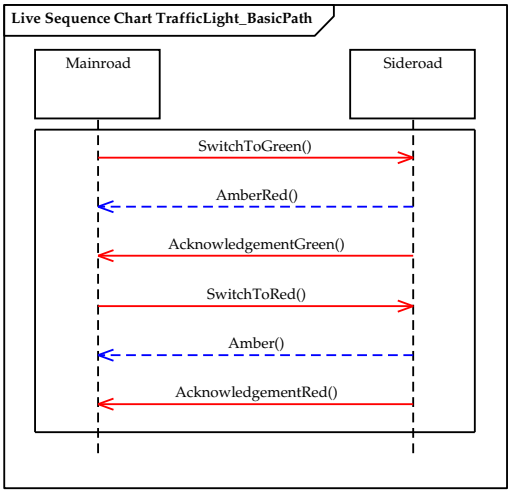


Abbildung 9.10: Signatur des korrekten Ablaufs der Kommunikation zwischen den Ampeln

9.2.2 *Ziel*

In der MPN-Sprache können mehrere Transitionen in einem Makroschritt schalten. Somit müssen alle für das gerade zu verarbeitende Ereignis infrage kommenden Transitionen ausgewertet werden. Ziel dieser Evaluation ist die Messung des Laufzeitunterschieds durch die Parallelisierung der Auswertung von Ereignissen auf einem FPGA gegenüber eines Mikrocontrollers mit ähnlicher Taktrate. Die Messungen werden an einem Ampelszenario zur Ermittlung der Laufzeiten an einer komplexen Signatur und der Ermittlung des Platzverbrauchs auf einem FPGA bei steigender Generationszahl durchgeführt. Zur Bestimmung des Verhaltens der Implementierung auf einem FPGA in Bezug auf den Platzbedarf und Laufzeit bei steigender Signaturgröße wird das konstruierte Beispiel aus Abschnitt 9.1.2 mit einer steigenden Anzahl von Transitionen verwendet.

9.2.3 *Messungen*

Die Frequenz des Mikrocontrollers ist 56 MHz und die des FPGAs 50 MHz. Zur Messung der Verarbeitungszeit wurde ein Logic Analyzer[4], der eine Abtastrate von 100 MHz unterstützt verwendet und die Zeit von der Übergabe des Ereignisses an den Monitor bis zum Ende der Abarbeitung gemessen. Diese Abtastrate entspricht einer Auflösung von 10 ns.

Tabelle 9.2 stellt die benötigte Laufzeit des Monitors auf dem FPGA und dem Mikrocontroller gegenüber. Betrachtet wird hierbei die Laufzeit, die der Monitor benötigt, um ein Ereignis zu verarbeiten. Alle Ereignisse wurden getrennt voneinander gemessen, um einen Vergleich der Laufzeiten für verschiedene Ereignisse zu ermöglichen.

[4] Saleae Logic16: http://www.saleae.com/logic16

Ereignis	Bearbeitungszeit	
	Mikrocontroller (µs)	FPGA (µs)
send.switchToGreen	25,39	0,06
recv.switchToGreen	22,82	0,06
send.isAmberRed	23,41	0,06
recv.isAmberRed	22,81	0,06
send.acknowledgmentGreen	23,03	0,06
recv.acknowledgmentGreen	24,38	0,06
send.switchToRed	22,57	0,06
recv.switchToRed	22,84	0,06
send.isAmber	23,48	0,06
recv.isAmber	22,81	0,06
send.acknowledgmentRed	24,05	0,06
recv.acknowledgmentRed	36,53	0,08
Durchschnitt	24,51	0,06

Tabelle 9.2: Vergleich der Ereignisverarbeitungszeiten

Generationen	Controller	
N_{gen}	Slices	Anteil (%)
1	63	1,35
2	120	2,58
3	173	3,72
4	215	4,62
5	268	5,76
10	514	11,04
20	1005	21,59
30	1500	32,22
40	1988	42,70
50	2481	53,29
60	2965	63,68

Tabelle 9.3: Messung der Slicenutzung der Module bei steigender Generationszahl

Die durchschnittliche Laufzeit beträgt für das Ampelszenario bei Verwendung des Mikrocontrollers 24,51 µs und bei Einsatz eines FPGAs konstante 60 ns. Die Schwankungen bei verschiedenen Ereignissen auf dem Mikrocontroller sind auf die Ausführung unterschiedlich umfangreicher Berechnungen zurückzuführen. Hierbei spielen die Anzahl der Plätze im Vor- und Nachbereich sowie die Anzahl der Transitionen mit demselben Ereignis eine Rolle. Im Gegensatz dazu werden auf dem FPGA alle Transitionen parallel ausgewertet und somit ist nur ein Schritt für die Ereignisverarbeitung notwendig. Der Anstieg um einen Takt beim FPGA auf 80 ns beim letzten Ereignis ist auf die nachfolgende Epsilontransition für die Endsynchronisation zurückzuführen. Diese kann nicht parallel ausgeführt werden und muss somit sequenziell zum vorherigen Ereignis verarbeitet werden. Die Ausführung ist in diesem Beispiel auf dem FPGA ungefähr 400-mal schneller.

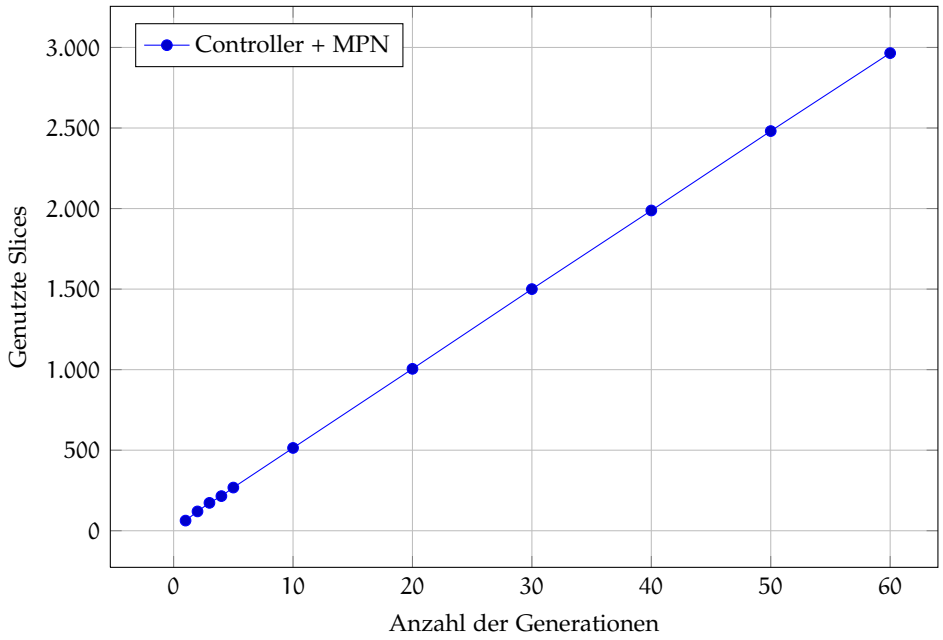


Abbildung 9.11: Benötigte Slices auf dem FPGA (nicht optimiert) im Verhältnis zur Anzahl der zu verwaltenden Generationen

Abbildung 9.11 stellt den in Tabelle 9.3 aufgelisteten steigenden Verbrauch der Slices mit steigender Generationsanzahl dar. Slices sind logische Blöcke eines FPGAs, die in diesem Fall aus zwei Flip-Flops, zwei Lookup-Tabellen (LUT) und zur Konfiguration benötigte Schaltungslogik bestehen. Hierbei ist zu beachten, dass die Werte den nicht-optimierten Stand des Sliceverbrauchs vor dem Mapping-Schritt darstellen, in dem auf dem Gitter des FPGAs die logischen Elemente platziert und verbunden werden. Optimierungen, die durch das Zusammenlegen von Slices auf dem FPGA entstehen, sind nicht beachtet. Durch diese Optimierungen ist eine deutliche Reduktion der tatsächlich benötigten Slices auf dem FPGA möglich. Da die Übersetzung der Generationen einer Vervielfachung der Netzstruktur auf dem FPGA entspricht, ist zu sehen, dass die Anzahl der benötigten Slices linear mit der Anzahl der Generationen ansteigt.

9.2.4 *VHDL-Evaluation an einem konstruierten Beispiel*

Im vorherigen Abschnitt wurden die Laufzeit und der Slice-Verbrauch auf einem FPGA bei zunehmender Anzahl von Generationen betrachtet. Hier werden nun das Laufzeitverhalten und der Verbrauch an Slices bei wachsender Größe der Signaturen untersucht.

In dieser Betrachtung wird das konstruierte Beispiel aus Abschnitt 9.1 verwendet. Abbildung 9.12 stellt die Laufzeiten der Monitore auf dem FPGA dar. Das FPGA läuft mit 50 MHz, wobei ein Takt somit eine Dauer von 20 ns hat. Hierbei ist die Laufzeit bei verschiedenen und bei identischen Ereignissen in der Signatur konstant bei 60 ns. In diesem Diagramm wurde das letzte Ereignis im Monitor, das eine abschließende Epsilontransition nach sich zieht vernachlässigt. Jede Ep-

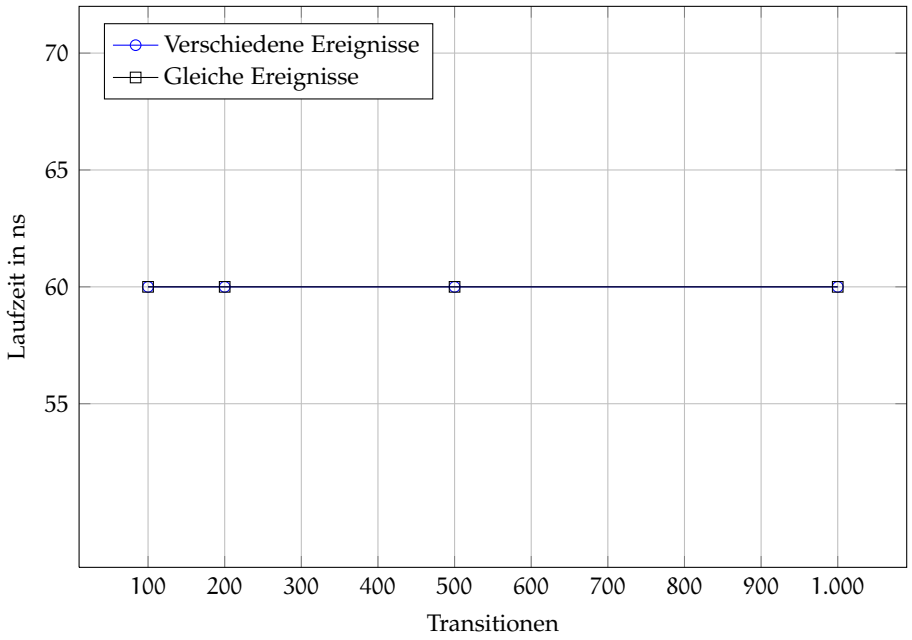


Abbildung 9.12: Laufzeiten der Monitore auf einem FPGA

silontransition, die auf eine Transition mit einem annotierten Ereignis folgt und schaltet, fügt jeweils 20 ns zur Verarbeitungszeit hinzu.

Abbildung 9.13 stellt den Bedarf an Slices auf dem FPGA für eine wachsende Anzahl von Transitionen im Monitor dar. Hierbei zeigt sich ein ähnliches Bild wie bei dem Speicherbedarf der C-Monitore auf einem Mikrocontroller in Abschnitt 9.1.4. Für Monitore mit verschiedenen Ereignissen an den Transitionen des MPNs wird mehr Speicher benötigt als für die Monitore mit identischen Ereignissen. Hierbei ist der Anstieg der benötigten Slices linear zur Größe der Monitorspezifikation.

9.2.5 *Schlussfolgerung*

Signaturen, die als Monitor-Petrinetze vorliegen, können auf ein FPGA bzw. die Hardwarebeschreibungssprache VHDL abgebildet werden. Dabei hat sich gezeigt, dass sich die Semantik der MPNs mit ihren Makroschritten gut auf eine parallele Schaltung auf dem FPGA übertragen lässt. Hierdurch sind konstante und sehr kurze Laufzeiten bei der Ereignisverarbeitung des Monitors zu erreichen. Selbst bei einem Wachstum der Signatur mit Transitionen mit identischen Ereignissen findet im Gegensatz zu den Implementierungen in C und Java kein Anstieg der Laufzeit statt.

Einschränkungen der prototypischen VHDL-Implementierung sind:

- Keine Subgenerationen
- Keine komplexen Variablen
- Bedingungen als Signale

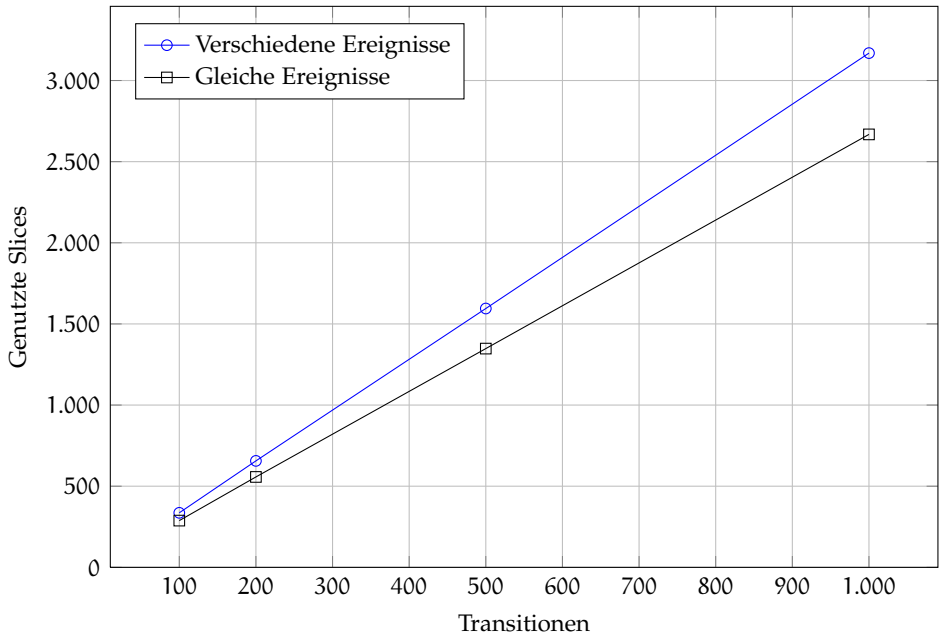


Abbildung 9.13: Benötigte Slices der Monitore auf einem FPGA

In der prototypischen Implementierung des Generierungsprozesses für ein FPGA wurde eine reduzierte Version der Monitor-Petrinetze mit Generationen zur Überwachung von verschiedenen Kommunikationssträngen und einfachen booleschen Signalen als Bedingungen umgesetzt. Jedoch ist auch eine Unterstützung annotierter Bedingungen, die sich in logische Schaltungen in der Zielsprache übersetzen lassen, ebenfalls umsetzbar. Dies würde zu keiner Verlängerung der Laufzeit führen, da es sich hierbei nur um eine logische Schaltung handelt. Erst sehr komplexe Schaltungen mit langen Laufzeiten würden dazu führen, dass das FPGA auf einem niedrigeren Takt betrieben werden müsste, um das Timing sicherzustellen.

9.3 Aufteilung der Monitore in C

In der Bachelorarbeit [Keß12] wurde eine Version der Codegenerierung für C entwickelt, die es unterstützt, Teilmonitore, wie sie in Abschnitt 6.6 eingeführt wurde, zu generieren. Eine solche Erweiterung führt dazu, dass neue Identifizierer eingeführt werden müssen, um die zu synchronisierenden Plätze durch Kommunikationsmethoden zu verknüpfen. Zusätzlich wurden die Monitore um eine Warteschlange (*engl. Queue*) erweitert, um den Monitor thread-sicher auf einem Echtzeitbetriebssystem betreiben zu können. Die Auswirkung dieser Erweiterungen wird in diesem Abschnitt genauer betrachtet. Hierzu wird das folgende Mautbrückenszenario genutzt.

9.3.1 *Beispielsystem Mautbrücke*

In diesem Vergleich wird das CARDME-Mautbrückenszenario aus dieser Arbeit in vereinfachter Form genutzt. Bisher wurde in der Evaluation immer von ei-

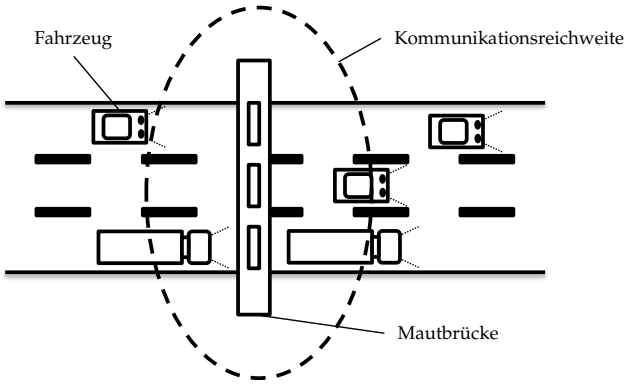


Abbildung 9.14: Mautbrückenszenario zur Evaluation eines verteilten Monitors

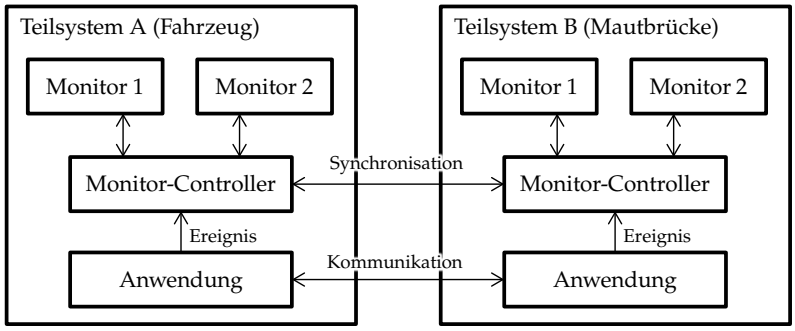


Abbildung 9.15: Prinzip der Verteilung ganzheitlich spezifizierter Monitore

nem zentralen Monitor ausgegangen, der alle für die Überwachung benötigten Ereignisse zur Verfügung gestellt bekommt. Abbildung 9.14 zeigt das verwendete Szenario. Fahrzeuge fahren auf eine Mautbrücke zu und kommunizieren mit dieser, wenn sie in die Kommunikationsreichweite der Mautbrücke kommen, wie es in Abschnitt 1.5 vorgestellt wurde.

Die Monitore werden ganzheitlich für die Kommunikation zwischen Fahrzeug und Mautbrücke in der MBSecMonSL spezifiziert und wie in Abschnitt 6.6 beschrieben auf Fahrzeug und Mautbrücke aufgeteilt. Abbildung 9.15 stellt diese Aufteilung abstrakt dar. Hierbei ergeben die Teilmonitore für *Monitor 1* bzw. *Monitor 2* auf den verschiedenen Teilsystemen jeweils den Gesamtmonitor der ursprünglichen Signatur. Der *Monitor-Controller* übernimmt hierbei die notwendige Synchronisation der Teilmonitore.

Die Mautbrücke und das Fahrzeug werden jeweils durch ein Evaluationsboard mit Mikrocontroller repräsentiert, auf dem C-Code läuft. Diese beiden Boards kommunizieren hierbei über den CAN-Bus. Der Monitor verwendet zur Synchronisation der Teilmonitore denselben Bus.

9.3.2 *Ziel*

Durch die Einführung des Aufteilens von Signaturen wird durch die Synchronisation ein Overhead in Laufzeit- und Speicherbedarf im Gegensatz zum zentralen

Modell	Statische Daten	Code
CARDME_Local	138	4867
CARDME_Local (TB)	186	3437
CARDME_Local (Vehicle)	188	4033

Tabelle 9.4: Speicherverbrauch des Monitorcodes in Byte

Monitor eingeführt. Andererseits wird die Größe der Signaturen auf den einzelnen Steuergeräten reduziert. In dieser Evaluation ist es das Ziel festzustellen, inwieweit sich diese beiden Aspekte beeinflussen und wie die Kommunikation zum Laufzeitoverhead beiträgt.

9.3.3 *Messungen*

Die folgenden Messungen werden auf einem Mikrocontroller mit dem Echtzeitbetriebssystem (RTOS) FreeRTOS[5] durchgeführt. Es wird wiederum ein Fujitsu SK-16FX-100PMC-Evaluationsboard (24 kB RAM, 544 kB Flash), das mit einem F^{2}MC-16FX MB96340 series Mikrocontroller (16 bit, 56 MHz) bestückt ist, verwendet.

Die Synchronisierung der Platzbelegungen zwischen Teilmonitoren sowie die Kommunikation des zu überwachenden Systems findet über einen CAN-Bus, der mit einer Taktrate von 50 kBit/s betrieben wird, statt. Die Laufzeit der Synchronisierungsnachrichten, die als Daten, die zu synchronisierende Platz-ID (16 Bit), die Generation (8 Bit) und den Wert, ob ein Token aus dem Platz entfernt oder auf den Platz gelegt werden soll (8 Bit), übermittelt. Eine solche Nachricht benötigt in dem hier beschriebenen Fall ca. 2 ms. Hier ist zu sehen, dass diese Zeit gegenüber der Verarbeitung eines Ereignisses (ca. 40 µs) im Monitor relativ hoch ist und stark vom verwendeten Kommunikationskanal abhängt.

9.3.4 *Auswertung*

Tabelle 9.4 zeigt den benötigten Speicher für das CARDME-Szenario. Hierbei wird von den Werten jeweils der vom Basissystem ohne Monitor benötigte Speicher abgezogen, um den Einfluss des Monitors zu untersuchen. Aufgeteilt wird die *CARDME_Local*-Signatur aus Abschnitt 6.1.

Es zeigt sich, dass durch die Aufteilung des Monitors ein erhöhter Bedarf an statischem Datenspeicher auf beiden Seiten entsteht. Dieser ist durch die Kommunikationsschnittstellen, die zur Synchronisierung der Plätze und Daten hinzukommen, zu erklären. Die Reduktion der Plätze hat hierauf einen untergeordneten Einfluss, da diese sehr speichereffizient verwaltet werden. Gleichzeitig reduziert sich im Gegensatz zur vollständigen Signatur der benötigte Codespeicher für die einzelnen Systeme.

Der Monitorcode für verteilbare Systeme wurde im Gegensatz zu den Monitoren in Abschnitt 9.1 neu strukturiert, um die Aufteilbarkeit der Monitore zu

[5] FreeRTOS: http://www.freertos.org/

eLSC	MPN	
Nachrichten	Plätze	Ereignisse/Transitionen
10	34	21
50	154	101
100	304	201

Tabelle 9.5: Eigenschaften der MPNs

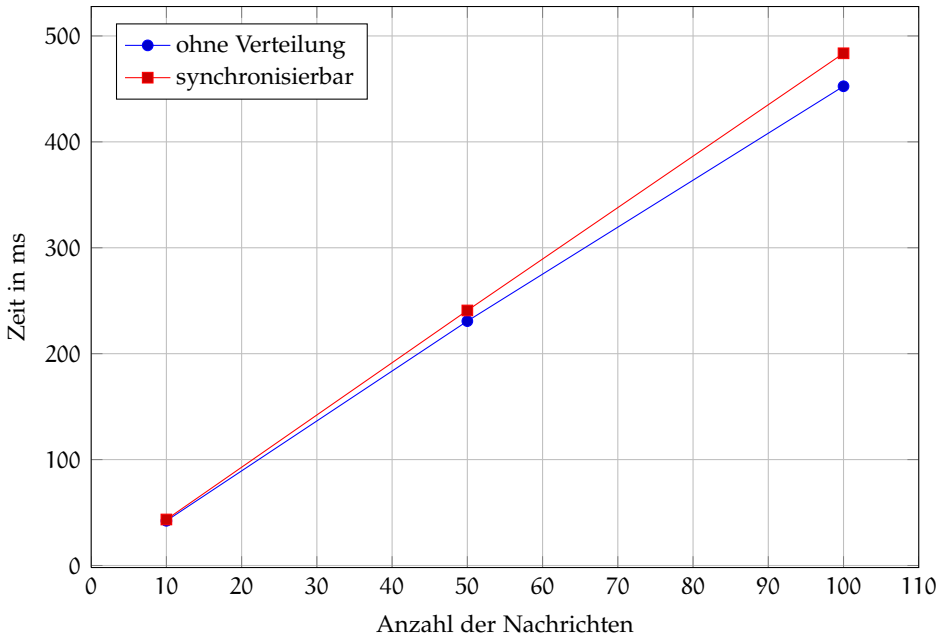


Abbildung 9.16: Laufzeit für 100 Iterationen des verteilten und nicht verteilten Monitors bei wachsender Spezifikation

erleichtern. Des Weiteren wurden Schnittstellen zur Kommunikation der Teilmonitore untereinander eingeführt. Im Folgenden wird die Auswirkung auf die Laufzeit und den Speicherbedarf der Monitore durch diese Neustrukturierung betrachtet.

Zur weiteren Messung wird wieder ein konstruiertes Beispiel verwendet, bei dem zwei Komponenten miteinander asynchron kommunizieren. Tabelle 9.5 zeigt die Eigenschaften dieser MPNs. Jede der Nachrichten hat einen unterschiedlichen Nachrichtentyp, wodurch für jede Nachricht zwei Ereignistypen entstehen. Zusätzlich existiert noch ein Ereignistyp für Epsilontransitionen.

Mithilfe dieser Modelle wird mit dem bisherigen nicht verteilten Ansatz Monitorcode generiert (ohne Verteilung) und mit dem neuen Ansatz (synchronisierbar) Code generiert. Abbildung 9.16 stellt die Laufzeit des Monitors für jeweils 100 Iterationen über die gesamte Spezifikation dar. Hierbei ist zu sehen, dass diese linear anwächst und bei dem verteilten Monitor ohne Synchronisation nur leicht erhöht ist. Diese höhere Laufzeit resultiert unter anderem durch häufigere Funktionsaufrufe während der Auswertung der Ereignisse.

Zur Messung des Speicherbedarfs der beiden Versionen der Codegenerierung zueinander wird ebenfalls das konstruierte Beispiel eingesetzt. Abbildung 9.17

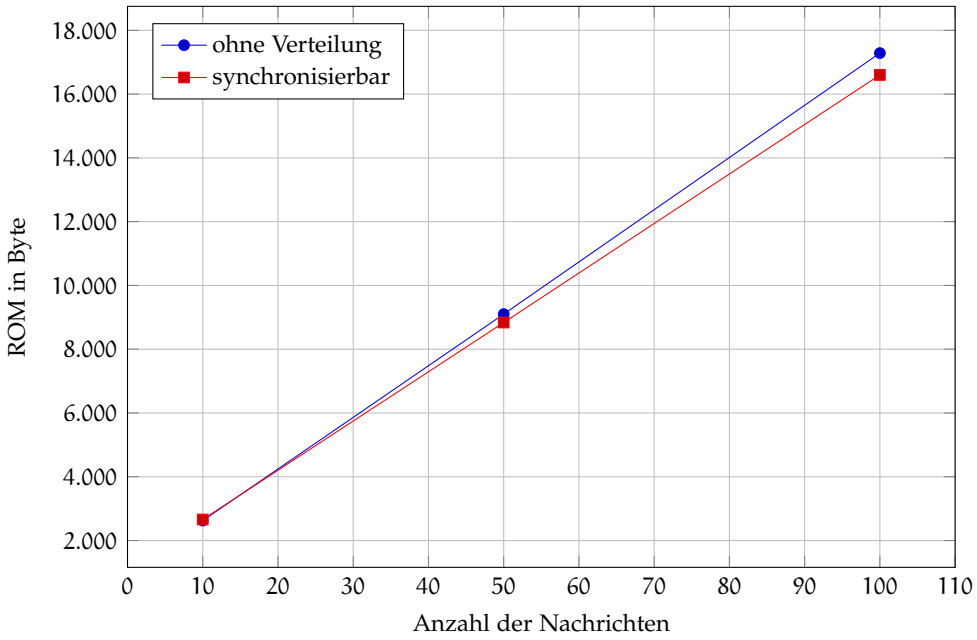


Abbildung 9.17: Codespeicherverbrauch (ROM) des Monitors für nicht verteilte und verteilte Monitore bei wachsender Spezifikation

zeigt den Codespeicher, der von den Monitoren benötigt wird. Die für die Synchronisierung vorbereitete Version benötigt durch die Umstrukturierung etwas weniger Codespeicher.

Abbildung 9.18 zeigt den benötigten Datenspeicher der Implementierungen. Hier ist deutlich zu sehen, dass durch die Verteilbarkeit des Monitors, die zusätzliche Identifizierer für die Zuordnung der Plätze einführt, nach einem niedrigeren Speicherbedarf bei kleinen Monitoren ein deutlicher Anstieg bei größeren Signaturen gegenüber dem nicht verteilbaren Monitor entsteht. Der erhöhte Datenspeicherbedarf bei kleinen Signaturen (10 Nachrichten) des nicht verteilbaren Monitors resultiert aus dem höheren Speicherbedarf des Controllers im Gegensatz zu der auf Verteilbarkeit optimierten Implementierung.

9.3.5 *Schlussfolgerung*

Durch die Aufteilung der Monitore wird ein Overhead dem System hinzugefügt. Dieser ist jedoch relativ gering und erlaubt die Überwachung komplexerer Signaturen, die Informationen des Gesamtsystems verwenden, auf verteilten Systemen. Hierbei muss jedoch auch ein Kommunikationskanal geschaffen werden, der in die Laufzeit des Monitors mit einberechnet werden muss.

Wenn die Laufzeit des Monitors zur Verarbeitung eines Ereignisses eine kritische Größe darstellt, sollten für einzelne Komponenten eigenständige Monitore generiert werden, die nicht oder nur sehr selten miteinander kommunizieren. Zur Reduktion der Verarbeitungszeit kann hierbei ein eigenständiger schneller Kommunikationskanal für die Monitore eingeführt werden. Ein solcher zusätzlicher Kanal wird in der Praxis jedoch aus Kostengründen nur in seltenen Fällen eingesetzt werden.

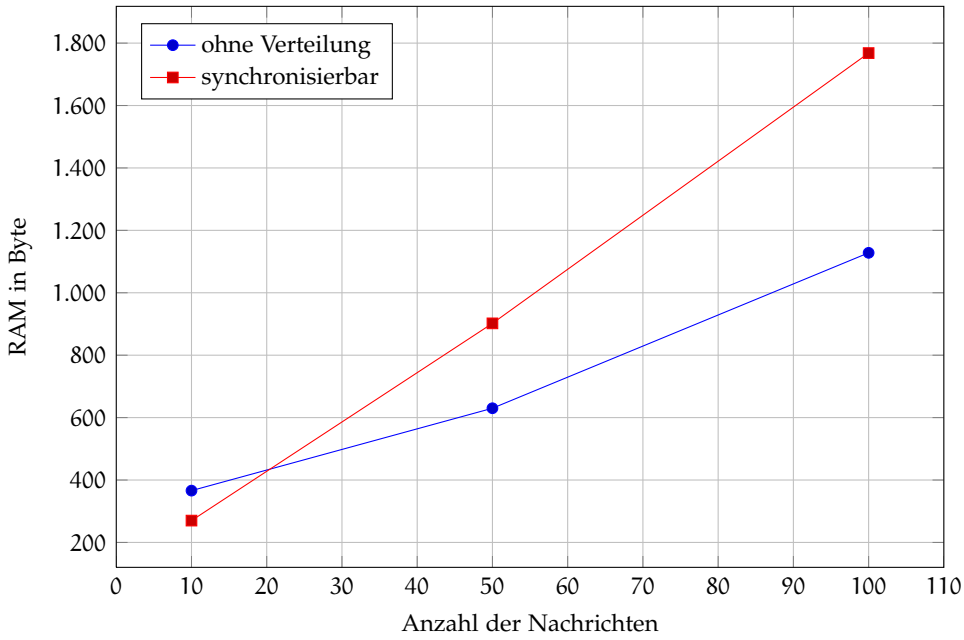


Abbildung 9.18: Datenspeicherverbrauch (RAM) des Monitors für nicht verteilte und verteilte Monitore bei wachsender Spezifikation

9.4 Vergleich von MBSecMon und JavaMOP

In diesem Abschnitt der Evaluation werden der in dieser Arbeit vorgestellte MBSecMon-Ansatz mit dem *Monitor-oriented Programming* (MOP)-Ansatz [MJG+12] verglichen, der in Abschnitt 4.3.7 vorgestellt wurde. Dieser Vergleich basiert auf der Masterthesis [Lan13] und kann dort in vollem Umfang nachgelesen werden. Hierbei werden die Spezifikationssprachen Kontextfreie Grammatiken und Erweiterte Reguläre Ausdrücke als repräsentative Auswahl aus den möglichen Spezifikationssprachen die MOP unterstützt gewählt.

In der Arbeit [Lan13] wurde die Monitorgenerierung von Java-Monitoren um ein Konzept zur Einbindung der generierten Monitore in ein Java-Programm durch Verwendung von AspectJ erweitert. JavaMOP verwendet eine aspektorientierte Spezifikationssprache als Teil der Monitorspezifikation zur Beschreibung von Ereignissen. Dieser orientiert sich an der AspectJ-Syntax.

9.4.1 *Beispielsystem Alarmanlage*

Zur Evaluation der beiden Ansätze wird in diesem Abschnitt eine Alarmanlage als zu überwachendes System verwendet. Abbildung 9.19 zeigt eine schematische Darstellung dieser Alarmanlage. Die Alarmanlage besteht grundsätzlich aus einer *Zentrale*, die Signale von den Sensoren erhält und gegebenenfalls über den *Aktor* Alarm gibt. Als Sensoren stehen hierbei u. a. *Tür-* und *Fensterkontakte* und *Bewegungsmelder* zur Verfügung, die durch eine *Alarmsteuerung* ausgewertet werden. Die Zentrale besitzt eine Ein- bzw. Ausgabeschnittstelle, über die sie die Signale der Sensoren gefiltert durch die Alarmsteuerung erhält. Daraufhin wird durch die

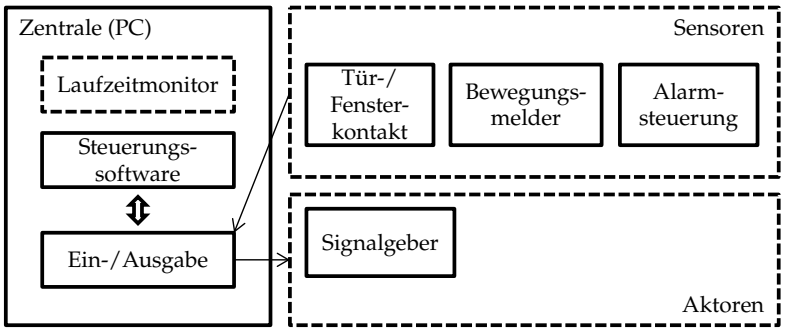


Abbildung 9.19: Schematische Darstellung einer Alarmanlage

Steuerungssoftware Alarm ausgelöst, indem der Signalgeber eine Ausgabe stattfindet.

Dieses zu überwachende System ist als Java-Programm implementiert. Das Ziel ist es hier einen *Laufzeitmonitor* zu generieren, der die Funktionsweise der Zentrale überwacht.

VERWENDETE SIGNATUREN Als Signaturen werden drei unterschiedliche Monitorspezifikationen, die in Abbildung 9.20 als eLSC dargestellt sind, verwendet. In MBSecMon wurden für die bessere Vergleichbarkeit synchrone Nachrichten gewählt, da JavaMOP nur ein Ereignis pro Methodenaufruf produziert.

Die Signatur *Einbruch* (E) stellt einen Usecase dar, der beschreibt, wann ein System einen Einbruch melden soll. Hierzu wird nach jeder *notify*-Nachricht, die vom Bewegungsmelder geschickt wird, die Variable *globalAlarm* der Zentrale auf *true* geprüft und erwartet, dass der Alarm durch den Aufruf der Methode *setAlarm* ausgelöst wird.

Fehlfunktion (F) überwacht als Misusecase, ob das System einen Alarm auslöst, obwohl dazu kein Grund besteht, indem die durch die *activate*-Methode übergebene PIN mit der gültigen PIN der Zentrale verglichen wird. Wenn sich diese unterscheiden und die Alarmsteuerung aktiviert wird, wurde der Misusecase erkannt.

Sequenznummern (S) ist als Usecase modelliert und überwacht, dass die Sequenznummern der *notify*-Nachrichten im System monoton steigend sind. Hierzu wird die Signaturinstanzen verschränkt ausgeführt und hierfür im Fall der MPNs das Konzept der Subgenerationen genutzt.

In JavaMOP werden die Ereignissequenzen als Kontextfreie Grammatiken (CFG) sowie als Erweiterte Reguläre Ausdrücke (ERE) spezifiziert. Hierbei stellt die Spezifikation als CFG die Besonderheit des MOP-Ansatzes dar, dessen effizienter Implementierung bei der Entwicklung große Bedeutung zugemessen wurde. ERE werden wie in Abschnitt 4.3.7 beschrieben vor der Codegenerierung in Endliche Automaten (FSM) übersetzt. Diese stehen in dieser Evaluation für alle Eingabesprachen, die in MOP nach FSM abgebildet werden.

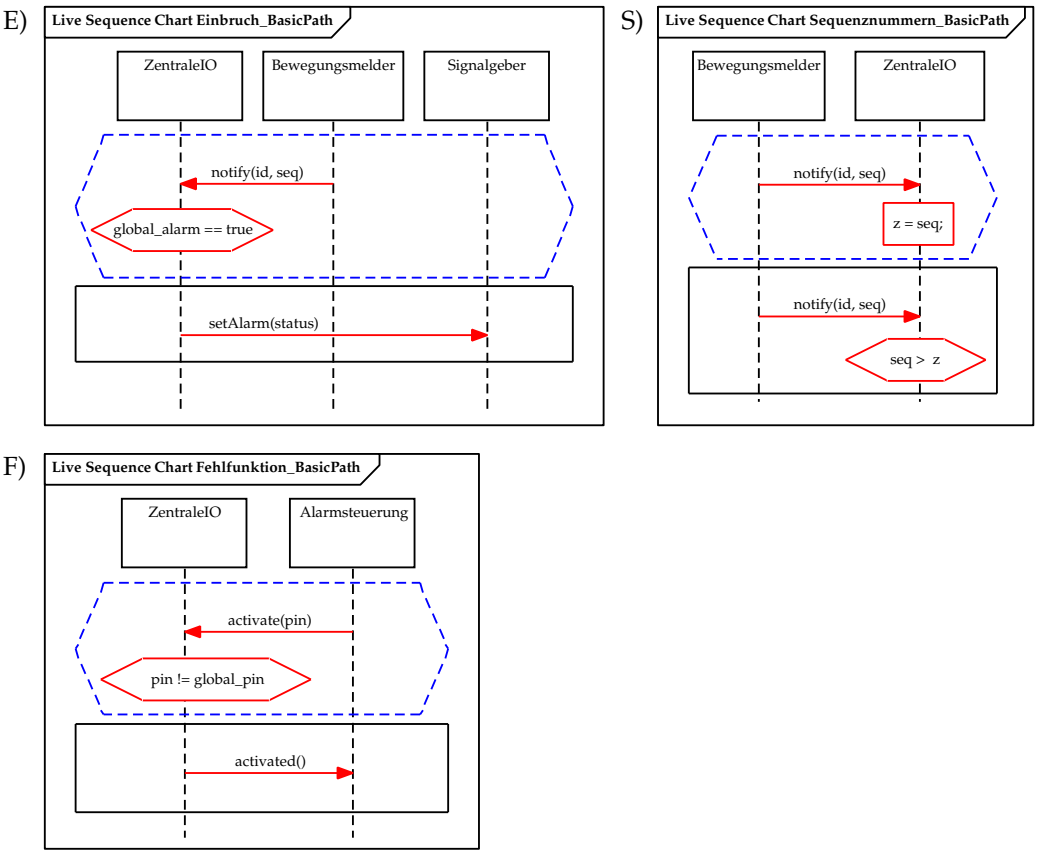


Abbildung 9.20: Signaturen zur Überwachung der Alarmanlage

9.4.2 *Ziel*

Ziel dieses Vergleichs ist die Betrachtung der Implementation JavaMOP mit der Java-Monitorgenerierung des MBSecMon-Prozesses. Hierbei wird insbesondere auf die Laufzeit und den bisher noch nicht betrachteten Speicherverbrauch der beiden Ansätze bei Überwachung paralleler Java-Programme eingegangen.

Bedingungen werden in MOP schon in den Aspekten definiert und zur Laufzeit, bevor ein Ereignis ausgelöst wird, überprüft. Im Gegensatz hierzu werden Ereignisse im MBSecMon-Prozess direkt an den Monitor übergeben und erst dort die Bedingung, die Teil der Signatur ist, überprüft. Diese zwei Konzepte führen zu einer unterschiedlichen Erzeugung von Monitorinstanzen.

9.4.3 *Messungen*

Zur Messung der Laufzeiteigenschaften werden zwei Millionen Kommunikationsstränge (Generationen) sequenziell ausgeführt. Die Messungen bestehen hierbei aus zwei Schritten: (1) Instantiierung der für den zu messenden Kommunikationsstrang notwendigen Objekte aus dem Alarmanlagenbeispiel und (2) Auslösen der Ereignisse durch Aufruf der Methoden im Beispiel.

Im Folgenden werden die vorgestellten Signaturen in Szenarien mit offenen und geschlossenen Kommunikationssträngen betrachtet. Offene sind dabei nicht beendete Monitore, deren Ergebnis noch nicht feststeht und geschlossene sind vollständig durchlaufene Monitore.

Als Ausführungsplattform wurde ein PC mit folgenden Eigenschaften verwendet:

- CPU: Intel Core i7-860 mit 2,80 GHz
- Arbeitsspeicher: 8 GB
- Software: Windows 7 Professional SP2, JavaMOP 3.0.0, MBSecMon-Tool

Zur besseren Vergleichbarkeit wurde die Garbage-Collection der JavaVM während der Laufzeit verhindert.

Im Folgenden werden verschiedene Szenarien betrachtet, die nach ihren genutzten Signaturen benannt sind.

- **F1** verwendet die Signatur *Fehlfunktion*, wobei nur die Methode *activate* aufgerufen wird. Es ergeben sich hierbei offene (nicht beendete) Monitorinstanzen. Dieses Szenario zeigt den schlimmsten Fall, in dem eine steigende Anzahl von offenen Monitorinstanzen erzeugt wird.
- **F2** ist dasselbe Szenario wie F1 mit dem Unterschied, dass anschließend zum Aufruf von *activate* noch die Methode *activated* aufgerufen wird. Hierdurch werden die Monitorinstanzen beendet. Dies zeigt, wie die Monitoransätze den Abschluss von Monitorinstanzen verarbeiten.
- **S1** verwendet die Signatur *Sequenznummern*, die verschränkt ausgeführt werden soll. Da MOP dies nicht unterstützt, wurde eine ähnliche Konstruktion gewählt, in der die Methode *notify* zweifach aufgerufen wird. Dies führt im MOP-Monitor zu einem geschlossenen und einem offenen Kommunikationsstrang.
- **E1** verwendet die Signatur *Einbruch*, in der drei Kommunikationsteilnehmer überwacht werden. Bei diesem Szenario werden die drei in der Signatur modellierten Methodenaufrufe nacheinander ausgeführt. Ziel ist die Betrachtung des Verhaltens der Monitore bei der Überwachung mehrerer Teilnehmer.
- **EF1** ist ein Szenario, in dem die beiden Signaturen *Einbruch* und Fehlfunktion eingesetzt werden. Hierbei werden nur die Methoden *activate* und *notify* ausgeführt, um eine wachsende Anzahl offener Monitorinstanzen zu erhalten. Hierdurch wird die Skalierbarkeit der beiden Ansätze bei Einsatz mehrerer Signaturen betrachtet.

Durchschnittliche Laufzeiten Tabelle 9.6 zeigt die durchschnittlichen Laufzeiten pro Kommunikationsstrang. Diese werden über die gemessenen zwei Millionen Kommunikationsstränge gemittelt, um eindeutige Messwerte zu erhalten. Im oberen Teil sind die einzelnen Szenarien aufgeführt und in der letzten Zeile der Durchschnitt über alle Szenarien.

	(1)	(2)	(3)	Vergleiche		
Untersuchung	MBSecMon	MOP-CFG	MOP-ERE	(1) vs. (2)	(1) vs. (3)	(2) vs. (3)
F1	0.0025	0.0023	0.0022	108%	116%	108%
F2	0.0024	0.0024	0.0021	97%	113%	116%
S1	0.0052	0.0025	0.0022	209%	242%	115%
E1	0.0037	0.0068	0.0053	55%	70%	128%
EF1	0.0053	0.0063	0.0062	85%	86%	101%
Gesamt	0.0038	0.0041	0.0036	94%	107%	113%

Tabelle 9.6: Vergleich der durchschnittlichen Ausführungszeiten je Kommunikationsstrang in [ms]

Es ist zu erwähnen, dass die Garbage-Collection während der Messung deaktiviert war. Da die Lebenszeit der Monitorinstanzen in MOP von der Lebenszeit der überwachten Objekte abhängt und diese Objekte während der Messung nicht freigegeben wurden, war während der Laufzeit bei JavaMOP eine Reorganisation der Datenstrukturen (HashMaps) notwendig. Dies hat Einfluss auf die Messergebnisse der Szenarien F2 und E1. Die hohen Werte für die Verarbeitung von Szenario S1, in dem bei MBSecMon Subgenerationen zum Einsatz kommen, sind nicht vollständig mit den Werten von Java MOP vergleichbar. Da JavaMOP die verschränkte Überwachung nicht unterstützt, wurde eine ähnliche aber funktional eingeschränkte Signatur verwendet, die schneller auszuwerten ist.

Die mit MBSecMon generierten Java-Monitore sind hinsichtlich der Verarbeitungszeit zwischen denen von MOP mit CFG als Spezifikationssprache und MOP mit ERE und damit FSMs als Spezifikationssprache einzuordnen. Die hier gemessenen Ausführungszeiten zeigen, dass beide Ansätze zur Überwachung von Programmen auf einem PC einsetzbar sind. Im schlechtesten Fall konnten durchschnittlich mehr als 200.000 Kommunikationsstränge in der Sekunde verarbeitet werden. Dies dürfte genügend Reserven bieten, um auch komplexere Muster zu überwachen. Der Vergleich der Laufzeiten der Java-Implementierung mit der C-Implementierung der generierten Monitore des MBSecMon-Prozesses in Abschnitt 9.1 lässt darauf schließen, dass die Java-Implementierung starkes Optimierungspotential bietet.

DURCHSCHNITTLICHER SPEICHERBEDARF Tabelle 9.7 zeigt den Datenspeicherbedarf der Monitore der verschiedenen Ansätze. Bei JavaMOP und MBSecMon ist auch hier zu bemerken, dass der Garbage-Collector während der Messung deaktiviert war und nur am Ende der Messung einmalig aufgerufen wurde, um nicht mehr referenzierte Objekte abzuräumen. Hierdurch findet keine Deallokation der erstellten Monitore zur Laufzeit statt. Auch der Unterschied bei S1 in der Umsetzung von Subgenerationen gilt für diese Messung.

Die Szenarien F1, S1 und EF1 stellen den schlechtesten Fall dar, da in einem realen Monitor davon auszugehen ist, dass der größte Teil der Kommunikationsstränge sich in einer absehbaren Zeit beendet (geschlossen wird). Nur wenige Kommunikationsstränge werden bei der Monitorausführung offenbleiben. Somit

	(1)	(2)	(3)	Vergleiche		
Untersuchung	MBSecMon	MOP-CFG	MOP-ERE	(1) vs. (2)	(1) vs. (3)	(2) vs. (3)
F1	1050.79	746.38	490.00	141%	214%	152%
F2	97.98	746.38	490.00	13%	20%	152%
S1	1299.42	762.40	506.02	170%	257%	151%
E1	187.90	2365.68	1412.06	8%	13%	168%
EF1	2098.54	1628.76	1284.41	129%	163%	127%
Gesamt	946.93	1249.92	836.50	76%	113%	149%

Tabelle 9.7: Vergleich des durchschnittlichen Speicherverbrauchs in [MB]

stellt das hier in diesen Szenarien dargestellte Verhalten eine obere Schranke des Speicherverbrauchs dar.

Auffällig ist der große Unterschied zwischen MBSecMon- und den MOP-Monitoren bei den Messungen F2 und E1. In diesen Fällen, in denen die Monitore bis zum Ende durchgeführt werden, ist zu sehen, dass bei MOP die Monitorinstanzen noch nicht freigegeben sind. In MBSecMon wurden alle vollständig durchgeführten Monitore direkt dereferenziert und konnten somit vom Garbage-Collector abgeräumt werden.

Auch bei der Evaluation des Speicherverbrauchs ist der MBSecMon-Ansatz zwischen JavaMOP mit CFG und JavaMOP mit ERE als Spezifikationssprache einzuordnen. Der geringere Speicherverbrauch von JavaMOP gegenüber MBSecMon ist darauf zurückzuführen, dass in der Monitorimplementierung von JavaMOP deutlich stärker auf primitive Datentypen gesetzt wird als in der Java-Implementierung von MBSecMon. Somit fallen die Kosten der Verwaltung von Monitorinstanzen in JavaMOP größtenteils wesentlich geringer aus. Dagegen bietet der MBSecMon-Ansatz in der MPN-Sprache selbst die verschränkte Überwachung einer Signatur und die Verwendung zusätzlicher Informationen in Bedingungen und Aktionen an, die in JavaMOP nur sehr eingeschränkt im aspektorientierten Teil der Spezifikation verwendet werden können.

9.4.4 *Schlussfolgerung*

Diese Betrachtung hat gezeigt, dass Monitor-Petrinetze auch bei kleinen Signaturen einsetzbar sind und die Java-Implementierung mit der des MOP-Ansatzes vergleichbar ist, obwohl sie weitere Konzepte wie die Subgenerationen, die von MOP nicht angeboten werden, besitzt. Die Java-Implementierung des MBSecMon-Ansatzes ist, wie in Abschnitt 9.1 gezeigt, nicht auf Speicherverbrauch und Laufzeit optimiert, sondern setzt auf objektorientierte Konzepte in denen viele Objekte angelegt und wieder verworfen werden. Des Weiteren waren die verwendeten Signaturen im Fall von MPN nicht minimal in Hinsicht auf ihre Plätze, da diese aus eLSC-Spezifikationen generiert wurden. Für jede Lebenslinie existierte ein Platz, da die Optimierungen, die in Abschnitt 7.3.2 besprochen wurden, noch nicht implementiert waren.

In diesem Vergleich wurden relativ einfache Ereignismuster ohne Parallelität in der eigenen Signatur und vor allem ohne asynchrone Nachrichten überwacht.

Hier hat sich gezeigt, dass sich die aus MPN-Signaturen generierten Monitore mit denen aus EREs (FSMs) und CFGs im JavaMOP-Konzept messen können. Ein genauerer Vergleich der Zwischensprachen, die kein Parallelitätskonzept von sich aus unterstützen, und den MPNs findet in Abschnitt 9.6 anhand der FSMs statt.

9.5 AUTOSAR-Integration: Automatikschaltung

In diesem Abschnitt der Evaluation werden der Einsatz und die Anpassbarkeit des MBSecMon-Ansatzes, insbesondere die Generierung von Monitoren aus der MPN-Sprache, für ein in Bezug auf die Schnittstellen und Datentypen restriktives System betrachtet. Die Evaluation basiert auf der Veröffentlichung [PPPM13]. Die *AUTomotive Open System ARchitecture (AUTOSAR)*[6] (Abschnitt 2.1) ist ein offener Industriestandard der Automobilindustrie, der zum Ziel hat eine gemeinsame Plattform für die Entwicklung von Fahrzeugsystemen zu bieten. Er bietet eine modulare Softwarearchitektur mit standardisierten Schnittstellen und eine Laufzeitumgebung (*engl. Runtime Environment*) (RTE), die die Softwarekomponenten der Applikationsebene von den darunter liegenden Modulen der Basic-Softwareschicht und der tatsächlichen Hardware trennt. Ziel dieser Trennung ist besser mit der wachsenden Komplexität der Software in Fahrzeugen zurechtzukommen.

Der AUTOSAR-Standard bietet, wie in Abschnitt 4.4 besprochen, nur die Spezifikation von Monitoren zur Überwachung des Kontrollflusses und von Zeitaspekten auf einem niedrigen Abstraktionsniveau an. Jedoch bietet es keine Unterstützung für die Spezifikation und Generierung komplexer Monitore auf dem Abstraktionsniveau der Softwarekomponenten.

In [PPPM13] wurde der MBSecMon-Prozess in den AUTOSAR-Entwicklungsprozess integriert. Im Folgenden werden die durch den MBSecMon-Ansatz generierten Monitore evaluiert und deren Einsetzbarkeit in einem AUTOSAR-System betrachtet.

9.5.1 *Integration des MBSecMon-Prozesses in den AUTOSAR-Entwicklungsprozess*

In diesem Abschnitt wird für die Evaluation des MBSecMon-Ansatzes ein kleines realistisches AUTOSAR-System verwendet. Abbildung 9.21 zeigt ein vereinfachtes System einer Automatikschaltung als Komponentendiagramm. Es besteht aus vier Komponenten – *ShiftLogic*, *Transmission*, *Engine* und *Vehicle*. Hierbei repräsentiert die Komponente *Vehicle* die physischen Eigenschaften des Fahrzeugs. Zur Kommunikation zwischen den Komponenten wird das Sender-Receiver-Muster verwendet, das zur asynchronen Kommunikation zwischen AUTOSAR-Komponenten vorgesehen ist. Hierbei übergibt der Sender den zu übertragenden Wert an die RTE und die empfangende Komponente ruft bei Bedarf entsprechenden Wert bei der RTE ab.

In Abbildung 9.21 sind zusätzlich die Eingabeports markiert, die die Werte von Gas (*engl. Throttle*) und Bremsmoment (*engl. BrakeTorque*) von außen übergeben bekommen. Überwacht werden sollen in diesem Beispiel die Kommunikation zwischen den Ports, die in der Abbildung als *Instrumentierte Ports* markiert sind.

[6] AUTOSAR-Website: http://www.autosar.org/

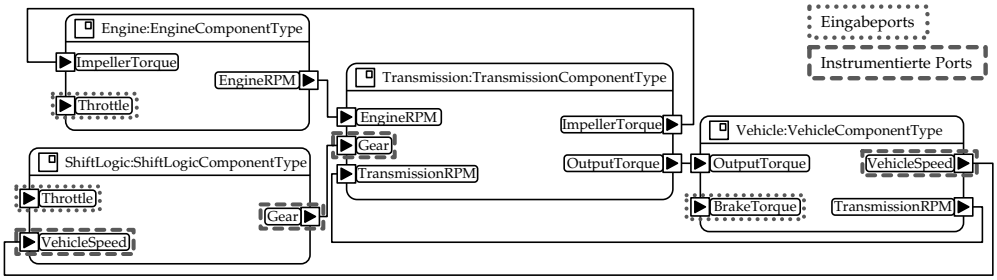


Abbildung 9.21: Die Automatikschaltung als Komponentendiagramm

Die Implementierung der Komponenten basiert auf einem dem Mathworks Matlab/Simulink-Werkzeug beiliegenden Beispielprojekt einer Automatikschaltung[7], das für die Generierung AUTOSAR-konformen C-Codes leicht angepasst wurde. Kontinuierliche Blöcke im Simulinkmodell, die nicht mit der Codegenerierung für AUTOSAR kompatibel sind, wurden durch ihre diskreten Gegenstücke ersetzt. Die AUTOSAR-konforme Implementierung wird auf Basis dieses Modells mit Mathworks Embedded Encoder generiert. Integriert wird diese Implementierung in eine AUTOSAR-Simulationsumgebung, die mithilfe des Werkzeugs OptiXware Embedded Architect auf Basis des in Abbildung 9.21 Komponentendiagramms generiert wurde.

9.5.2 *Ziel*

Das Ziel dieser Fallstudie ist die Bestimmung des Overheads von Monitoren in einem kleinen realen System sowie die Integration von Monitoren in eine vorhandene in Bezug auf Schnittstellen und Datentypen restriktive Softwarearchitektur.

9.5.3 *Anpassungen des MBSecMon-Prozesses*

Abbildung 9.22 zeigt den an die Generierung von Monitoren für AUTOSAR-Systeme angepassten MBSecMon-Prozess. Auf der linken Seite ist ein stark vereinfachter AUTOSAR-Prozess dargestellt. Als Eingabe dient in diesem die AUTOSAR-Systemspezifikation, die im ARXML-Format vorliegt. Diese beinhaltet unter anderem eine Systembeschreibung auf Komponentenbasis, wie sie in Abbildung 9.21 dargestellt ist, und weitere Informationen, wie die Namen und Typen der über die Ports übermittelten Daten. Aus dieser wird über ein AUTOSAR-Werkzeug Applikationscode erzeugt, der dann auf der RTE ausgeführt wird.

Zur Einbindung dieser Spezifikation in den Monitorgenerierungsprozess wird, wie in [Pat14] beschrieben, die Systemstruktur als Komponentendiagramm in das *MBSecMon-EA-Add-in* importiert. Auf Basis dieser Daten können nun die Signaturen in der MBSecMonSL modelliert werden.

Für die Generierung der Monitore mussten nur kleine Erweiterungen an der Codegenerierung und der PSI-Datei vorgenommen werden. Als Basis diente die

[7] Automatic Transmission Beispiel: `http://www.mathworks.de/help/simulink/examples/modeling-an-automatic-transmission-controller.html`

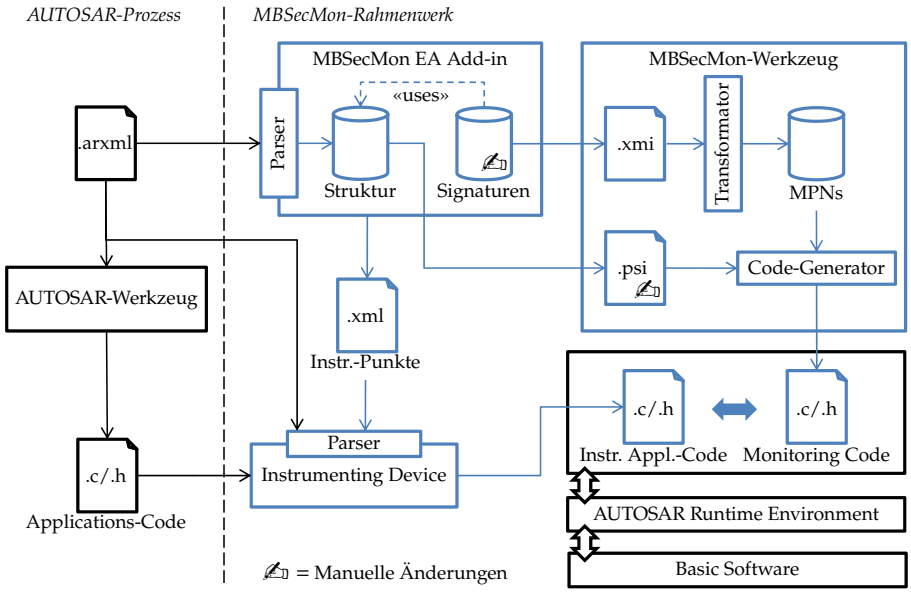


Abbildung 9.22: Der angepasste Generierungsprozess für Monitore in AUTOSAR

vorhandene Monitorgenerierung für C. Der AUTOSAR-Standard sieht Namenskonventionen für Schnittstellen vor, die im System genutzt werden. Des Weiteren muss zur Einbindung in das AUTOSAR-System, die Schnittstellensignaturen, die das *Instrumenting-Device* [PWMS12] zur Instrumentierung des Systemcodes mit Aufrufen des Monitors benötigt, bekannt sein. Die PSI wurden um Einträge für die Beschreibung von individuellen Schnittstellen erweitert, die Informationen über den Namen der benötigten Schnittstellen des Monitors und deren Parametern, wie den speziellen AUTOSAR-Datentyp, enthalten. Diese können beim Export der Signaturen aus dem *MBSecMon-EA-Add-in* automatisch aus der importierten Systemspezifikation auf Basis der Signaturen gewonnen werden.

Mit diesen kleinen Anpassungen ist es nun möglich Monitore für AUTOSAR-Systeme zu generieren, die sich an den AUTOSAR-Standard halten und diese über das Instrumenting-Device in das System einzubinden.

9.5.4 *Evaluation der generierten Monitore*

Im Folgenden werden die mit dem angepassten Prozess generierten Monitore genauer auf ihren Speicherverbrauch und ihr Laufzeitverhalten gegenüber dem AUTOSAR-System betrachtet.

SIGNATUREN In dieser Evaluation werden Signaturen verwendet, die das Subgenerationskonzept der MPNs verwenden und Zuweisungen von Variablen sowie Bedingungen besitzen. Hierdurch soll evaluiert werden, ob auch solche relativ komplexen Signaturen, in denen mehrere Instanzen der Signaturen zur Laufzeit erstellt und verworfen werden, in einem AUTOSAR-System einsetzbar sind. Abbildung 9.23 zeigt die beiden eingesetzten Signaturen *AcceptedSpeedChanges* (ASC) und *InvalidGearChanges* (IGC). Durch die jeweils nebenläufige Ausführung

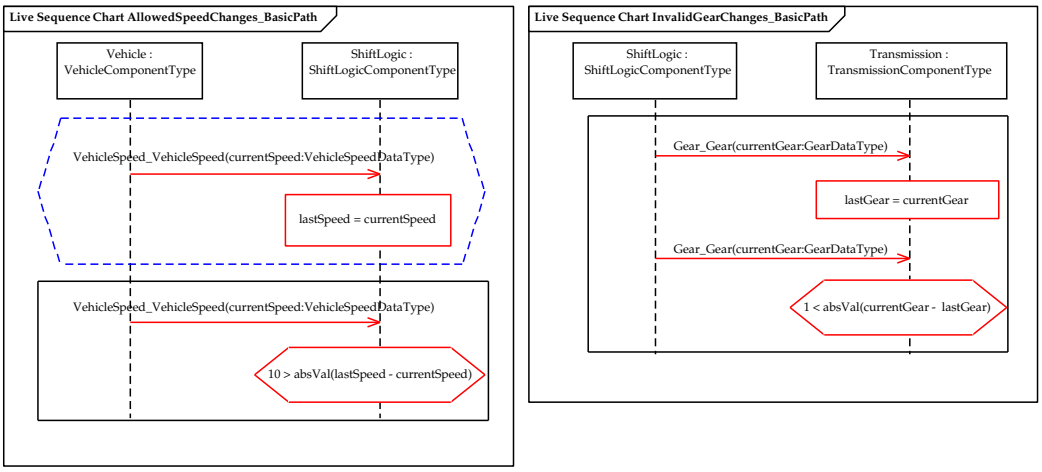


Abbildung 9.23: Nebenläufig ausgeführte Signaturen zur Überwachung des AUTOSAR-Systems

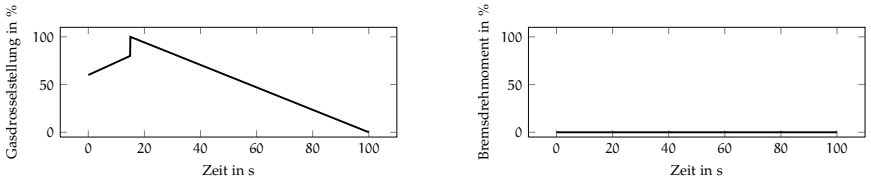


Abbildung 9.24: Eingabedaten für das AUTOSAR-System

überwachen diese Signaturen ähnlich wie die in Abschnitt 5.4 vorgestellten verschränkten Signaturen in jedem Schritt die Veränderung des übergebenen Wertes. Hierbei handelt es sich bei ASC um eine positive Signatur, deren Fehlschlagen einen Fehler darstellt und bei IGC um eine negative Signatur.

MESSUNGEN Die Messungen werden in der OptXware Embedded Architect Simulationsumgebung für AUTOSAR durchgeführt. Diese läuft auf einem AMD Phenom II X4 955 Prozessor, mit 3.20 GHz. Die Messungen werden mit den Windows Win32-API-Funktionen MyQueryPerformanceCounter und MyQueryPerformanceFrequency gemessen, die eine CPU-Tick-Granularität bieten. Diese Messungen werden in Beziehung mit den Laufzeiten des AUTOSAR-Systems gesetzt, um einen Eindruck über die Dimension des Overheads, der durch die Monitore eingeführt wird, zu gewinnen.

Als Eingabe für die Messung der Laufzeit des Systems wird die in Abbildung 9.24 gezeigte Kurve gewählt, die ein Überholmanöver darstellt.

LAUFZEITANALYSE Zunächst wird der Einfluss der beiden jeweils verschränkt ausgeführten Monitore auf den Aufruf der RTE-Methoden betrachtet. Tabelle 9.8 zeigt für die einzelnen instrumentierten Ports die Laufzeit ohne (Original) und mit Monitoren (Instr.). Die Zugriffe auf die RTE beinhalten bei der Verwendung des Sender-Receiver-Musters lediglich das Schreiben eines Wertes in eine Varia-

Komponente	RTE-Aufruf	Mittelwert		
		Original (ticks)	*Instr. (ticks)*	*Diff. (%)*
ShiftLogic()	Rte_Read_VehicleSpeed ...	24,76	38,87	57
ShiftLogic()	Rte_Write_Gear ...	18,00	23,60	31
Transmission()	Rte_Read_Gear ...	25,28	37,63	49
Vehicle()	Rte_Write_VehicleSpeed ...	21,56	27,62	28

Tabelle 9.8: Vergleich der Ausführungszeit der originalen und der überwachten Aufrufe der RTE

Komponente	Mittelwert			
	Original (ticks)	*Instr. (ticks)*	*Diff. (ticks)*	*Diff. (%)*
ShiftLogic()	56,03	70,97	14,94	28
Transmission()	57,28	63,77	6,49	11
Vehicle()	48,63	58,39	9,76	20

Tabelle 9.9: Vergleich der Ausführungszeit der originalen und überwachten Komponenten

ble bzw. das Lesen aus einer Variablen, die in der RTE verwaltet wird. Hierbei wird ggf. die Einhaltung des in der Systemspezifikation definierten Wertebereichs des Variablentyps überprüft. Für den schreibenden Zugriff auf die RTE liegt der Overhead bei 28% und 31% und für den lesenden Zugriff bei 49% und 57%. Die gemessene Zeit beinhaltet bei der instrumentierten Variante der RTE-Aufrufe den Wrapper um die Aufrufsmethode, den Aufruf des Monitors und die Verarbeitung des übergebenen Ereignisses und Datenwerts durch den Monitor.

Der erhöhte Wert für die lesenden Zugriffe resultiert aus der Struktur der Monitore, die auf der empfangenden Seite Zuweisung von Variablen und Auswertungen von Bedingungen enthalten. Diese Bedingungen und Aktionen haben somit einen großen Anteil an der Gesamtlaufzeit des Monitors.

In Tabelle 9.9, wird im Gegensatz zur vorherigen Betrachtung der Einfluss des Laufzeitoverheads der Monitore auf die Laufzeit der Softwarekomponenten betrachtet. Bei der *ShiftLogic*-Komponente sind zwei Ports instrumentiert, was zu einem höheren Overhead in Relation zu den anderen SW-Komponenten, bei denen nur ein Port instrumentiert ist, führt. Insgesamt liegt der Overhead der Komponente *ShiftLogic* bei 28%. Dies entspricht ungefähr dem Overhead für einen lesenden und einen schreibenden Zugriff auf die Ports.

SPEICHER-ANALYSE Zur Messung des Speicherverbrauchs wird das Werkzeug *objdump* der *GNU Binutils Toolsuite* angewendet. Dieses arbeitet auf den kompilierten Objektdateien und liefert eine Übersicht über den Speicherverbrauch des Quellcodes (Code) und der drei Datensektionen *Data*, *bss* (uninitialisierte Daten) und *rdata* (Read-Only-Daten).

Tabelle 9.10 zeigt den Speicherbedarf des Systems und der Monitore. Hierbei ist zu erkennen, dass die Wrapper um die RTE-Aufrufe, die die Ereignisse und Daten an den Monitor weiterleiten, nur einen sehr kleinen Einfluss auf den benötigten Codespeicher und keinen Einfluss auf den benötigten Datenspeicher haben. Der

	Wrapper				Monitor		
Typ	*Read_VS*	*Write_VS*	*Read_G*	*Write_G*	*Contr.*	*ASC*	*IGC*
Code	48	48	48	48	1632	2512	2160
Data	0	0	0	0	88	40	32

Tabelle 9.10: Speicheroverhead der Monitore in Byte

Monitor besteht wie in Kapitel 7 beschrieben aus einem *Controller* (Contr.), der die einzelnen Signaturen verwaltet und anspricht, und den Signaturen *AllowedSpeedChanges* (ASC) und *InvalidGearChanges* (IGC) selbst. Der durch den Monitor eingeführte Overhead der Datensektion ist sehr gering und wächst nur langsam mit der Größe der Signatur, da nur der Zustand der Monitorinstanzen und die zwischengespeicherten Variablen dort gespeichert werden.

SCHLUSSFOLGERUNG In dieser Evaluation wurde ein kleines AUTOSAR-System, bestehend aus rudimentären SW-C-Implementierungen, mit Monitoren überwacht, die das Subgenerationskonzept der MPNs verwenden. Die Implementierungen des Systems bestehen größtenteils nur aus wenigen Simulink-Blöcken und Tabellen. Aus diesem Grund zeigt sich bei der Betrachtung ein relativ hoher Overhead in Bezug auf die Laufzeit und den Speicherbedarf des Systems an sich. Jedoch ist zu beachten, dass die Verarbeitung eines Ereignisses im Extremfall nur 15 CPU-Ticks auf einem PC-Prozessor in Anspruch nimmt. Zusammen mit der Betrachtung der Skalierbarkeit der Monitore auf eingebetteten Systemen in Abschnitt 9.1 lässt sich schlussfolgern, dass die generierten Monitore mit MPN als Zwischensprache in einem solchen realen System einsetzbar sind.

Diese Evaluation hat gezeigt, dass der MBSecMon-Prozess an restriktive Systeme mit starken Anforderungen an den generierten Code mit relativ kleinem Aufwand angepasst werden kann. Informationen der AUTOSAR-Systemspezifikation wurden in den MBSecMon-Prozess einbezogen. Hierdurch können Monitore automatisch aus den Signaturen, die in der MBSecMonSL vorliegen, generiert und über eine Instrumentierung in das zu überwachende System eingebunden werden.

9.6 LAUFZEIT UND SPEICHERVERBRAUCH IM GEGENSATZ ZU FSMS

Basierend auf den unterstützen Sprachen in JavaMOP wird in diesem Abschnitt ein Vergleich zwischen speziellen Konzepten und deren Abbildung in die verschiedenen Zwischensprachen betrachtet. Hierbei wird insbesondere auf für die einzelnen Sprachen problematischen Aspekte eingegangen.

CFG und ERE sowie FSM können keine parallelen Abläufe darstellen. Hierdurch muss im Fall, dass eine Signatur in einer Spezifikationssprache, die Parallelität unterstützt, modelliert wurde, der parallele Teil ausmultipliziert werden. Beispiele für solche Modellierungselemente in eLSCs sind:

- Asynchrone Nachrichten,
- Par-Fragmente,

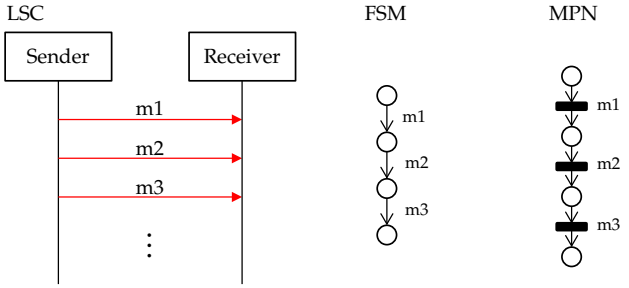


Abbildung 9.25: Abbildung einer synchronen Signatur als FSM und MPN

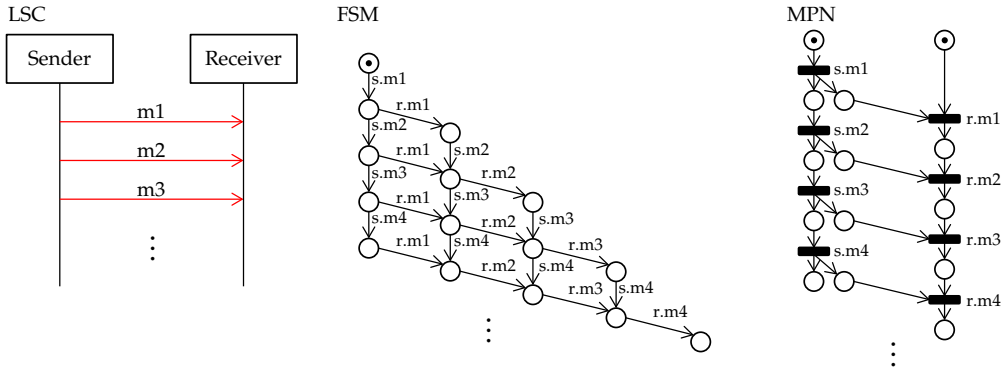


Abbildung 9.26: Abbildung einer asynchronen Signatur als FSM und MPN

- Ignore-/Forbidden-Fragmente,
- kalte Elemente,
- sowie zwei unabhängige Abläufe auf bzw. zwischen verschiedenen Lebenslinien einer Signatur.

Im Folgenden wird die Übersetzung einiger dieser Modellierungselemente in die Zwischensprachen am Beispiel genauer betrachtet. Abbildung 9.25 zeigt hierzu zunächst eine Übersetzung einer synchronen Kommunikation in FSM und MPN. Auffällig hierbei ist, dass sich die Modellgröße der beiden Zwischensprachen identisch verhält. Sowohl die FSMs als auch die MPNs wachsen linear in Bezug auf Plätze/Zustände und Transitionen. Wie in Abschnitt 9.1 beschrieben, ist die Laufzeit des MPNs zur Verarbeitung eines Ereignisses von der Anzahl der Transitionen in dem MPN mit demselben annotierten Ereignis abhängig. Dies unterscheidet sich bei einer FSM. Eine FSM besitzt nur einen aktiven Zustand und die Laufzeit einer Ereignisverarbeitung hängt von der Anzahl der ausgehenden Transitionen des aktiven Zustandes ab. Somit ist die Verarbeitung eines Ereignisses in einer FSM lokal, während bei einem MPN alle infrage kommenden Transitionen im Netz überprüft werden müssen.

In Abbildung 9.26 ist die Übersetzung einer asynchroner Kommunikation als eLSC nach FSM und MPN dargestellt. Die eLSCs haben eine schwache partielle Ordnung der Sende- und Empfangsereignisse, was bei FSMs zu vielen verschie-

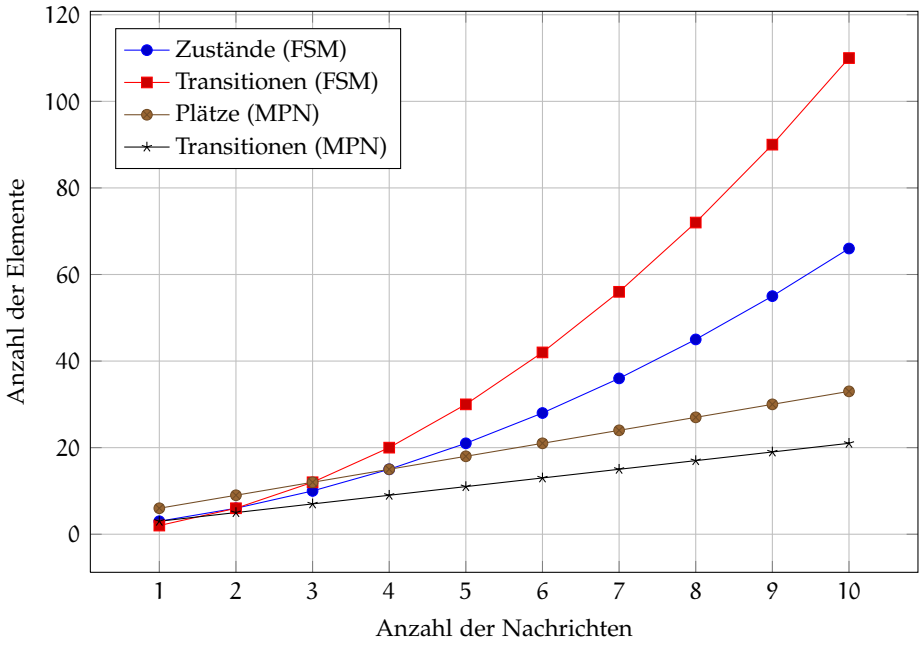


Abbildung 9.27: Anstieg der Anzahl der Elemente in FSM und MPN bei asynchroner Kommunikation

denen möglichen Sequenzen führt, die explizit ausmultipliziert werden müssen. Schon bei diesem asynchronen Muster ist in Abbildung 9.27 zu erkennen, dass die Anzahl der Zustände in FSMs mit wachsender Anzahl der Nachrichten mehr als linear anwachsen. Dies gilt sowohl für die Zustände als auch für die Anzahl der Transitionen. MPNs hingegen wachsen in beiden Aspekten linear mit der Anzahl der Nachrichten und haben schon bei sehr wenigen Nachrichten (2) einen Vorteil gegenüber den FSMs.

In der eLSC-Sprache existiert zusätzlich noch das Par-Fragment, das eine parallele Ausführung verschiedener Sequenzen beschreibt. Abbildung 9.28 zeigt ein solches Fragment in Kombination mit jeweils nur einer synchronen Nachricht in den parallelen Teilen. Zur Bestimmung der Werte wurden bei der Erstellung der FSM Zustände zusammengefasst, auf die identische Ereignissequenzen folgen. Auch hier wächst, wie in Abbildung 9.29 zu sehen ist, die Anzahl der Elemente für die MPN-Signatur bei der Übersetzung nur linear an, während die der FSM exponentiell steigt.

Ignore-Fragmente beschreiben, wie in den vorherigen Beispielen gezeigt, zu ignorierende Ereignisse. Diese müssen in der Zwischenrepräsentation explizit modelliert werden, wenn es sich um eine kontinuierliche Signatur handelt, in der nur modellierte Ereignisse vorkommen dürfen. Durch die parallel aktiven Plätze in MPNs kann ein zusätzlicher Platz erzeugt werden, an den diese Transitionen mit den zu ignorierenden Ereignissen annotiert werden können. Im Gegensatz hierzu muss in FSMs an jeden der Zustände in dem Bereich, in dem ein Ereignis ignoriert werden soll, explizit eine Transition modelliert werden. Dasselbe gilt für die Ereignisse in den Forbidden-Fragmenten.

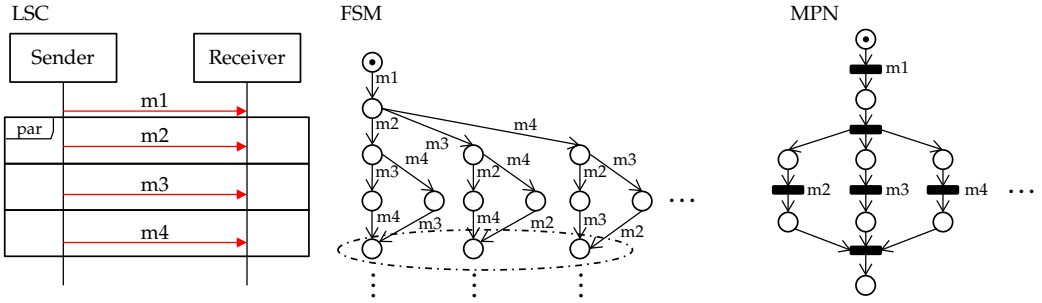


Abbildung 9.28: Abbildung eines Par-Fragments in einer synchronen Signatur als FSM und MPN

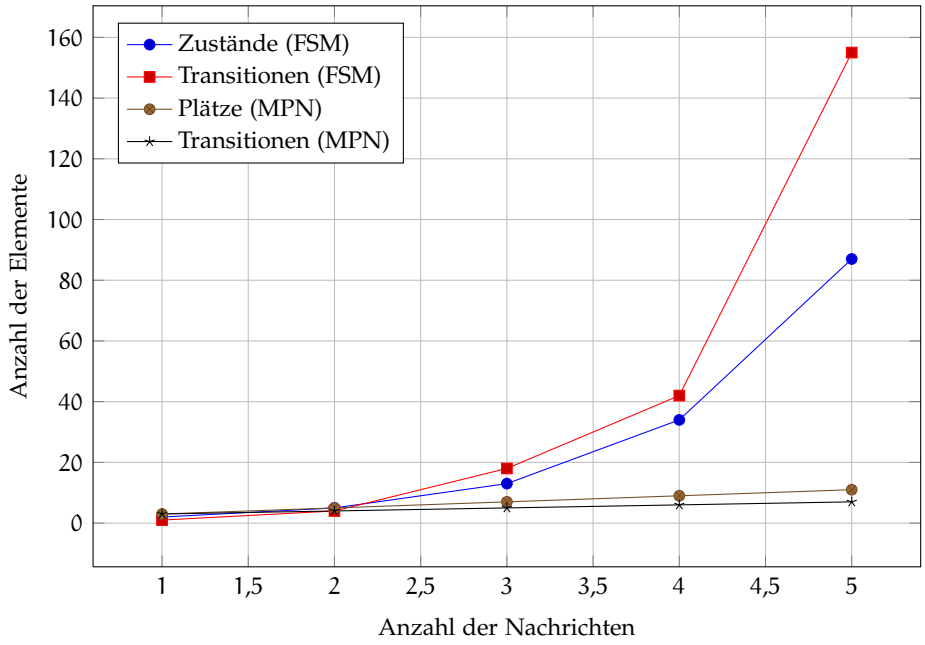


Abbildung 9.29: Anstieg der Anzahl der Elemente in FSM und MPN bei Einsatz eines Par-Fragments in den eLSCs

Kalte Elemente (Nachrichten) in LSCs werden zu Übersprungstransitionen übersetzt. In MPNs können diese an die entsprechenden Plätze angehängt werden. In FSMs führen diese durch das Ausmultiplizieren der Zustände zu einer entsprechend höheren Anzahl von Transitionen in der Zwischenrepräsentation.

Des Weiteren kommt es bei der Spezifikation auf Ebene der eLSCs auch dazu, dass zwei größtenteils unabhängige Abläufe in einer Signatur, die keine überlappenden LSC-Objekte besitzen, auftreten. Dies führt ähnlich zum Par-Fragment zu einer Ausmultiplikation in einem FSM während MPNs diese Parallelität direkt abbilden können.

SCHLUSSFOLGERUNG Wie schon an den einfachen Beispielen zu sehen ist, sind FSMs, wie sie u. a. im Monitorrahmenwerk MOP als Spezifikationssprache eingesetzt werden, nur eingeschränkt zur Modellierung komplexer, asynchroner Kom-

munikationsthreads mit beliebig verschachtelten nebenläufigen Kommunikationsprozessen geeignet. Die hier betrachteten Szenarien spiegeln nur das einzelne Auftreten paralleler Konzepte wieder. In realen Signaturen werden die vorgestellten Konzepte jedoch auch kombiniert angewendet. So wird ein Par-Fragment mit Sequenzen an asynchronen Nachrichten in den Teilen des Par-Fragments kombiniert.

Auch Erweiterte Reguläre Ausdrücke, Temporale Logiken, die im MOP nach FSM übersetzt werden, und Kontextfreie Grammatiken, bieten keine Konzepte zur Beschreibung von Parallelität. Parallelität lässt sich in diesem Fall eingeschränkt durch von außen angesteuerte Teilsignaturen simulieren, wodurch die formale Grundlage nur für die Teilsignaturen gilt. MOP verwendet diese Formalismen nur zur Beschreibung von erlaubten und verbotenen Ereignissequenzen und beschreibt Bedingungen in AspectJ und Reaktionen in der Zielsprache.

Die Monitor-Petrinetze sind auf die Spezifikation von Monitoren angepasst und skalieren beim Speicherverbrauch bei parallelen Sequenzen, besser als FSMs. Bei kleinen, einfachen Sequenzen haben FSM in der Ausführungszeit ihre Vorteile, jedoch werden MPNs schon bei ein wenig komplexeren Signaturen effizienter. Des Weiteren sind viele beim Monitoring benötigte Aspekte in der MPN-Sprache formalisiert enthalten.

9.7 Auswirkungen des Referenzsystems

Das in Kapitel 6 eingeführte Referenzsystem ermöglicht nicht nur die einfache Modellierung von Zusammenhängen zwischen Teilsignaturen, sondern reduziert auch die Redundanz in den einzelnen Signaturen. So können Vorbedingungen, die schon in der Basissignatur geprüft werden, aus den abhängigen Teilsignaturen entfernt werden und durch einen Erweiterungspunkt, der den Gültigkeitsbereich der Teilsignatur in der Hauptsignatur angibt, ersetzt werden. Dieses Referenzsystem führt natürlich wiederum selbst einen Overhead ein, der Laufzeit beansprucht. Im Folgenden wird an einem kleinen Beispiel eine erste Evaluation stattfinden, ob das Referenzsystem Vorteile gegenüber der Einzelmodellierung der Signaturen bringt.

Signaturen Für eine erste Evaluation des Referenzsystems werden die in Abbildung 9.30 und 9.31 in der MBSecMonSL modellierten Signaturen herangezogen. In beiden Monitorspezifikationen ist die Signatur *CARDME_Local* identisch und stellt den vereinfachten Ablauf des CARDME-Protokolls dar.

In Abbildung 9.30 ist der Monitor, der aus Use- und Misusecase besteht, ganzheitlich modelliert. Die erste Signatur *CARDME_Local* stellt den normalen Ablauf des CARDME-Protokolls dar, der auf Einhaltung geprüft wird. Die zweite Signatur ist ein Misusecase der die Daten der ersten und zweiten Präsentationsphase auf Plausibilität überprüft. Hierzu werden, nachdem Präsentationsphase 1 und 2 abgeschlossen sind, die übermittelten Daten an die Methode plausibleData(data _P, data_P2) übergeben und falls diese ein negatives Ergebnis zurück liefert ein Fehler erkannt. Eine mögliche Gegenmaßnahme wurde in diesem Beispiel ver-

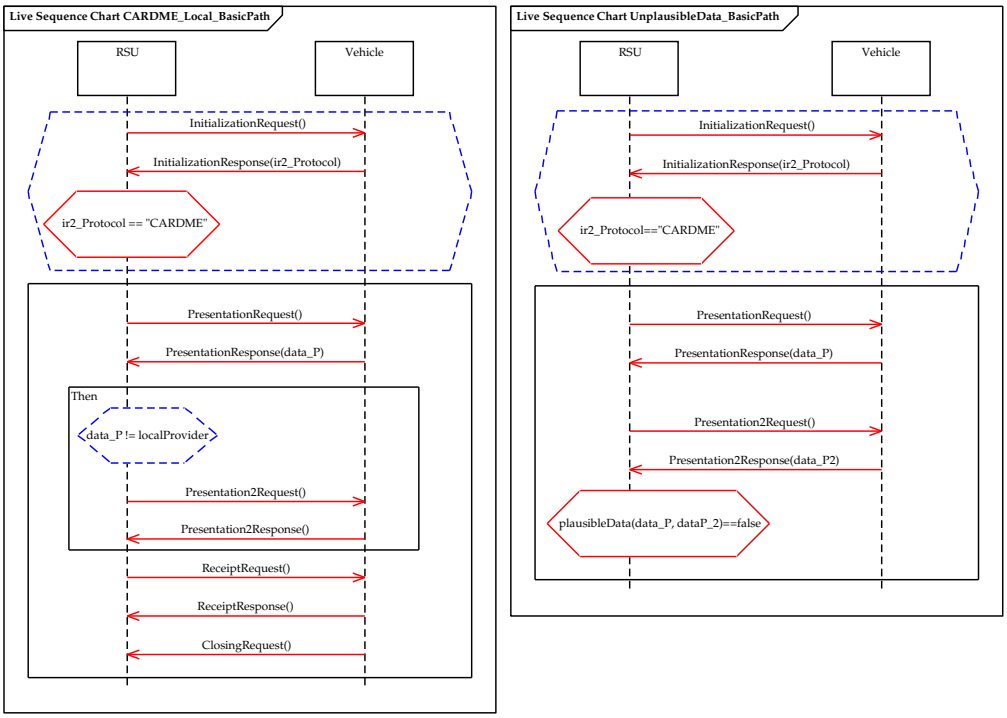


Abbildung 9.30: Signatur mit Use- und Misusecase ohne Referenzsystem

nachlässigt, da sie sich nicht als eigenständige Signatur ohne das Referenzsystem modellieren lässt und so in beiden Fällen identisch wäre.

In beiden Signaturen wird dieselbe Vorbedingung geprüft, um eine unabhängige Auswertung der beiden Teilsignaturen zu ermöglichen. Solche ganzheitlich modellierten Signaturen, die nicht voneinander abhängen, können beliebig kombiniert werden.

Im Gegensatz dazu zeigt Abbildung 9.31 denselben Monitor als kombinierte Signaturen, wobei der dargestellte Misusecase dem Mainchart des Usecases über eine «threaten»-Beziehung zugeordnet ist. Der Misusecase wird somit erst überwacht, wenn die Überwachung des Maincharts in der Signatur *CARDME_Local* beginnt.

MESSUNGEN Aus den Signaturen werden mit dem MBSecMon-Tool MPNs generiert und daraus wiederum C-Quelltext erzeugt. Die Messung wird auf einem PC[8] mithilfe des QueryPerformanceCounters der WIN32-API durchgeführt. Zur Bestimmung der Messwerte wird die Signatur pro Messung 100-mal durchlaufen und die Messung 20-mal wiederholt. Über den Messwerten wird anschließend der Median gebildet.

AUSWERTUNG Abbildung 9.32 zeigt die Laufzeit, die für 100 Durchläufe durch die Signaturen benötigt wird. Wie zu sehen ist, wirkt sich der Einsatz des Referenzsystems positiv auf die Laufzeit des Monitors aus. Methodenaufrufe und an-

[8] CPU: Intel Core2 Duo, P8600, 2.40 GHz; Arbeitsspeicher: 8 GB RAM; Betriebssystem: Windows 7 Professional 64 Bit

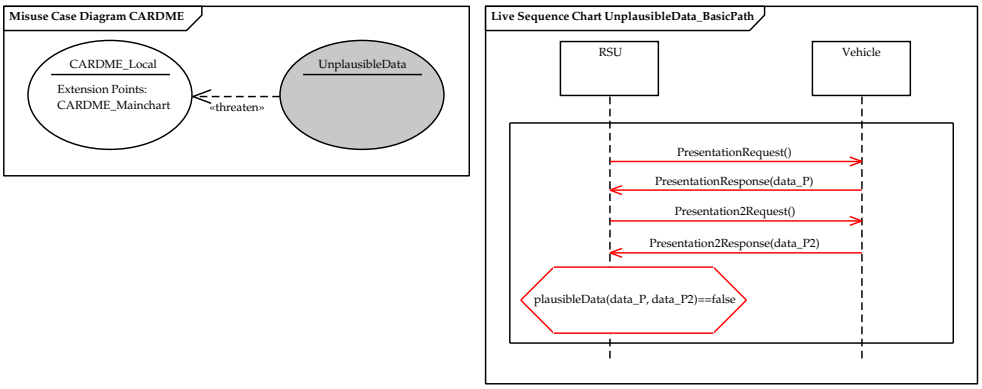


Abbildung 9.31: Signatur mit Use- und Misusecase mit Referenzsystem

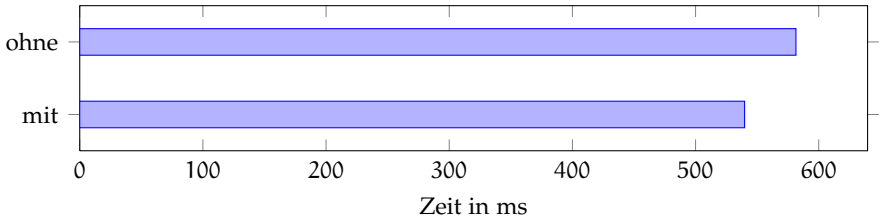


Abbildung 9.32: Laufzeit des CARDME-Monitors mit und ohne das Referenzsystem der MPNs

dere Aktionen in den Bedingungen werden in beiden Monitoren ausgewertet. Die gemessene Laufzeitersparnis des Monitors mit Referenzsystem entsteht aus der reduzierten Auswertung der Ereignisse des Misusecases in der Vorbedingung. In diesem Beispiel werden die Neuinitialisierungen des Monitors nicht verringert, da in beiden Fällen die in der Signatur *UnplausibleData* nicht enthaltene Ereignisse bei der Evaluation ignoriert werden. Diese Ersparnis würde bei einem Monitor, der die Ereignisse vor der Auswertung nicht filtert, hinzukommen. Das Referenzsystem selbst führt einen Laufzeitoverhead ein, der den Laufzeitgewinn wiederum einschränkt, wobei sich in diesem Beispiel die Laufzeit reduziert.

Abbildung 9.33 zeigt den Speicherbedarf des Codes der beiden Monitorversionen mit und ohne das Referenzsystem. Dieser nimmt durch die kleinere Signatur *UnplausibleData* beim Monitor mit Referenzsystem ab. Die Ersparnis durch die Reduzierung der Auswertungsmethoden für Transitionen des MPNs ist größer

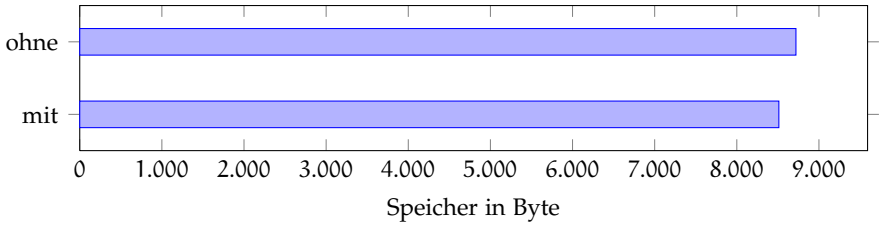


Abbildung 9.33: Codespeicherverbrauch des CARDME-Monitors mit und ohne das Referenzsystem der MPNs

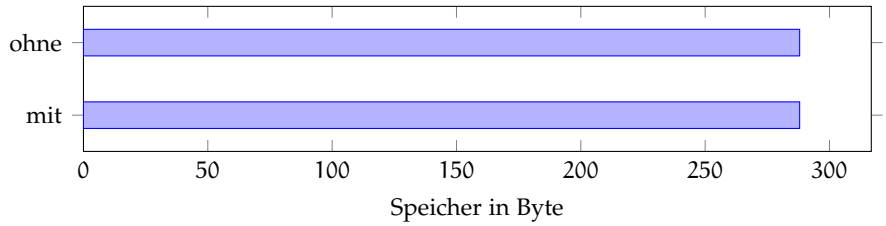


Abbildung 9.34: Datenspeicherverbrauch des CARDME-Monitors mit und ohne das Referenzsystem der MPNs

als die eingeführte Auswertungslogik des Referenzsystems. Beim Datenspeicherverbrauch, der in Abbildung 9.34 zu sehen ist, fällt die Reduzierung der Größe der Signatur nicht auf, da die Speicherung der Belegung der Plätze eine untergeordnete Rolle spielt. Hinzu kommen durch das Referenzsystem zum Beispiel Variablen, die anzeigen, ob eine Signaturinstanz aktiviert ist. Im Beispiel bleibt der Speicherverbrauch gleich.

SCHLUSSFOLGERUNG In dieser Evaluation wurde an einem kleinen Beispiel gezeigt, dass sich das Referenzsystem der MPN-Sprache bei der Laufzeit des Monitors positiv bemerkbar macht und damit einsetzbar ist. Auch der Speicherverbrauch lässt sich durch den Einsatz des Referenzsystems reduzieren.

Teilweise ist es bei der ganzheitlichen Modellierung von Use- und Misusecases schwer eine passende und dennoch kompakte Vorbedingung zu formulieren. Hierzu bietet z. B. die MBSecMonSL die MUC-Sprache an, um die Abhängigkeiten zwischen Signaturen zu beschreiben.

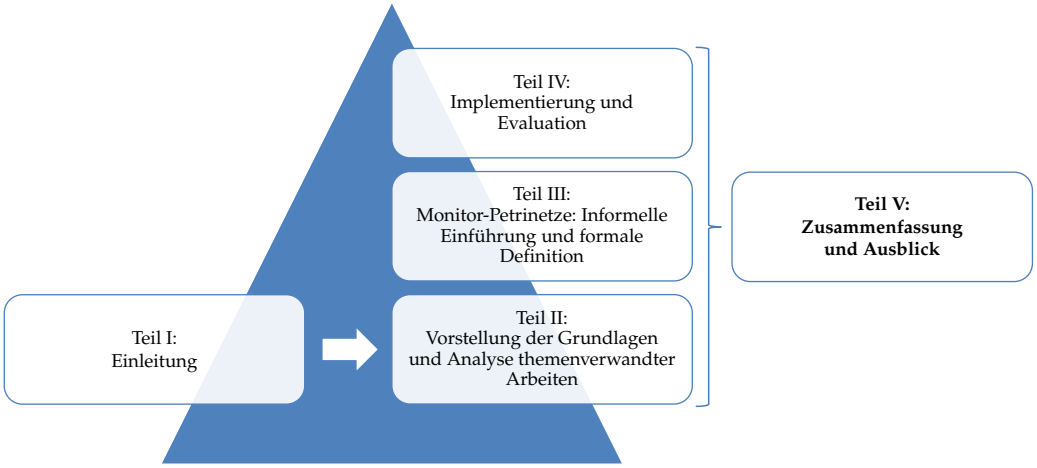
Bei der Übersetzung der Signaturen aus einer Spezifikationssprache, die Konzepte wie die MUC-Sprache unterstützt, in die MPN-Sprache müsste ohne das Referenzsystem eine Erzeugung von passenden Vorbedingungen automatisiert durchgeführt werden. Dieser Schritt wäre sehr komplex. Hier hilft das Referenzsystem Abhängigkeiten zwischen Signaturen in der Spezifikationssprache direkt in die MPN-Zwischensprache zu übernehmen, ohne komplexe Analysen auf der Quellspezifikation ausführen zu müssen.

Des Weiteren wird der Nachteil der Redundanz in der Überwachung, der in Abschnitt 4.2 betrachteten Ansätze im Bereich der IDS, die Signaturen in einzelne Programmmodule übersetzen, durch das Referenzsystem der MPN-Sprache abgemildert. Dabei ist jedoch zu beachten, dass das Referenzsystem an sich auch einen gewissen Overhead einführt und somit in Extremfällen (kleine und relativ unabhängige Teilsignaturen) zu Verschlechterungen der Laufzeit und des Speicherbedarfs führen kann.

In diesem Beispiel wurde nur ein relativ kleiner Anteil des Referenzsystems, das die Repräsentation von «threaten»- und «mitigate»-Beziehungen unterstützt, evaluiert. Eine Evaluation der «include»- und «extend»-Beziehungen aus der MBSecMonSL muss in Zukunft noch ausgeführt werden.

Teil V

ZUSAMMENFASSUNG UND AUSBLICK



10

ZUSAMMENFASSUNG

Ziel dieser Arbeit ist es einen Formalismus zu entwickeln, der sich als universelle Zwischensprache für die Spezifikation von Monitoren für hoch verschränkte Kommunikationsabläufe eignet. Dieser soll in der Domäne der Laufzeitüberwachung eingesetzt werden und spezielle Konzepte, die in diesem Gebiet benötigt werden, bieten.

Zusammengefasst sind die Beiträge dieser Arbeit:

- Die Monitor-Petrinetze als ein neuer Formalismus mit deterministischer Semantik, der als Zwischensprache zur Repräsentation von Signaturen hoch nebenläufiger Abläufe dient.
- Die Einbettung dieses MPN-Formalismus in einen durchgängigen Entwicklungsprozess für Laufzeitmonitore.
- Ein Konzept zur Generierung effizienter Monitore aus diesen MPNs.
- Die Implementierung des Prozesses als umfangreicher Prototyp.
- Die umfangreiche Evaluation des Prototypen in verschiedensten Einsatzbereichen und für unterschiedliche Zielsprachen.

Die in dieser Arbeit entwickelten Monitor-Petrinetze (MPN) sind zur effizienten Überwachung stark verschränkter nebenläufiger Abläufe geeignet. Die MPNs sind als allgemeine Zwischensprache für Werkzeugketten zur Generierung von Laufzeitmonitoren entwickelt worden. Sie erfüllen einen Großteil der für eine Zwischensprache zur Monitorgenerierung in Abschnitt 3.5 aufgestellten Anforderungen und bieten somit eine auf das Monitoring angepasste formalisierte Sprache.

ANFORDERUNGEN AN DIE MPN-SPRACHE Tabelle 10.1 zeigt einen Überblick, inwieweit MPNs die gestellten Anforderungen erfüllen und durch welche Konzepte dies erreicht wird. In der rechten Spalte der Tabelle sind zusätzlich Verweise auf die Abschnitte in denen diese Anforderungen erfüllt und besprochen werden. Unter dem Punkt *AS Ausdrucksstärke* sind alle Anforderungen an eine Signatursprache, die in Abschnitt 3.5 beschrieben sind, zusammengefasst. Diese werden durch die MPN-Sprache vollständig erfüllt, wie durch die Abbildung der Signaturmodellierungssprache eLSC auf die MPNs in [Pat14] gezeigt wird.

Die MPN-Sprache ist vollständig formalisiert (AP1) und erlaubt hierdurch die Generierung sicherheitsrelevanter Monitore. Die Signaturen lassen sich durch ihre Formalisierung analysieren und die Generierung eines Monitors basiert auf der

Anforderung		MPN	Konzept	Abschn.
AS	Ausdrucksstärke	+		[Pat14]
AP1	Formale Spezifikation	+		5, 6
AP2	Deterministische Semantik	+	Mikro-/Makroschrittsemantik	5
AP3	Keine Zustandsraumexplosion	+	Basierend auf Petrinetzen, 1-beschränkt	9.6
AP4	Effizient ausführbar	+	Einfache Schaltsemantik	5.3.2, 9
AP5	Parallele Kommunikation	+	Eingabegenerationen	5.3
AP6	Verschränkte Überwachungsversuche	+	Subgenerationen	5.5
AP7	Einfache Generierung/Ausführung	+	Wenige syntaktische Elemente	5.3.1
AP8	Kompakte Repräsentation	+	Auswertung auf Basis von Markierung	5
AP9	Zeitaspekte	o	Über Aktionen und Bedingungen	6.4
AP10	Aufteilbarkeit	+	Durch Übersetzung und Parallelitätskonzept der Petrinetze	6.6, 9.3
AP11	Bedingte Überwachung	+	Antecedent-/Consequent-Plätze, Referenzsystem	6.2
AP12	Gegenmaßnahmen	+	Ausführbare MPNs (Referenzsystem)	6.5
AP13	Plattformunabhängigkeit	+	Plattformspezifische Informationen	8.4
AP14	Erlaubte u. verbotene Signaturen	+	Positive und negative MPNs	5
AP15	Deontische Bedingungen	+	Durch Übersetzung von eLSC gezeigt	[Pat14]
AP16	Ereignisklassen	-	-	
AP17	Lokale/globale Variablen	+	Variablenklassen (Aktionen und Bedingungen)	6.3

+ erfüllt; o teilweise erfüllt; - nicht erfüllt

Tabelle 10.1: Anforderungen an die Zwischensprache in Bezug auf Monitor-Petrinetze

fest vorgegebenen Syntax und Semantik. Des Weiteren ist die Formalisierung von Spezifikationssprachen durch die Abbildung auf die MPN-Sprache möglich.

Durch ihre deterministische Semantik (AP2) sind MPNs an die Domäne der Monitorbeschreibung angepasst, die von außen übergebene Eingaben auf Korrektheit überprüft. Nichtdeterministische Modellierungskonzepte würden teilweise zu einer kompakteren Darstellung der Signaturen führen. Jedoch würde es, wenn die Signaturen nicht wie üblich auf einen deterministischen Formalismus (z. B. FSM) abgebildet werden, die Codegenerierung erschweren und die Komplexität des daraus resultierenden Monitors erhöhen.

Durch die Codierung von Zuständen in Kombinationen von Platzbelegungen (Markierungen) ist die modellierte Signatur in MPN relativ kompakt. Die dedizierten Start- und Endplätze sowie die impliziten Abbruchkriterien in der Semantik der MPNs führen im Gegensatz zu klassischen Petrinetzen zu kleineren Netzen. Zusätzlich führt die Begrenzung von nur einem Token auf einem Platz dazu, dass es nicht zu einer Zustandsraumexplosion (AP3) kommt.

Der als Zwischensprache verwendete Formalismus MPN muss, nachdem er in einen lauffähigen Monitor gewandelt wurde, effizient ausführbar sein (AP4). Die Evaluation hat gezeigt, dass aus MPNs effiziente Monitore generiert werden können.

Zur effizienten Überwachung einer hohen Anzahl paralleler Kommunikationen desselben oder eines ähnlichen Typs (AP5) dienen in den MPNs die Eingabegenerationen. In MPNs sind die Token mit einem Identifizierer markiert, der die

Zuordnung von äußeren Kommunikationssträngen zu internen Markierungen erlaubt. In der praktischen Implementierung können diese Identifizierer natürliche Zahlen oder komplexe Objekte sein. Hierzu wird eine Abbildung auf eine interne Eingabegeneration vorgenommen. Eine Instanziierung eines weiteren Monitors für einen neuen Kommunikationsstrang kann hierdurch allein durch eine neue Markierung dargestellt werden.

Das Konzept der Eingabegenerationen ist jedoch für die Laufzeitüberwachung nicht ausreichend. Eine verschränkte (*engl. concurrent*) Überwachung einer Signatur (AP6), ist notwendig, wenn der Beginn der Signatur mit dem ersten Ereignis nicht eindeutig zu erkennen ist. Hierbei kann es vorkommen, dass eine Signatur noch in der Vorbedingung ist, während der eigentliche Anfang der Signatur auftritt. Hierzu bieten die MPNs das Konzept der Subgenerationen. Diese bietet eine verschränkte Überwachung der Signatur einer Eingabegeneration, die während der Überwachung einer Signaturinstanz eine neue Instanz überwacht. Die Reihenfolge der Verarbeitung der Subgenerationen basiert auf deren Erstellungszeitpunkt und wird intern im MPN verwaltet.

Allgemein bedeutet im MPN-Konzept ein nicht erfolgreicher Überwachungsversuch einer Signatur nur das Zurücksetzen der Markierung für die entsprechende Eingabe- bzw. Subgeneration.

Eine einfache Generierung von Monitoren (AP7) ist durch die geringe Anzahl aktiver und passiver Elemente der Sprache gegeben. Ein MPN besteht wie ein Petrinetz aus Plätzen und Transitionen, die miteinander über Kanten verknüpft sind. Hierdurch sind nur das Schalten von Transitionen und die Auswertung der Belegung der speziellen Plätze im zu generierenden Monitor umzusetzen.

Diese implizite Auswertung auf Basis des Schaltens und der speziellen Plätze erlaubt im Gegensatz zu Petrinetzen, eine Reduktion der zu modellierenden Elemente. Es müssen nicht alle möglichen Enden einer Signatur modelliert werden, wenn diese durch die Semantik schon gegeben sind. Dies führt zusammen mit der Codierung des Zustandes als Markierung des Netzes zu einer kompakten Repräsentation (AP8) der Signatur.

Zeitaspekte (AP9) sind bisher über den Einsatz von Aktionen zum Speichern von Zeitpunkten und das Starten bzw. Stoppen von Timern sowie Bedingungen auf Zeitvariablen und aktuellen Zeitpunkten modellierbar. Eine externe Zeit kann über Epsilonereignisse dem Monitor diskret übergeben werden.

In verteilten Systemen in denen lokale Monitore für verschiedene Steuergeräte erzeugt werden sollen, ist es notwendig, auf Änderung der Verteilung der zu überwachenden Komponenten zur Generierungszeit zu reagieren. Signaturen, die die Kommunikation mehrerer Teilnehmer beschreiben, können, soweit dies in der Übersetzung der Spezifikationssprache betrachtet wurde, in Teilmonitore aufgeteilt (AP10) werden. Dies wird durch die Parallelität des Petrinetzkonzeptes in den MPNs unterstützt, indem die Kommunikation und die zu überwachenden Schritte der einzelnen Kommunikationspartner unabhängig voneinander, mit synchronisierenden Anteilen, modelliert werden können. Durch die Zuordnung in der Umsetzung der Transformation der Elemente der MPNs zu Kommunikationsinstanzen ist während der Codegenerierung durch Transformation der MPN-

Signatur eine Generierung von Teilsignaturen möglich. Für die Synchronisierung der Teilmonitore wurde ein erstes Konzept entworfen.

Zur effizienteren Überwachung bieten die MPNs Konzepte zur bedingten Überwachung (AP11) an. Dies ist die Unterscheidung von Vorbedingungen zu Hauptteilen der Signatur. In Überwachungssystemen werden häufig vollständige Einzelsignaturen, die alle unabhängig voneinander ausgewertet werden, überwacht. Hierdurch kommt es dazu, dass unnötigerweise ähnliche oder identische Teilsignaturen mehrfach ausgewertet werden. Zur Reduktion dieses Überwachungsaufwandes bietet das Referenzsystem der MPNs durch Markierung von Teilen (Plätzen) einer Basissignatur an, Bereiche in denen andere Signaturen überwacht werden sollen, zu spezifizieren.

Wenn ein Fehler oder Angriff von einem Monitor erkannt wurde, ist es sinnvoll Gegenmaßnahmen (AP12) ausführen zu können, die das System wieder in einen sicheren und stabilen Zustand bringen. Zusätzlich zu Aktionen an Transitionen in den eigentlichen Signaturen, können diese Gegenmaßnahmen im Referenzsystem durch ausführbare MPNs modelliert werden, die dieselbe Semantik wie jedes MPN besitzen. Eine Unterscheidung verschiedener Gegenmaßnahmen kann durch die Markierung des fehlgeschlagenen MPNs durchgeführt werden. Zusätzlich können in MPN-Signaturen durch Ereignisse und Bedingungen sowie die Unterscheidung zwischen Vorbedingung (*engl. antecedent*) und Hauptteil (*engl. consequent*) der Signatur, verschiedene Alternativen, die sich auf das weitere Verhalten des Systems beziehen, modelliert werden.

Die MPN-Sprache ist plattformunabhängig (AP13), da ihre Syntax und Semantik nur allgemeingültige Konzepte der Laufzeitüberwachung beinhaltet. Diese werden unter Verwendung zusätzlicher plattformspezifischer Informationen in ausführbare Monitore übersetzt. Die Annotationssprache für Aktionen und Bedingungen an den Transitionen ist nicht im Fokus dieser Arbeit. Hier kann jedoch eine beliebige in die Zielsprache übersetzbare Annotationssprache mit ausreichender Ausdrucksstärke gewählt werden.

Die Unterscheidung zwischen erlaubten und verbotenen Signaturen (AP14) findet bei den MPNs pro Signatur statt. Diese ist nicht nur implizit durch ihre Struktur mit entsprechenden Terminalplätzen gegeben, sondern wird für jede Signatur explizit festgelegt. Die implizite Auswertung des Ergebnisses beim Nichtschalten des MPNs verwendet diese Eigenschaft. Somit vereinfacht sich die Struktur des MPNs und damit auch dessen Generierung.

Deontische Bedingungen (AP15) erweitern die Ausdrucksstärke der klassischen Aussagenlogik mit den Operatoren Erlaubnis und Verpflichtung. Sie ist ein Spezialfall der Modallogik, die durch eLSCs unterstützt wird. Durch die Abbildung der eLSC-Konzepte auf die MPNs in [Pat14] ist gezeigt, dass auch MPNs diese Ausdrucksstärke besitzen.

Zur weiteren Abstraktion von Signaturen ist es wünschenswert Ereignisklassen und komplexe Ereignisse (AP16) einzuführen. Dieses Thema wurde in dieser Arbeit aufgrund seiner Komplexität ausgespart. Komplexe Ereignisse und Ereignisklassen können die Effizienz der Monitore, durch weniger ähnliche Signaturen, steigern.

Durch lokale und globale Variablen (AP17) können Werte, die durch Eingaben dem Monitor übergeben werden, für die weitere Überwachung zur Verwendung in Aktionen und Bedingungen zwischengespeichert werden. Die Unterscheidung zwischen verschiedenen Gültigkeitsbereichen, die in Abschnitt 6.3 beschrieben sind, schränkt deren Zugriff ein und verhindert hierdurch Fehler in den resultierenden Monitoren. So können Variablen zwischen Eingabegenerationen bzw. Subgenerationen geteilt werden oder spezifisch diesen zugewiesen werden.

ANALYSE VON MPN-SIGNATUREN: Spezielle Varianten der Petrinetze wie die MPNs können durch ihre stark angepasste Semantik nicht mehr mit den Standardmethoden auf besondere Eigenschaften wie Lebendigkeit, Beschränktheit und Erreichbarkeit geprüft werden. Jedoch können Standardtechniken der Place/Transition-Netze wie der Markierungsgraph bzw. Überdeckungsgraph herangezogen werden, um Eigenschaften wie die Anzahl der gleichzeitig möglichen Änderungen im Netz (Abschnitt 8.5) zu bestimmen. Des Weiteren erlauben sie auch die Optimierung der Spezifikationen durch die Analyse auf tote Transitionen (Abschnitt 7.3.2), die zwar nicht schalten können, jedoch in den generierten Monitoren zur Laufzeit überprüft werden. Diese Anzahl der Transitionen mit identischen Ereignissen in einem MPN ist wie in Kapitel 9 gezeigt zur Laufzeit eine kritische Größe, da die Laufzeit, die zur Verarbeitung eines solchen Ereignisses benötigt wird, mit der Anzahl an „identischen“ Transitionen linear ansteigt.

In Monitor-Petrinetzen sind im Gegensatz zu anderen Ansätzen wie JavaMOP, die Kontextfreie Grammatiken oder Endliche Automaten zur Spezifikation der Monitorsignaturen verwenden, monitoringspezifische Konzepte integriert. Hierdurch ist es nicht notwendig, diese in speziellen zusätzlichen Spezifikationssprachen zu beschreiben. Die Abbildung auf die Zielsprache und Zielplattform findet bei den MPNs im MBSecMon-Prozess erst direkt bei der Monitorgenerierung statt. Hierdurch können Monitore zielsystem- und zielsprachenunabhängig in einer beliebigen auf die MPNs abbildbaren Spezifikationssprache beschrieben und dann auf beliebigen Zielsystemen eingesetzt werden.

FAZIT Monitor-Petrinetze wurden als allgemeine Zwischensprache für Werkzeugketten zur Generierung von Monitoren entworfen und in einem Entwicklungsprozess für Monitore eingesetzt. Der MBSecMon-Prozess, der auf dem MDE-Ansatz aufbaut, verwendet die MPNs als Zwischensprache zur Repräsentation von Signaturen, die in der MBSecMonSL modelliert wurden.

Die Spezifikation der zu überwachenden Signaturen wird in einer ausdrucksstarken für die Modellierung von komplexen nebenläufig und verschränkten Zusammenhängen geeigneten Sprache, der MBSecMonSL, durchgeführt. Jedoch werden durch die neue Zwischensprache MPN vielfältige Spezifikationssprachen als Eingabe unterstützt, für die nur eine Übersetzung in die MPN-Sprache umgesetzt werden muss. Die Transformation der Spezifikationen, die in der MBSecMonSL modelliert wurden, in die formal definierten Monitor-Petrinetze, die eine explizitere Repräsentation der Signaturen bieten, ermöglicht die effiziente Monitorgenerierung bzw. Interpretation der Signaturen. Durch die Anpassung der MPNs vor der Codegenerierung an die Zielplattform und die Einbeziehung plattformspezi-

fischer Informationen, die auch aus vorhandenen Systemspezifikationen gewonnen werden, ist die Generierung von plattformspezifischen Monitoren erreicht worden. Es ist eine Vielzahl an Zielplattformen durch leichte Anpassungen der Generierung aus der MPN-Sprache unterstützbar.

EVALUATION Die Generierung von Monitoren aus der MPN-Sprache und deren Effizienz wurde in dieser Arbeit detailliert evaluiert. Hierbei wurden stellvertretend für verschiedene Zielsprachen und Zielplattformen Prototypen umgesetzt, die miteinander verglichen wurden. Als objektorientierte Sprache ist Java, als prozedurale Sprache C und als Hardwarebeschreibungssprache VHDL gewählt worden. Diese wurden auf verschiedenen Plattformen auf Laufzeit und Speicherbedarf evaluiert. Hierunter sind Java auf einem PC, C auf PC und Mikrocontroller und VHDL auf einem FPGA. Es hat sich gezeigt, dass die generierten Monitore insbesondere in der optimierten C-Variante Ereignissequenzen effizient zur Laufzeit überwachen können. Die Skalierbarkeit bei wachsender Komplexität der Spezifikation ist insbesondere bei Beschreibung verschränkter Abläufe durch den Petrinetzformalismus gegeben. Im Vergleich zu einem ähnlichen Ansatz, dem Monitor-oriented Programming (MOP), sind die Monitore in Bezug auf ihre Effizienz selbst für die von MOP unterstütze Untermenge von Konzepten des MBSecMon-Ansatzes zur Laufzeit gleichwertig, ohne dass die Java-Variante stark optimiert wurde. MOP ist bei der Überwachung von stark verschränkten Ereignissequenzen und parallelen Abläufen durch den exponentielle Speicherplatzkomplexität der verwendeten Zwischensprache FSM und kubischer Laufzeitkomplexität des CFG-Ansatzes auf eingebetteten Systemen nicht einsetzbar. Die Einbindung des Prozesses in AUTOSAR hat gezeigt, dass der Prozess und die Codegenerierung aus der MPN-Sprache an die Zielplattform mit Restriktionen in Bezug auf Schnittstellen und Datentypen gut anpassbar sind. Die generierten Monitore wurden automatisch an die zu überwachenden Systeme in Java über AspectJ und in C im AUTOSAR-Beispiel angebunden.

11

AUSBLICK

In dieser Arbeit wurden einige weiterführende Konzepte und Probleme nur initial betrachtet. Hier werden diese noch einmal benannt und weiterführende Forschungsaufgaben angesprochen.

Evaluation des Referenzsystems In der Evaluation wurde nur eine erste Umsetzung des Referenzsystems mit den Repräsentationen für «threaten»- und «mitigate»-Beziehungen der MBSecMonSL auf die Einsetzbarkeit in realen Monitoren betrachtet. Hier konnte festgestellt werden, dass sich durch das Konzept keine Verschlechterung der Laufzeit und des Speicherverbrauchs, sondern eine leichte Verbesserung einstellte. Es muss in Zukunft evaluiert werden, ob und inwieweit das vollständige Referenzsystem ohne große Leistungseinbußen, durch Verwaltung und Auswertung des Referenzsystems, auch auf eingebetteten Systemen nutzbar ist.

Erweiterung des Referenzsystems Bisher wurde in der Formalisierung des Referenzsystems die Einschränkung gemacht, dass ein als nicht-nebenläufig gekennzeichneter Erweiterungspunkt (EP) nachträglich nach dem Fehlschlagen (*failed*) des Basis-MPNs ausgeführt wird. Dies ist für die Umsetzung der eLSC-Sprache ausreichend. In Zukunft kann falls erforderlich dieses Konzept verallgemeinert werden. Der EP kann dann konfiguriert werden, dass er auf das Fehlschlagen (*failed*) oder auf das Beenden (*terminated*) reagiert. In Verbindung mit den Aktivierungsplätzen lassen sich hierdurch feingranulare Bedingungen für die Ausführung von Gegenmaßnahmen formulieren.

Aufteilung der Monitore In dieser Arbeit wurde die Aufteilung der Monitore nur oberflächlich betrachtet. Eine Kommunikation zwischen den Teilmonitoren führt einen hohen Laufzeitoverhead ein, der zu der Ereignisverarbeitungszeit hinzugerechnet werden muss. Des Weiteren muss sichergestellt werden, dass für die Übertragung der Synchronisationsnachrichten ein schneller oder priorisierter Kanal zur Verfügung steht, der die Reihenfolge der Synchronisationsnachricht und des folgenden Ereignisses sicherstellt.

Gleichzeitige Verarbeitung mehrerer Ereignisse Bisher wurde nur ein Ereignis auf einmal von einem MPN verarbeitet. Gleichzeitigkeit wurde durch eine vorher definierte Zeitspanne in der zwei Ereignisse auftreten definiert. Eine mögliche Erweiterung der MPN-Semantik ist die Verarbeitung mehrerer Ereignisse in einem Makroschritt. Hierbei würden die Ereignisse als gleichwertig be-

trachtet und alle Schaltvorgänge auf einer Markierung ausgeführt werden. Eine solches Konzept wäre z. B. bei der Überwachung eines Systems mit mehreren Sensoreingängen denkbar. Es ist zu evaluieren, ob eine solche Gleichzeitigkeit bei der Überwachung von realen Systemen benötigt wird, oder, ob das vorhandene Konzept ausreichend ist.

EREIGNISKLASSEN Auch die Möglichkeiten komplexe Ereignisse zu definieren wurde in der Arbeit nur angerissen. Für die Modellierung von Mustern kann die Zusammenfassung mehrerer aufeinanderfolgender Ereignisse zu einem komplexen Ereignis eine Abstraktion der Signaturen ermöglichen, die hierdurch kompakter ausfallen. Diese können mit einer zusätzlichen Sprache definiert werden oder durch MPNs, die als Aktion ein komplexes Ereignis dem Monitor übergeben. Des Weiteren sind auch übergeordnete Ereignistypen, die mehrere Ereignistypen unter sich vereinigen, denkbar. Diese Konzepte wurden in der Formalisierung nur grundlegend behandelt und in der prototypischen Implementierung nur für das AUTOSAR-Beispiel zur Abbildung von internen Ereignissen zu externen Schnittstellen umgesetzt. Ein MPN kann momentan keine Ereignisse dem Monitor über eine Aktion übergeben. Dies könnte zu Seiteneffekten führen, die in Zukunft betrachtet werden müssen.

EINGABESPRACHEN In Zukunft könnten andere Eingabesprachen wie Endliche Automaten durch die Übersetzung in MPNs für die Monitorgenerierung unterstützt werden. Diese Übersetzung ist um einiges einfacher als die Übersetzung der komplexen LSCs, da es sich bei FSMs um eine Untermenge klassischer Petrinetze handelt. Des Weiteren ist zu untersuchen, inwieweit Sprachen wie Temporale Logiken und die EDL effizient in MPN abgebildet werden können.

ZEIT IN DER MPN-FORMALISIERUNG Zeitaspekte werden in der in dieser Arbeit vorgestellten Form der MPNs als Aktionen und Bedingungen an den Transitionen modelliert. Zukünftig ist es wünschenswert hierfür eine eigene Sprache ähnlich zu den im MARTE-Profil [OMG08] definierten Erweiterungen der UML2 in die MPNs formal einzuführen. Zeitpunkte, an denen Ereignisse auftreten, könnten während der Laufzeit, wie es das MARTE-Profil der OMG vorschlägt, gespeichert werden und für die Auswertung an den Transitionen automatisch zur Verfügung stehen. Hierbei wäre im Gegensatz zum MARTE-Profil zur Reduzierung des Speicherbedarfs der Monitore eine Einschränkung auf den letzten Zeitpunkt des Auftretens eines Ereignisses ausreichend, da das Variablenkonzept der MPNs falls notwendig eine weitergehende Speicherung der Zeitpunkte erlaubt.

OPTIMIERUNG DER CODEGENERIERUNG Viele Transitionen in den MPNs mit „identischen“ Ereignissen in einer Signatur führen, wie in der Evaluation festgestellt, zu einem Anstieg der Laufzeit des Monitors. Auch wenn dies in den betrachteten nicht künstlich generierten Signaturen nicht in erhöhtem Maße vorkommt, bieten solche Signaturen Optimierungsmöglichkeiten. Die Auswertung welche Transitionen schalten, könnte nicht auf Basis des an den Transitionen annotierten Ereignisses, sondern an den aktuell aktivierten Transitionen (Vorbereich

belegt), durchgeführt werden. Hierzu müssten Informationen über die aktuell aktivierten Transitionen vorgehalten werden. Hierdurch könnte die Verarbeitungszeit der einzelnen Ereignisse auf Kosten von weiterem Speicherverbrauch (RAM) reduziert werden. Das Aktualisieren der gerade aktivierten Transitionen kostet jedoch auch Rechenzeit. Es muss evaluiert werden, in welchen Situationen Vorteile zu erwarten sind.

ÜBERWACHUNG SICH ÄNDERNDER STRUKTURELLER EIGENSCHAFTEN EINES SYSTEMS Neben Kommunikationen und Programmabläufen können auch strukturelle Eigenschaften eines Systems überwacht werden. Dies kann, wie in [Dec13] vorgeschlagen, durch Mustersuche (Pattern-Matching) auf einer Graphstruktur, die das System repräsentiert, umgesetzt werden. Hierbei werden Temporale Logiken zur Modellierung von zeitlichen Abfolgen vorgeschlagen. Durch eine erweiterte Spezifikationssprache, die zeitliche Abfolgen von gefundenen Graphmustern und anderen Ereignissen kombiniert, kann die MPN-Sprache ebenfalls als Zwischensprache zur Monitorgenerierung eingesetzt werden. Es müssen für gefundene und verschwundene Graphmuster Ereignisse generiert werden. Erlaubte und verbotene Ereignisabfolgen und zeitliche Aspekte lassen sich dann durch MPNs beschreiben und durch generierte Monitore überwachen.

PROAKTIVE ÜBERWACHUNG Die hier umgesetzten Prototypen überwachen ein System und reagieren im Fall einer Verletzung der Signaturen mit einer Aktion, die in einem aktiven MPN modelliert ist. Hierbei handelt es sich um einen reaktiven Ansatz. Eine weitere Möglichkeit ist die proaktive Überwachung wie sie in Abschnitt 4.3.2 für Edit Automata vorgestellt wurde. Auch hierzu eignen sich die Signaturen in der MPN-Sprache. Es können Monitore als Wrapper generiert werden, die erst die durch das System gesendete Nachricht oder aufgerufene Operation auswerten und dann, auf Basis des Ergebnisses, diese Aktion zulassen, blockieren oder mit einer erlaubten ersetzten.

POLICY ENFORCEMENT Bei der Einführung der ausführbaren MPNs muss darauf geachtet werden, dass die Monitore und das System in einem konsistenten Zustand gebracht werden. Zusätzlich zu verschiedenen Entscheidungen, die als Bedingung im ausführbaren MPN modelliert werden können, ist es sinnvoll bei mehreren möglichen Eingriffsmöglichkeiten, die Monitore testweise mit den entsprechenden Ereignissen der Gegenmaßnahme zu stimulieren und zu überprüfen, ob hierdurch keine anderen Monitore fehlschlagen. Erst wenn dies gegeben ist, kann das System tatsächlich mit der ausführbaren Signatur stimuliert werden. Momentan unterstützt die MPN-Sprache über das Referenzsystem mehrere Alternativen mit unterschiedlichen Vorbedingungen als Gegenmaßnahme zu spezifizieren.

EINSATZ DER MPN-SPRACHE ZUM TESTEN Spezifikationssprachen wie die im MBSecMon-Prozess eingesetzten eLSCs werden in der Praxis auch zur Testbeschreibung eingesetzt. Die in dieser Arbeit generierten Monitore und die MPN-Sprache könnten ebenfalls zur Überwachung der Testausführung als Testorakel

genutzt werden. Hierbei würde der Monitor parallel zum System laufen und Ein- und Ausgaben überwachen. Interne Signale könnten über Instrumentierungstechniken nach außen geführt werden. Um MPNs auch als Testtreiber (Eingaben in das System) nutzen zu können, müssten diese Eingaben als Aktionen im MPN modelliert werden.

TESTFALLGENERIERUNG AUS MPN-SIGNATUREN Die Signaturen für Laufzeitmonitore in der MPN-Sprache beschreiben positive und negative Abläufe des Systems. Positive Signaturen sind somit eine Spezifikation des Normalverhaltens des Systems. Beschreiben sie dabei das Kommunikationsverhalten des Systems an Schnittstellen, können sie als Quelle für eine Testfallgenerierung dienen. Ein Überdeckungskriterium, das die zu erreichende Abdeckung definiert, kann auf den MPNs definiert werden. Ähnlich wie im Ansatz in [WWH08] können negative MPNs als Test für im System implementierte Gegenmaßnahmen eingesetzt werden.

LITERATURVERZEICHNIS

[AAC+05] Allan, Chris; Avgustinov, Pavel; Christensen, Aske S.; Hendren, Laurie; Kuzins, Sascha; Lhoták, Ondřej; Moor, Oege de; Sereni, Damien; Sittampalam, Ganesh; Tibble, Julian: Adding Trace Matching with Free Variables to AspectJ. In: *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, ACM, 2005, S. 345–364 (Zitiert auf Seite 73.)

[AKRS06] Amelunxen, Carsten; Königs, Alexander; Rötschke, Tobias; Schürr, Andy: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. Version: 2006. http://dx.doi.org/10.1007/11787044_27. In: Rensink, Arend; Warmer, Jos (Hrsg.): *Second European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006)* Bd. 4066. DOI. – 10.1007/1178704 4_27. – ISBN 978–3–540–35909–8, S. 361–375 (Zitiert auf Seite 171.)

[Ale03] Alexander, Ian F.: Misuse Cases Help to Elicit Non-Functional Requirements. In: *Computing & Control Engineering Journal* 14 (2003), Februar, Nr. 1, S. 40–45. – ISSN 0956–3385 (Zitiert auf Seite 56.)

[ALPS11] Anjorin, Anthony; Lauder, Marius; Patzina, Sven; Schürr, Andy: eMoflon: Leveraging EMF and Professional CASE Tools. In: *3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2011)*. Bonn : Gesellschaft für Informatik, 2011 (Lecture Notes in Informatics) (Zitiert auf Seite 155.)

[AUT11a] AUTOSAR: *Specification of Operating System (R4.0 Rev 3)*. Online. Version: 2011. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_OS.pdf (Zitiert auf Seite 74.)

[AUT11b] AUTOSAR: *Specification of Watchdog Manager (R4.0 Rev 3)*. Online. Version: 2011. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_WatchdogManager.pdf (Zitiert auf Seite 74.)

[BDL06] Basin, David A.; Doser, Jürgen; Lodderstedt, Torsten: Model Driven Security: From UML Models to Access Control Infrastructures. In: *ACM Transactions on Software Engineering and Methodology* 15 (2006), Nr. 1, S. 39–91. http://dx.doi.org/10.1145/1125808.1125810. – DOI 10.1145/1125808.1125810. – ISSN 1049–331X (Zitiert auf Seite 69.)

[BFMW01] Bartetzko, Detlef; Fischer, Clemens; Möller, Michael; Wehrheim, Heike: Jass – Java with Assertions. In: *Electronic Notes in Theoretical Computer Science* 55 (2001), Nr. 2, 103–117. http://dx.doi.org/10.1016/S1571-0661(04)00247-6. – DOI 10.1016/S1571–0661(04)00247–6 (Zitiert auf Seite 67 und 76.)

[BJ08] Bauer, Andreas; Jürjens, Jan: Security Protocols, Properties, and their Monitoring. In: *Proceedings of the 4th International Workshop on Software Engineering for Secure Systems (SecSE 2008)* ACM, 2008. – ISBN 978–1–60558–042–5, S. 33–40 (Zitiert auf Seite 63.)

[BJ10] Bauer, Andreas; Jürjens, Jan: Runtime Verification of Cryptographic Protocols. In: *Computers & Security* 29 (2010), Nr. 3, S. 315 – 330. http://dx.doi.org/10.1016/j.cose.2009.09.003. – DOI 10.1016/j.cose.2009.09.003. – ISSN 0167–4048 (Zitiert auf Seite 63, 68 und 69.)

[BLS05] Barnett, Mike; Leino, K. Rustan M.; Schulte, Wolfram: The Spec# Programming System: An Overview. Version: 2005. http://dx.doi.org/10.1007/978-3-540-30569-9_3. In: Barthe, Gilles; Burdy, Lilian; Huisman, Marieke; Lanet, Jean-Louis; Muntean, Traian (Hrsg.): *International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)* Bd. 3362. DOI. – 10.1007/978–3–540–30569– 9_3. – ISBN 978–3–540–24287–1, S. 49–69 (Zitiert auf Seite 67.)

[BLS06] Bauer, Andreas; Leucker, Martin; Schallhart, Christian: Monitoring of Real-Time Properties. Version: 2006. http://dx.doi.org/10.1007/11944836_25. In: Arun-Kumar, S.; Garg, Naveen (Hrsg.): *Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2006)* Bd. 4337. DOI. – 10.1007/1194483 6_25. – ISBN 978–3–540–49994–7, S. 260–272 (Zitiert auf Seite 63 und 68.)

[BMI97] BMI: Entwicklungsstandard für IT-Systeme des Bundes, Vorgehensmodell. In: *Allgemeiner Umdruck* 250 (1997), Juli, Nr. 250. http://v-modell.iabg.de/index.php?option=com_docman&task=cat_view&gid=16&Itemid=30 (Zitiert auf Seite 7.)

[BPP+11] Biedermann, Alexander; Piper, Thorsten; Patzina, Lars; Patzina, Sven; Huss, Sorin A.; Schürr, Andy; Suri, Neeraj: Enhancing FPGA Robustness via Generic Monitoring Cores. In: *1st International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2011)*, 2011, S. 379–386 (Zitiert auf Seite 5.)

[BRH10] Barringer, Howard; Rydeheard, David; Havelund, Klaus: Rule Systems for Run-time Monitoring: from Eagle to RuleR. In: *Journal of Logic and Computation* 20 (2010), Nr. 3, S. 675–706. http://dx.doi.org/10.1093/logcom/exn076. – DOI 10.1093/logcom/exn076 (Zitiert auf Seite 63.)

[BS05] Bontemps, Yves; Schobbens, Pierre-Yves: The Complexity of Live Sequence Charts. Version: 2005. http://dx.doi.org/10.1007/978-3-540-31982-5_23. In: Sassone, Vladimiro (Hrsg.): *Foundations of Software Science and Computational Structures (FOSSACS 2005)* Bd. 3441. DOI. – 10.1007/978–3–540–31982– 5_23. – ISBN 978–3–540–25388–4, S. 364–378 (Zitiert auf Seite 7.)

[BTZ05] BARTELT, Christian; TERNITÉ, Thomas; ZIEGER, Matthias: Modellbasierte Entwicklung mit dem V-Modell XT. In: *OBJEKTspektrum* 5 (2005), S. 53–64 (Zitiert auf Seite 7.)

[Büc66] BÜCHI, J. R.: Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic. Version: 1966. http://dx.doi.org/10.1016/S0049-237X(09)70564-6. In: ERNEST NAGEL, Patrick S.; TARSKI, Alfred (Hrsg.): *Logic, Methodology and Philosophy of Science Proceeding of the 1960 International Congress* Bd. 44. Elsevier, 1966. – DOI 10.1016/S0049–237X(09)70564–6, S. 1–11 (Zitiert auf Seite 64 und 65.)

[Bun12] BUNDESSTELLE FÜR INFORMATIONSTECHNIK (BIT): *V-Modell XT*. Online. http://www.cio.bund.de/DE/Architekturen-und-Standards/V-Modell-XT/vmodell_xt_node.html. Version: Mai 2012, Abruf: 04.01.2014 (Zitiert auf Seite 7.)

[CFB12a] COTARD, Sylvain; FAUCOU, Sébastien; BÉCHENNEC, Jean-Luc: A Dataflow Monitoring Service Based on Runtime Verification for AUTOSAR OS: Implementation and Performances. In: PARMER, Gabriel; BASTONI, Andrea (Hrsg.): *8th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2012)*, 2012, S. 46–55 (Zitiert auf Seite 75.)

[CFB+12b] COTARD, Sylvain; FAUCOU, Sébastien; BÉCHENNEC, Jean-Luc; QUEUDET, Audrey; TRINQUET, Yvon: A Data Flow Monitoring Service Based on Runtime Verification for AUTOSAR. In: *14th International Conference on High Performance Computing and Communication (HPCC-ICESS 2012)*, IEEE Computer Society, 2012, S. 1508–1515 (Zitiert auf Seite 75.)

[CKLP06] CHALIN, Patrice; KINIRY, Joseph R.; LEAVENS, Gary T.; POLL, Erik: Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. Version: 2006. http://dx.doi.org/10.1007/11804192_16. In: BOER, Frank S.; BONSANGUE, Marcello M.; GRAF, Susanne; ROEVER, Willem-Paul (Hrsg.): *Formal Methods for Components and Objects* Bd. 4111. DOI. – 10.1007/1180419 2_16. – ISBN 978–3–540–36749–9, S. 342–363 (Zitiert auf Seite 67.)

[DBS92] DEBAR, Hervé; BECKER, Monique; SIBONI, Didier: A Neural Network Component for an Intrusion Detection System. In: *IEEE Symposium on Research in Security and Privacy (RISP 1992)*, IEEE Computer Society, 1992, S. 240–250 (Zitiert auf Seite 58.)

[Dec13] DECKWERTH, Frederik: Generating Monitors for Usage Control. In: WAGNER, Stefan; LICHTER, Horst (Hrsg.): *Software Engineering 2013 - Workshopband* Bd. 215, Gesellschaft für Informatik, 2013 (LNI), S. 577–582 (Zitiert auf Seite 239.)

[Den87] DENNING, Dorothy E.: An Intrusion-Detection Model. In: *IEEE Transactions on Software Engineering* 13 (1987), Nr. 2, S. 222–232. http://dx.

doi.org/10.1109/TSE.1987.232894. – DOI 10.1109/TSE.1987.232894. – ISSN 0098–5589 (Zitiert auf Seite 58.)

[DH01] Damm, Werner; Harel, David: LSCs: Breathing Life into Message Sequence Charts. In: *Formal Methods in System Design* 19 (2001), Nr. 1, S. 45–80 (Zitiert auf Seite 30.)

[DR98] Desel, Jörg; Reisig, Wolfgang: Place/Transition Petri Nets. Version: 1998. http://dx.doi.org/10.1007/3-540-65306-6_15. In: Reisig, Wolfgang; Rozenberg, Grzegorz (Hrsg.): *Lectures on Petri Nets I: Basic Models* Bd. 1491. DOI. – 10.1007/3–540–65306– 6_15. – ISBN 978–3–540–65306–6, S. 122–173 (Zitiert auf Seite 37.)

[Dru00] Drusinsky, Doron: The Temporal Rover and the ATG Rover. In: Havelund, Klaus; Penix, John; Visser, Willem (Hrsg.): *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification* Bd. 1885, Springer, 2000 (LNCS). – ISBN 978–3–540–41030–0, S. 323–330 (Zitiert auf Seite 63.)

[Fah07] Fahland, Dirk: Synthesizing Petri Nets from LTL Specifications - An Engineering Approach. In: Philippi, Stephan; Pinl, Alexander (Hrsg.): *Tagungsband des 14. Workshop Algorithmen und Werkzeuge für Petrinetze (AWPN 2007), Arbeitsbericht aus dem Fach Informatik*, 2007, S. 69–74 (Zitiert auf Seite 77.)

[FGP09] Frankowiak, Marcos R.; Grosvenor, Roger I.; Prickett, Paul W.: Microcontroller-Based Process Monitoring Using Petri-Nets. In: *EURASIP Journal on Embedded Systems* 2009 (2009), S. 1–12. http://dx.doi.org/10.1155/2009/282708. – DOI 10.1155/2009/282708 (Zitiert auf Seite 66 und 76.)

[FHSL96] Forrest, Stephanie; Hofmeyr, Steven A.; Somayaji, Anil; Longstaff, Thomas A.: A Sense of Self for Unix Processes. In: *IEEE Symposium on Security and Privacy (SP 1996)*, IEEE Computer Society, 1996. – ISSN 1081–6011, S. 120–128 (Zitiert auf Seite 58.)

[FM12] Flegel, Ulrich; Meier, Michael: Modeling and Describing Misuse Scenarios Using Signature-Nets and Event Description Language. In: *it - Information Technology* 54 (2012), März, Nr. 2, S. 71–81. http://dx.doi.org/10.1524/itit.2012.0666. – DOI 10.1524/itit.2012.0666 (Zitiert auf Seite 60.)

[For82] Forgy, Charles L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In: *Artificial Intelligence* 19 (1982), Nr. 1, S. 17–37. http://dx.doi.org/10.1016/0004-3702(82)90020-0. – DOI 10.1016/0004–3702(82)90020–0 (Zitiert auf Seite 58.)

[GMS12] Gay, Richard; Mantel, Heiko; Sprick, Barbara: Service Automata. In: Barthe, Gilles; Datta, Anupam; Etalle, Sandro (Hrsg.): *Formal Aspects of Security and Trust* Bd. 7140, Springer, 2012 (LNCS). – ISBN 978–3–642–29419–8, S. 148–163 (Zitiert auf Seite 65.)

[GR98] GIARRATANO, Joseph C.; RILEY, Gary D.: *Expert Systems: Principles and Programming, Third Edition*. 3. Course Technology, 1998. – ISBN 978–0–534–95053–8 (Zitiert auf Seite 57, 58 und 76.)

[GR09] GROLL, André; RULAND, Christoph: Secure and Authentic Communication on Existing In-Vehicle Networks. In: *IEEE Intelligent Vehicles Symposium (IV 2009)*, IEEE Computer Society, Juni 2009, S. 1093–1097 (Zitiert auf Seite 4 und 10.)

[GV03] GIRAULT, Claude; VALK, Rüdiger: *Petri Nets for Systems Engineering*. Springer, 2003. – 607 S. – ISBN 978–3–540–41217–5. – ISBN: 3-540-41217-4 (Zitiert auf Seite 175.)

[Har87] HAREL, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987), Nr. 3, S. 231–274. http://dx.doi.org/10.1016/0167-6423(87)90035-9. – DOI 10.1016/0167–6423(87)90035–9 (Zitiert auf Seite 35.)

[HK02] HAREL, David; KUGLER, Hillel: Synthesizing State-Based Object Systems from LSC Specifications. In: *International Journal of Foundations of Computer Science* 13 (2002), Nr. 1, S. 5–51. DOI. – 10.1007/3–540–44674–5_1 (Zitiert auf Seite 97.)

[HKP05] HAREL, David; KUGLER, Hillel; PNUELI, Amir: Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In: *Formal Methods in Software and Systems Modeling, Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday* Bd. 3393, 2005. – ISBN 3–540–24936–2, 309–324 (Zitiert auf Seite 5.)

[HM03] HAREL, David; MARELLY, Rami: *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003. – 382 S. – ISBN 978–3–540–00787–6 (Zitiert auf Seite 30.)

[Hoa85] HOARE, C. A. R.: *Communicating sequential processes*. Prentice-Hall, Inc., 1985 http://www.usingcsp.com/cspbook.pdf. – ISBN 978–0–131–53271–7. – Version 2004 (Zitiert auf Seite 67.)

[HR01] HAVELUND, Klaus; ROSU, Grigore: Monitoring Java Programs with Java PathExplorer. In: *Electronic Notes in Theoretical Computer Science* 55 (2001), Nr. 2, S. 200–217. http://dx.doi.org/10.1016/S1571-0661(04)00253-1. – DOI 10.1016/S1571–0661(04)00253–1 (Zitiert auf Seite 63.)

[IEE06] IEEE Standard for Verilog Hardware Description Language. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), April, S. 0_1–560. http://dx.doi.org/10.1109/IEEESTD.2006.99495. – DOI 10.1109/IEEESTD.2006.99495. ISBN 0–7381–4851–2 (Zitiert auf Seite 27.)

[IEE09] IEEE Standard VHDL Language Reference Manual. In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (2009), Februar, S.

c1–626. http://dx.doi.org/10.1109/IEEESTD.2009.4772740. – DOI 10.1109/IEEESTD.2009.4772740. ISBN 978–0–7381–6853–1 (Zitiert auf Seite 27.)

[Ill12] Illy, Christian: *Entwicklung von Hardware-Monitoren – Umsetzung von Monitor-Petri-Netzen auf einem FPGA*, Technische Universität Darmstadt, Bachelorthesis, Jan. 2012 (Zitiert auf Seite 198.)

[IT00] ITU-T: *Recommendation Z.120: Message Sequence Chart (MSC)*. Geneva, 2000 (SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS) (Zitiert auf Seite 5 und 30.)

[Jen91] Jensen, Kurt: Coloured Petri Nets: A High Level Language for System Design and Analysis. In: Rozenberg, Grzegorz (Hrsg.): *Advances in Petri Nets 1990* Bd. 483, Springer, 1991 (LNCS), S. 342–416 (Zitiert auf Seite 60.)

[Jür02] Jürjens, Jan: UMLsec: Extending UML for Secure Systems Development. In: Jézéquel, Jean-Marc; Hussmann, Heinrich; Cook, Stephen (Hrsg.): *«UML» 2002 – The Unified Modeling Language (UML 2002)* Bd. 2460, Springer, 2002 (LNCS). – ISBN 3–540–44254–5, S. 412–425 (Zitiert auf Seite 68.)

[Jür05] Jürjens, Jan: *Secure Systems Development with UML*. Springer, 2005. – 316 S. – ISBN 978–3–540–00701–2 (Zitiert auf Seite 68.)

[KCR+10] Koscher, Karl; Czeskis, Alexei; Roesner, Franziska; Patel, Franziska; Kohno, Tadayoshi; Checkoway, Stephen; McCoy, Damon; Kantor, Brian; Anderson, Danny; Shacham, Hovav; Savage, Stefan: Experimental Security Analysis of a Modern Automobile. In: *IEEE Symposium on Security and Privacy (SP 2010)*, IEEE Computer Society, 2010, S. 447–462 (Zitiert auf Seite 10.)

[Keß12] Kessler, Jan: *Anpassung von Laufzeitmonitoren an ein verteiltes, multithreading-fähiges Echtzeitbetriebssystem*, Technische Universität Darmstadt, Studienarbeit, Nov. 2012 (Zitiert auf Seite 204.)

[KF09] Kindel, Olaf; Friedrich, Mario: *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. dpunkt, 2009. – ISBN 978–3–898–64563–8 (Zitiert auf Seite xv, 23, 24, 25 und 74.)

[KHB99] Karaorman, Murat; Hölzle, Urs; Bruno, John: jContractor: A Reflective Java Library to Support Design By Contract. Version: Juli 1999. http://dx.doi.org/10.1007/3-540-48443-4_18. In: Cointe, Pierre (Hrsg.): *Meta-Level Architectures and Reflection* Bd. 1616. DOI. – 10.1007/3–540–48443– 4_18. – ISBN 978–3–540–66280–8, S. 175–196 (Zitiert auf Seite 67.)

[KHH+01] Kiczales, Gregor; Hilsdale, Erik; Hugunin, Jim; Kersten, Mik; Palm, Jeffrey; Griswold, William G.: An Overview of AspectJ. In:

Knudsen, Jørgen L. (Hrsg.): *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, Springer, 2001 (LNCS). – ISBN 978–3–540–42206–8, S. 327–353 (Zitiert auf Seite 46.)

[KKL+01] Kim, Moonzoo; Kannan, Sampath; Lee, Insup; Sokolsky, Oleg; Viswanathan, Mahesh: Java-MaC: A Run-time Assurance Tool for Java Programs. In: *Electronic Notes in Theoretical Computer Science* 55 (2001), Nr. 2, S. 218 – 235. http://dx.doi.org/10.1016/S1571-0661(04)00254-3. – DOI 10.1016/S1571–0661(04)00254–3 (Zitiert auf Seite 63.)

[KLM+97] Kiczales, Gregor; Lamping, John; Mendhekar, Anurag; Maeda, Chris; Lopes, Cristina; Loingtier, Jean-Marc; Irwin, John: Aspect-Oriented Programming. Version: 1997. http://dx.doi.org/10.1007/BFb0053381. In: Aksit, Mehmet; Matsuoka, Satoshi (Hrsg.): *European Conference on Object-Oriented Programming (ECOOP 1997)* Bd. 1241. Springer, 1997. – DOI 10.1007/BFb0053381. – ISBN 978–3–540–63089–0, S. 220–242 (Zitiert auf Seite 46.)

[KM09] Kumar, Rahul; Mercer, Eric G.: Verifying Communication Protocols Using Live Sequence Chart Specifications. In: *Electronic Notes in Theoretical Computer Science* 250 (2009), Nr. 2, S. 33–48. http://dx.doi.org/10.1016/j.entcs.2009.08.016. – DOI 10.1016/j.entcs.2009.08.016 (Zitiert auf Seite 76.)

[Kum95] Kumar, Sandeep: *Classification and Detection of Computer Intrusions*, Purdue University, Diss., 1995 (Zitiert auf Seite 4, 58, 59, 66, 76 und 77.)

[Lan11] Langbein, Tobias: *Entwurf und Implementierung eines generischen Interpreters für Monitor-Petrinetze*, Technische Universität Darmstadt, Bachelor-Thesis, Januar 2011 (Zitiert auf Seite 171.)

[Lan13] Langbein, Tobias: *Entwicklung und Evaluation eines Rahmenwerks zur Generierung von Laufzeitmonitoren*, Technische Universität Darmstadt, Master-Thesis, April 2013 (Zitiert auf Seite 70, 174 und 209.)

[LBD02] Lodderstedt, Torsten; Basin, David; Doser, Jürgen: SecureUML: A UML-Based Modeling Language for Model-Driven Security. In: *«UML» 2002 – The Unified Modeling Language (UML 2002)* Bd. 2460, Springer, 2002, S. 426–441 (Zitiert auf Seite 69.)

[LBW05a] Ligatti, Jay; Bauer, Lujo; Walker, David: Edit Automata: Enforcement Mechanisms for Run-Time Security Policies. In: *International Journal of Information Security* 4 (2005), Nr. 1-2, S. 2–16. http://dx.doi.org/10.1007/s10207-004-0046-8. – DOI 10.1007/s10207–004–0046–8 (Zitiert auf Seite 65.)

[LBW05b] Ligatti, Jay; Bauer, Lujo; Walker, David: Enforcing Non-Safety Security Policies with Program Monitors. In: Vimercati, Sabrina de C.;

Syverson, Paul; Gollmann, Dieter (Hrsg.): *In 10th European Symposium on Research in Computer Security (ESORICS 2005)* Bd. 3679, Springer, 2005 (LNCS), S. 355–373 (Zitiert auf Seite 65.)

[LJ09] Lloyd, John; Jürjens, Jan: Security Analysis of a Biometric Authentication System Using UMLsec and JML. Version: 2009. http://dx.doi.org/10.1007/978-3-642-04425-0_7. In: Schürr, Andy; Selic, Bran (Hrsg.): *Model Driven Engineering Languages and Systems* Bd. 5795. DOI. – 10.1007/978–3–642–04425– 0_7. – ISBN 978–3–642–04424–3, S. 77–91 (Zitiert auf Seite 68.)

[LP99] Lindqvist, Ulf; Porras, Phillip A.: Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). In: *IEEE Symposium on Security and Privacy (SP 1999)* IEEE, 1999. – ISBN 0–769–50176–1, S. 146–161 (Zitiert auf Seite 57, 58 und 76.)

[MBH02] Meier, Michael; Bischof, Niels; Holz, Thomas: SHEDEL – A Simple Hierarchical Event Description Language for Specifying Attack Signatures. Version: 2002. http://dx.doi.org/10.1007/978-0-387-35586-3_44. In: Ghonaimy, M. A.; El-Hadidi, Mahmoud T.; Aslan, Heba K. (Hrsg.): *Security in the Information Society* Bd. 86. DOI. – 10.1007/978–0–387–35586– 3_44. – ISBN 978–1–4757–1026–7, S. 559–571 (Zitiert auf Seite 61.)

[Mei04] Meier, Michael: A Model for the Semantics of Attack Signatures in Misuse Detection Systems. In: Zhang, Kan; Zheng, Yuliang (Hrsg.): *7th International Conference on Information Security (ISC 2004)* Bd. 3225, Springer, 2004 (LNCS). – ISBN 978–3–540–23208–7, S. 158–169 (Zitiert auf Seite 29, 48 und 60.)

[Mey88] Meyer, Bertrand: *Object-Oriented Software Construction*. Prentice Hall New York, 1988. – ISBN 0–136–29049–3 (Zitiert auf Seite 66.)

[MFMP06] Mellado, Daniel; Fernández-Medina, Eduardo; Piattini, Mario: A Comparative Study of Proposals for Establishing Security Requirements for the Development of Secure Information Systems. Version: 2006. http://dx.doi.org/10.1007/11751595_109. In: Gavrilova, Marina; Gervasi, Osvaldo; Kumar, Vipin; Tan, C. J. K.; Taniar, David; Laganá, Antonio; Mun, Youngsong; Choo, Hyunseung (Hrsg.): *Computational Science and Its Applications (ICCSA 2006)* Bd. 3982. DOI. – 10.1007/1175159 5_109. – ISBN 978–3–540–34075–1, S. 1044–1053 (Zitiert auf Seite 9.)

[MJCR08] Meredith, Patrick; Jin, Dongyun; Chen, Feng; Roşu, Grigore: Efficient Monitoring of Parametric Context-Free Patterns. In: *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, IEEE/ACM, 2008. – ISBN 978–1–424–42188–6, S. 148–157 (Zitiert auf Seite 71.)

[MJG+12] MEREDITH, Patrick O.; JIN, Dongyun; GRIFFITH, Dennis; CHEN, Feng; ROŞU, Grigore: An Overview of the MOP Runtime Verification Framework. In: *International Journal on Software Tools for Technology Transfer* 14 (2012), Nr. 3, S. 249–289. http://dx.doi.org/10.1007/s10009-011-0198-6. – DOI 10.1007/s10009–011–0198–6 (Zitiert auf Seite 5, 6, 64, 70, 76 und 209.)

[MJH+10] MONTRIEUX, Lionel; JÜRJENS, Jan; HALEY, Charles B.; YU, Yijun; SCHOBBENS, Pierre-Yves; TOUSSAINT, Hubert: Tool support for code generation from a UMLsec property. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2010 (ASE '10). – ISBN 978–1–450–30116–9, S. 357–358 (Zitiert auf Seite 68.)

[MS05] MEIER, Michael; SCHMERL, Sebastian: Effiziente Analyseverfahren für Intrusion-Detection-Systeme. In: FEDERRATH, Hannes (Hrsg.): *Sicherheit 2005: Sicherheit - Schutz und Zuverlässigkeit, Beiträge der 2. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.v. (GI)* Bd. 62, GI, 2005 (LNI). – ISBN 3–885–79391–1, S. 209–220 (Zitiert auf Seite 60 und 76.)

[Mur89] MURATA, Tadao: Petri Nets: Properties, Analysis and Applications. In: *Proceedings of the IEEE* 77 (1989), April, Nr. 4, S. 541–580. http://dx.doi.org/10.1109/5.24143. – DOI 10.1109/5.24143. – ISSN 0018–9219 (Zitiert auf Seite 187.)

[MZ88] MURATA, Tadao; ZHANG, Du: A Predicate-Transition Net Model for Parallel Interpretation of Logic Programs. In: *IEEE Transactions on Software Engineering* Bd. 14, 1988, S. 481–497 (Zitiert auf Seite 36.)

[Oeh02] OEHRY, Bernhard: *The CARDME Concept*. Online. www.uv.es/fsoriano/efc/cardme.pdf. Version: June 2002 (Zitiert auf Seite xv, 17 und 18.)

[OMG06] OMG: *OMG Meta Object Facility (MOF) Core Specification*. online. Version: Januar 2006. http://www.omg.org/spec/MOF/2.0/ (Zitiert auf Seite 43.)

[OMG08] OMG: *A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems*. online. Version: Juni 2008. http://www.omg.org/docs/ptc/08-06-08.pdf (Zitiert auf Seite 238.)

[OMG11] OMG: *OMG Unified Modeling LanguageTM (OMG UML), Superstructure*. online. Version: August 2011. http://www.omg.org/spec/UML/2.4.1/Superstructure (Zitiert auf Seite 34 und 35.)

[OSM02] OH, Nahmsuk; SHIRVANI, Philip P.; MCCLUSKEY, Edward J.: Control-flow Checking by Software Signatures. In: *IEEE Transactions on Reliability* 51 (2002), Nr. 1, S. 111–122. http://dx.doi.org/10.1109/24.994926. – DOI 10.1109/24.994926. – ISSN 0018–9529 (Zitiert auf Seite 75.)

[Pat14] PATZINA, Sven: *Entwicklung einer Spezifikationssprache zur modellbasierten Generierung von Security-/Safety-Monitoren zur Absicherung von (Eingebetteten) Systemen*, Technische Universität Darmstadt, Diss., 2014. http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:tuda-tuprints-41327. – URN urn:nbn:de:tuda–tuprints–41327 (Zitiert auf Seite v, vii, 7, 9, 12, 29, 30, 31, 49, 55, 155, 216, 231, 232 und 234.)

[Pet62] PETRI, Carl A.: Fundamentals of a Theory of Asynchronous Information Flow. In: *IFIP Congress (Information Processing 1962)*, North Holland Publ. Comp., 1962, S. 386–390 (Zitiert auf Seite 36.)

[Phi06] PHILIPPI, Stephan: Automatic Code Generation from High-Level Petri-Nets for Model Driven Systems Engineering. In: *Journal of Systems and Software - Architecting Dependable Systems* 79 (2006), Oktober, Nr. 10, S. 1444–1455. http://dx.doi.org/10.1016/j.jss.2005.12.022. – DOI 10.1016/j.jss.2005.12.022. – ISSN 0164–1212 (Zitiert auf Seite 174 und 175.)

[PPPM13] PATZINA, Lars; PATZINA, Sven; PIPER, Thorsten; MANNS, Paul: Model-Based Generation of Run-Time Monitors for AUTOSAR. Version: 2013. http://dx.doi.org/10.1007/978-3-642-39013-5_6. In: GORP, Pieter; RITTER, Tom; ROSE, Louis M. (Hrsg.): *Modelling Foundations and Applications (ECMFA 2013)* Bd. 7949. DOI. – 10.1007/978–3–642–39013– 5_6. – ISBN 978–3–642–39012–8, S. 70–85. – Best Paper Award (Zitiert auf Seite 74 und 215.)

[PPPS10] PATZINA, Lars; PATZINA, Sven; PIPER, Thorsten; SCHÜRR, Andy: Monitor Petri Nets for Security Monitoring. In: *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems (S&D4RCES 2010)*, ACM, 2010 (S&D4RCES). – ISBN 978–1–4503–0368–2, S. 3:1–3:6 (Zitiert auf Seite 81.)

[PPS11] PATZINA, Sven; PATZINA, Lars; SCHÜRR, Andy: Extending LSCs for Behavioral Signature Modeling. In: CAMENISCH, Jan; FISCHER-HÜBNER, Simone; MURAYAMA, Yuko; PORTMANN, Armand; RIEDER, Carlos (Hrsg.): *Future Challenges in Security and Privacy for Academia and Industry (IFIP SEC 2011)* Bd. 354, 2011 (IFIP Advances in Information and Communication Technology). – ISBN 978–3–642–21423–3, S. 293–304 (Zitiert auf Seite 48.)

[PWMS12] PIPER, Thorsten; WINTER, Stefan; MANN, Paul B.; SURI, Neeraj: Instrumenting AUTOSAR for Dependability Assessment: A Guidance Framework. In: *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)* IEEE, 2012, S. 1–12 (Zitiert auf Seite 217.)

[RC12] ROŞU, Grigore; CHEN, Feng: Semantics and Algorithms for Parametric Monitoring. In: *Logical Methods in Computer Science* 8 (2012), Nr. 1, S. 1–47 (Zitiert auf Seite 73.)

[Rei10] REISIG, Wolfgang: *Petrinetze - Modellierungstechnik, Analysemethoden, Fallstudien*. 1. Vieweg+Teubner Verlag, 2010 (XLeitfäden der Informatik). – 248 S. – ISBN 978–3–834–81290–2 (Zitiert auf Seite 38.)

[RQJZ07] RUPP, Chris; QUEINS, Stefan; JECKLE, Mario; ZENGLER, Barbara: *UML2 glasklar: Praxiswissen für die UML-Modellierung*. 3. Hanser Verlag, 2007. – 554 S. – ISBN 978–3–446–41118–0 (Zitiert auf Seite 29 und 55.)

[Sam08] SAMEK, Miro: *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. 2. Newnes, 2008. – 728 S. – ISBN 978–0–750–68706–5 (Zitiert auf Seite 34 und 175.)

[SBPM08] STEINBERG, Dave; BUDINSKY, Frank; PATERNOSTRO, Marcelo; MERKS, Ed: *EMF: Eclipse Modeling Framework (2nd Edition)*. 2nd Revised. Addison-Wesley Professional, 2008. – ISBN 978–0–321–33188–5 (Zitiert auf Seite 44.)

[Sch00] SCHNEIDER, Fred B.: Enforceable Security Policies. In: *ACM Transactions on Information and System Security (TISSEC)* 3 (2000), Februar, Nr. 1, S. 30–50. http://dx.doi.org/10.1145/353323.353382. – DOI 10.1145/353323.353382 (Zitiert auf Seite 65.)

[Sch04] SCHMERL, Sebastian: *Entwurf und Entwicklung einer Effizienten Analyseeinheit für Intrusion-Detection-Systeme*, Lehrstuhl Rechnernetze, BTU Cottbus, Diplomarbeit, Oktober 2004 (Zitiert auf Seite 48.)

[SEJK08] SMITH, Randy; ESTAN, Cristian; JHA, Somesh; KONG, Shijin: Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In: *SIGCOMM Computer Communication Review* 38 (2008), August, Nr. 4, S. 207–218. http://dx.doi.org/10.1145/1402946.1402983. – DOI 10.1145/1402946.1402983 (Zitiert auf Seite 77.)

[SES07] SOLHAUG, Bjornar; ELGESEM, Dag; STOLEN, Ketil: Specifying Policies Using UML Sequence Diagrams–An Evaluation Based on a Case Study. In: *8th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2007)*, IEEE Computer Society, 2007. – ISBN 0–7695–2767–1, S. 19–28 (Zitiert auf Seite 53.)

[SO01] SINDRE, Guttorm; OPDAHL, Andreas L.: Capturing Security Requirements through Misuse Cases. In: *Norsk Informatikkonferanse 2001 (NIK 2001)*, 2001 (Zitiert auf Seite 56.)

[SP05] *Kapitel* 14. Implementing a Petri Net Specification in a FPGA Using VHDL. In: SOTO, Enrique; PEREIRA, Miguel: *Design of Embedded Control Systems*. Springer, 2005. – ISBN 978–0–387–23630–8, S. 167–174 (Zitiert auf Seite 198.)

[Var01] VARDI, Moshe Y.: Branching vs. Linear Time: Final Showdown. In: MARGARIA, Tiziana; YI, Wang (Hrsg.): *7th International Conference on*

Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001) Bd. 2031, Springer, 2001 (LNCS). – ISBN 3–540–41865–2, S. 1–22 (Zitiert auf Seite 63.)

[VEK00] Vigna, Giovanni; Eckmann, Steve T.; Kemmerer, Richard A.: The STAT Tool Suite. In: *DARPA Information Survivability Conference and Exposition (DISCEX 2000)* Bd. 2, IEEE Computer Society, 2000, S. 46–55 (Zitiert auf Seite 58 und 59.)

[WJ06] Whittle, Jonathan; Jayaraman, Praveen K.: Generating Hierarchical State Machines from Use Case Charts. In: *14th IEEE International Conference on Requirements Engineering (RE 2006)*, 2006, S. 16–25 (Zitiert auf Seite 56.)

[WJ08] Whittle, Jon; Jayaraman, Praveen: MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation. Version: 2008. http://dx.doi.org/10.1007/978-3-540-69073-3_3. In: Giese, Holger (Hrsg.): *Models in Software Engineering* Bd. 5002. DOI. – 10.1007/978–3–540–69073– 3_3. – ISBN 978–3–540–69069–6, S. 16–27 (Zitiert auf Seite 56.)

[WWH08] Whittle, Jonathan; Wijesekera, Duminda; Hartong, Mark: Executable Misuse Cases for Modeling Security Concerns. In: *30th International Conference on Software Engineering (ICSE 2008)*, ACM, 2008. – ISBN 978–1–60558–079–1, S. 121–130 (Zitiert auf Seite 56, 76 und 240.)

CURRICULUM VITAE

Name	Lars Patzina
Geburtsdatum	01. September 1981
Geburtsort	Bad Soden, Deutschland
E-Mail	lars.patzina.fges@gmail.com

2011 - 2013	Wissenschaftlicher Mitarbeiter am Fachgebiet Echtzeitsysteme der TU Darmstadt
2008 - 2011	Promotionsstipendium des Center for Advanced Security Research Darmstadt (CASED)
2008	Dipl.-Ing. Elektro- und Informationstechnik,
	Diplomarbeit: *„Systematische Generierung von Werkzeug-Adaptern aus Metamodellen am Beispiel Matlab/Simulink/-Stateflow“*
2007 -2008	Fachpraktikum und Studienarbeit bei BMW AG, München (10 Monate),
	Studienarbeit: *„Analyse und Definition der Anforderungen an einen Testfallgenerator aus Rhapsody-Modellen“*
2002 - 2008	Studium der Elektro- und Informationstechnik mit Vertiefung Datentechnik an der TU Darmstadt
1994 - 2001	Gymnasium Oberursel, Abitur

Teil VI

ANHANG

A

ANHANG

A.1 Quelltext eines generierten Monitors in C

In diesem Abschnitt ist der generierte C-Code des CARDME-Beispiels aus Abschnitt 8.2 dargestellt. Hierbei werden nur die C-Quellcodedateien gezeigt.

Auflistung A.1 zeigt den generierten Code der CARDME_BasicPath-Signatur. Dieser bildet die Verwaltung der MPN-spezifischen Konzepte, wie die Speicherung der Markierung, die Überprüfung und das Schalten der Transitionen sowie die Auswertung von Bedingungen und die Ausführung von Aktionen, ab.

Auflistung A.1: CARDME_BasicPath.c

```
1  #include <string.h>
2  #include <stdio.h>
3  #include "../include/CARDME_BasicPath.h"
4  #include "../include/Context.h"
5
6  //Helper for initialization of arrays
7  CARDME_BASICPATH_Places CARDME_BASICPATH_initplaces;
8  //Arrays, which contain active places of respective generation
9  CARDME_BASICPATH_Places CARDME_BASICPATH_activePlaces[5];
10 //Indicates which generation is active
11 char CARDME_BASICPATH_activeGenerations[1] = {0};
12
13 //Init arrays
14 void CARDME_BASICPATH_init(){
15   int i;
16   for (i = 0; i < 3; i++) {
17     CARDME_BASICPATH_initplaces.list[i] = 0;
18   }
19   for (i = 0; i < 5; i++) {
20     CARDME_BASICPATH_activePlaces[i] = CARDME_BASICPATH_initplaces;
21   }
22 }
23
24 //Activates generation and sets initials places
25 void CARDME_BASICPATH_newGeneration(unsigned char generation){
26   CARDME_BASICPATH_activeGenerations[generation/8] |= 1<<(generation%8);
27   CARDME_BASICPATH_activePlaces[generation].list[0] |= 1;
28   CARDME_BASICPATH_activePlaces[generation].list[0] |= 2;
29   CARDME_BASICPATH_activePlaces[generation].list[0] |= 4;
30 }
```

```
31
32 //Deactivates generation and its active places
33 void CARDME_BASICPATH_removeGeneration(unsigned char generation){
34  CARDME_BASICPATH_activeGenerations[generation/8] &= ~(1<<(generation%8));

35  CARDME_BASICPATH_activePlaces[generation] = CARDME_BASICPATH_initplaces;
36 }
37
38 //Transitions
39 int CARDME_BASICPATH_evaluateEvent_EPSILON(unsigned char generation){...}
40
41 int CARDME_BASICPATH_evaluateEvent_SEND_INITREQ(unsigned char generation){
42  int result = 0;
43  if (((CARDME_BASICPATH_activePlaces[generation].list[0]&1) == 1)){
44   //Store changes to the marking
45   deletedToken[nextChangeDelete++] = 0;
46   createdToken[nextChangeCreate++] = 3;
47   createdToken[nextChangeCreate++] = 4;
48   //Action
49   //fprintf("CARDME_BASICPATH generation %d transition 7 fired\n",
         generation);
50   result = 1;
51  }
52  return result;
53 }
54
55 int CARDME_BASICPATH_evaluateEvent_RECV_INITREQ(unsigned char generation){
56  int result = 0;
57  if (((CARDME_BASICPATH_activePlaces[generation].list[0]&8) == 8)
58   && ((CARDME_BASICPATH_activePlaces[generation].list[0]&2) == 2)){
59   deletedToken[nextChangeDelete++] = 3;
60   deletedToken[nextChangeDelete++] = 1;
61   createdToken[nextChangeCreate++] = 5;
62   //fprintf("CARDME_BASICPATH generation %d transition 8 fired\n",
         generation);
63   result = 1;
64  }
65  return result;
66 }
67 int CARDME_BASICPATH_evaluateEvent_SEND_INITRESP(unsigned char generation)
       {...}
68 int CARDME_BASICPATH_evaluateEvent_RECV_INITRESP(unsigned char generation)
       {...}
69 int CARDME_BASICPATH_evaluateEvent_SEND_CLOSINGREQUEST(unsigned char
       generation){...}
70 int CARDME_BASICPATH_evaluateEvent_RECV_CLOSINGREQUEST(unsigned char
       generation){...}
71
72
73 functionPtrCARDME_BASICPATH CARDME_BASICPATH_functionArray[
       EVENT_LAST_ELEMENT] = {
74  &CARDME_BASICPATH_evaluateEvent_EPSILON,
```

```
75   &CARDME_BASICPATH_evaluateEvent_SEND_INITREQ,
76   &CARDME_BASICPATH_evaluateEvent_RECV_INITREQ,
77   &CARDME_BASICPATH_evaluateEvent_SEND_INITRESP,
78   &CARDME_BASICPATH_evaluateEvent_RECV_INITRESP,
79   &0,
80   &0,
81   &CARDME_BASICPATH_evaluateEvent_SEND_CLOSINGREQUEST,
82   &CARDME_BASICPATH_evaluateEvent_RECV_CLOSINGREQUEST
83 };
84
85 //Evaluate an event with a context on the MPN
86 char CARDME_BASICPATH_evaluateEvent(unsigned char event, unsigned char
       generation){
87   char result = 0;
88   result = (*CARDME_BASICPATH_functionArray[event])(generation) || result;
89   if (result) CARDME_BASICPATH_applyChanges(generation);
90   return result;
91 }
92
93 //Check antecedent places
94 char CARDME_BASICPATH_checkPrecondition(unsigned char generation, unsigned
        char event){
95   if((CARDME_BASICPATH_activePlaces[generation].list[0]&4) == 4){
96     fprintf(stderr,"TERMINATED; CARDME_BASICPATH ended in precondition;
          Generation %d; event %d\n",generation,event);
97     return 1;
98   }else{
99     fprintf(stderr,"FAILURE; No match in CARDME_BASICPATH; Generation %d
         ; event %d\n",generation,event);
100    return 2;
101  }
102 }
103
104 //Check terminal places
105 char CARDME_BASICPATH_checkPlaces(unsigned char generation, unsigned char
       event){
106   if (0){
107     fprintf(stderr,"FAILURE; CARDME_BASICPATH reached failure place;
          Generation %d; event %d\n",generation,event);
108     return 2;
109   } else
110   if (((CARDME_BASICPATH_activePlaces[generation].list[1]&128) == 128)){
111     fprintf(stderr,"TERMINATED; CARDME_BASICPATH reached end place;
          Generation %d; event %d\n",generation,event);
112     return 1;
113   }
114   return 0;
115 }
```

Auflistung A.2 zeigt die Kontrollstruktur, die die Schnittstellen des Monitors zur Verfügung stellt, die Speicherung der mit dem Ereignis übergebenen Parameterwerte sowie die Zwischenspeicherung der Änderungen an der MPN-Markierung übernimmt und die einzelnen MPNs des Monitors ansteuert.

Auflistung A.2: Controller.c

```
#include "../include/Controller.h"
#include <stdio.h>
#include "../include/Context.h"

struct cntxt contextTemp = {{0,0,0,0,0}};
int deletedToken[3];
int createdToken[3];
int nextChangeDelete = 0;
int nextChangeCreate = 0;


void MPN_init(){
  CARDME_BASICPATH_init();
}

// Event specific interface
void dispatchEvent_RECV_INITRESPONSE(IRData*param, int gen){
  contextTemp.ir_param[gen]=(IRData)*param;
  CARDME_BASICPATH_dispatchEvent(EVENT_RECV_INITRES, gen);
}
...

char dispatchEvent(unsigned char event, unsigned char generation){
  char cardme_basicpathFired = 0;

  char evaluateEpsilonTransition;
  char monitorResult;

  // Evaluate CARDME_BASICPATH signature
  if(CARDME_BASICPATH_functionArray[event] != 0){
    if(!(((CARDME_BASICPATH_activeGenerations[generation/8]) & (1<<(
        generation%8))) == (1<<(generation%8)))){
      CARDME_BASICPATH_newGeneration(generation);
    }
    cardme_basicpathFired = CARDME_BASICPATH_evaluateEvent(event,
        generation);
    evaluateEpsilonTransition = 1;
    while (evaluateEpsilonTransition) {
      evaluateEpsilonTransition = CARDME_BASICPATH_evaluateEvent(
          EVENT_EPSILON, generation);
      if (evaluateEpsilonTransition) cardme_basicpathFired = 1;
    }
    monitorResult = evaluateMonitoringResult(generation,
        cardme_basicpathFired, &CARDME_BASICPATH_checkPrecondition, &
        CARDME_BASICPATH_checkPlaces, event);
    if ((monitorResult == 1) || (monitorResult == 2)){
      CARDME_BASICPATH_removeGeneration(generation);
    }
  } else {
    fprintf(stderr,"Event %d not in CARDME_BASICPATH\n",event);
  }
```

```
47   return (cardme_basicpathFired);
48 }
49
50 char evaluateMonitoringResult(unsigned char generation, char mpnFired,
       char (*checkPreconditionPointer)(unsigned char, unsigned char), char
       (*checkPlacesPointer)(unsigned char, unsigned char), unsigned char
       event){
51   char result;
52   if(!mpnFired){
53     result = checkPreconditionPointer(generation, event);
54     return result;
55   }
56   result = checkPlacesPointer(generation, event);
57   return result;
58 }
59
60 // Apply changes to the marking
61 void CARDME_BASICPATH_applyChanges(unsigned char generation){
62   int i;
63   for(i = 0; i < nextChangeDelete; i++){
64     CARDME_BASICPATH_activePlaces[generation].list[(deletedToken[i])/8] &=
         ~(1<<(deletedToken[i]%8));
65   }
66   for(i = 0; i < nextChangeCreate; i++){
67     CARDME_BASICPATH_activePlaces[generation].list[(createdToken[i])/8] |=
         1<<(createdToken[i]%8);
68   }
69   nextChangeDelete = 0;
70   nextChangeCreate = 0;
71 }
```